Earth Sounding Analysis: Processing versus Inversion

Jon F. Claerbout

Directory

Last Revision Date: February 19, 2008

Table of Contents

FREEWARE, COPYRIGHT, LICENSE, AND CRED

ITS

This disk contains freeware from many authors. Freeware is software you can copy and give away. But it is restricted in other ways. Please see author's copyrights and "public licenses" along with their programs.

As you saw on the copyright page and will find in the electronic files, my electronic book is copyrighted. However, the programs I wrote that display the book and its figures are available to you under the GNU public license (see below). I have signed over copyright of the book text to a traditional book publisher[1]; however, I did not grant them the electronic rights. I license you, the general public, to make electronic copies of the entire book provided that you do not remove or alter this licensing statement. Please respect the publisher's legal rights and do not make paper copies from your copy of the electronic book.

We (you and I) are indebted to many people who have generously contributed software to the public good. I'll mention here only those outside the Stanford Uni-

---

[1] Blackwell Scientific Publications, 3 Cambridge Center, Cambridge, MA 02142

versity research group whose contributions are widely used and on which we deeply depend:

TEX	Don Knuth, Stanford University
LATEX	Leslie Lamport, Stanford Research Institute
ratfor77	Ozan Yigit, Arizona, and Wes Bauske, IBM
ratfor90	Bob Clapp
dvips	Tomas Rokicki, Stanford University

I feel sure the list of valuable contributors is much longer. I am afraid I may have overlooked the names of some, and others have modestly omitted leaving their name and copyright.

this program; if not, write to the Free Software Foundation, Inc., 675 Massachusetts Ave., Cambridge, MA 02139, USA.

PREFACE TO THE ELECTRONIC BOOK

Reproducibility

Each figure caption is followed by an [R] or an [NR] which denotes Reproducible or Not Reproducible. To actually burn and rebuild the illustrations you will need to have "seplib" installed at your site.

SEP software

Contained on the CD-ROM distribution are two interactive programs, `ed1D` and `Zplane`. I originally wrote these programs in Sunview, an interactive software development platform from Sun Microsystems. Fortunately, Steve Cole converted them to the X Window system, using the X toolkit and Xlib graphics, so they are now available on machines from many manufacturers. Unfortunately, in 1998, we do not have them compiled for our main machines at SEP, linux PC's and SGI.

Acknowledgement

This textbook itself was updated in minor ways since the 1991 CD-ROM was produced. The electronic document, however, is greatly enhanced through systems improvements made by Martin Karrenbach, Steve Cole, and Dave Nichols. Most of the features described in this preface were absent or incomplete in 1991.

A note to the reader

In many branches of engineering and science there is a substantial computational element. Earth-imaging seismology is one of these. In taking up computational problems we should abandon books, journals, and reports and replace them with electronic documents that can be used to recreate any print document, including its figures, from its underlying data and computations. Today, few published results are reproducible in any practical sense. To verify them requires almost as much effort as it took to create them originally. After a time, authors are often unable to reproduce

their own results! For these reasons, many people ignore most of the literature. In the past this scandalous waste of time and energy may have been justified by the high cost and incompatibility of data-processing machines. But with standards for Fortran, C, UNIX,[2] LaTeX, Postscript,[3] Xwindow,[4] CD-ROM, and shirt-pocket-sized two-gigabyte tapes, there is no longer any excuse for nonreproducible research. It is time to plunge into this new era.

This paper book of 300 pages presents theory implemented by sixty subroutines, all included in the book, which in turn made the book's 150 figures. Behind the paper book are about seventy figure-making directories, a large volume of Stanford Exploration Project utility software, and some real datasets you can experiment with if you have access to the electronic form of the book. I made nearly all of the figures myself. Even without the electronic book, from the printed subroutines only, you should be able to produce results similar to mine and, beyond this, use the subroutines in your own work.

---

[2] AT&T

[3] Adobe Systems, Inc.

[4] Massachusetts Institute of Technology

If you have access to the electronic form of this book, you can read it from a computer screen and press the buttons in the figure captions to rebuild and redisplay the figures. Some of the figures are in color, some are interactive, and some are movies. But this is not the goal of the electronic book. Its goal is to enable you to reproduce all my figures with reasonable ease, to change parameters, to try other datasets, to modify the programs, and to experiment with the theoretical concepts.

I could have written the programs in this book in vanilla Fortran or C and suffered the verbosity and blemishes of these languages. Instead I chose to write the programs in a Fortran dialect that, like mathematics, is especially suited to the exposition of technical concepts. At Stanford we translate these programs to Fortran automatically by passing them first through a home-made processor named `sat`, which overcomes Fortran's inability to create temporary arrays of arbitrary dimension, and second through AT&T's `Ratfor` (Rational Fortran) preprocessor. If you wish, a program called `f2c`, freely available from AT&T, will translate the Fortran to C.

My goal in writing the programs in this book was not to write the best possible code with the clearest possible definitions of inputs and outputs. That would be a laudable goal for a reference work such as *Numerical Recipes* (Press et al.). Instead,

I present a full mathematical analysis with simple and concise code along with meaningful examples of its use. I use the code as others might use pseudocode—to exemplify and clarify the concepts. These programs, which also made the book's figures, are not guaranteed to be free of errors. Since the word processor and the compiler got the programs from the same place, however, there can be no errors of transcription.

Why another book?

I decided to write this book for five reasons. First, seismologists and explorationists, as well as many others in science and engineering, share the ability to synthesize the data implied by any physical model. They have much to learn, however, about "inverse modeling," that is, given the data, the process of finding the most appropriate model. This task is also called "model fitting," words that hardly hint at the ingenuity that can be brought to bear. There is no shortage of books about least-squares regression, also called "inversion." These books provide a wide range of mathematical concepts—often too many, and often with no real examples. In my teaching and research I have found that people are mostly limited, not by lack of

theory, but by failure to recognize where elementary theory is applicable. To cite an example, "zero padding" is a tiny bit of technology used nearly everywhere, but few people seem to recognize its mathematical adjoint and so are ill prepared to invoke $(\mathbf{A}'\mathbf{A})^{-1}\mathbf{A}'\mathbf{d}$ or set up a conjugate-gradient optimization. Therefore, a keystone chapter of this book shows how adjoint operators can be a simple byproduct of any modeling operator. In summary, the first reason I am writing this book is to illuminate the concept of "adjoint operator" by examining many examples.

The second reason for writing the book is to present the conjugate-gradient optimization algorithm in the framework of many examples. The inversion theory found in most textbooks, while appearing generally applicable, really is not. Matrix inversions and singular-value decompositions are limited in practice to matrices of dimension less than about one thousand. But practical problems come in all dimensions, from one to many millions (when the operator is a multidimensional wave equation). Conjugate-gradient methods—only beginning to find routine use in geophysics—point the way to overcoming this dimensionality problem. As in the case of inversion, many books describe the conjugate-gradient method, but the method is not an end in itself. The heart of this book is the many examples that are set up in the conjugate-gradient framework. Setting up the problems is where inge-

nuity is required. Solving them is almost routine—especially using the subroutine library in this book.

My third reason for writing the book is much narrower. Seismogram deconvolution—by far the largest use of geophysical inversion theory—is in a state of disarray. I see serious discrepancies between theory and practice (as do others). I believe the disarray stems from a tendency to cling to a large body of old quasi-analytic theory. This theory had a place in my first book, *Fundamentals of Geophysical Data Processing*, but I have omitted it here. It can be replaced by a simpler and less restrictive numerical approach.

My fourth reason for writing the book is to illuminate the place of missing seismograms. Much data is analyzed assuming that missing data is equivalent to zero-valued data. I show how to handle the problem in a better way.

Finally, I am writing this book to illuminate the subtitle, *Processing versus Inversion*, by which I mean the conflicting approaches of practitioners and academics to earth soundings analysis.

This book should be readable by anyone with a bachelor's degree in engineering or physical science. It is easier for students to use than my first book, *Fundamentals of Geophysical Data Processing*. It is written at about the level of my second book,

Imaging the Earth's Interior.

Organization

Page numbers impose a one-dimensional organization on any book. I placed basic things early in the book, important things in the middle of the book, and theoretical, less frequently used things at the end. Within chapters and sections, this book answers the questions *what* and *how* before it answers *why*. I chose to avoid a strictly logical organization because that would result in too much math at the beginning and too long a delay before the reader encountered applications. Thus, you may read about a single subject at different points in the book. It is not organized like an encyclopedia but is ordered for learning. For reference, please make use of the index.

Dedication

I am especially indebted to all those students who complained that I did not give enough examples in my classes. (Even with access to the book in its present form,

they still complain about this, so there is work left to do.)

Acknowledgements

In this book, as in my previous book, *Imaging the Earth's Interior*, I owe a great deal to the many students at the Stanford Exploration Project. The local computing environment from my previous book is still a benefit, and for this I thank Stew Levin, Dave Hale, and Richard Ottolini. In preparing this book I am specially indebted to Joe Dellinger for his development of the intermediate graphics language `vplot` that I used for all the figures. I am also very grateful to Kamal Al-Yahya for converting my thinking from the `troff` typesetting language to LaTeX, for setting up the initial structure of the book in LaTeX, and for the conversion program `tr2tex` (which he made publicly available and which is already widely used) that I needed to salvage my older materials. I have benefited from helpful suggestions by Bill Harlan and Gilles Darche. Biondo Biondi, Dave Nichols, and I developed the `saw` and **`sat`** Fortran preprocessors. Dave Nichols found the `cake` document maintenance system, adapted it to our local needs, and taught us all how to use it, thereby giving us a machine-independent software environment. Martin Karren-

bach implemented the caption pushbuttons and had many ideas for integrating the paper book with the interactive book. Steve Cole adapted `vplot` to Postscript and X, redesigned `xtex` for Sun computers, and generously offered assistance in all areas. Mark Chackerian prepared the first CD-ROM of the electronic book and gave assistance with LaTeX. I am thankful to my editor, JoAnn Heydron, for careful work, to Joe Stefani for detecting typographical errors in mathematics, and to Diane Lau for office assistance.

Jon Claerbout
Stanford University
most final revisions in 1992
(electronic media keep changing)

Introduction

Prospecting for petroleum is a four-step process: (1) echo soundings are recorded; (2) they are analyzed for reflections; (3) the reflections are interpreted as a geological model; and (4) the prospect is tested by drilling. The first two stages, data acquisition and analysis, are on a worldwide basis a multibillion-dollar-per-year activity. This book describes only the echo soundings analysis. Together with my 1985 book, *Imaging the Earth's Interior*, it provides a complete introduction to echo soundings analysis.

The subtitle of this book, *Processing versus Inversion*, places the book equidistant from two approaches, one generally practical and industrial and the other generally theoretical and academic. This book shows how the two approaches are related and contribute to each other.

Adjoint processing defined

"Data **processing**" in earth soundings analysis could mean anything anybody does to seismic data. A narrower definition is those processes that are routinely applied in industry, such as those described in Oz Yilmaz's book, *Seismic Data Processing*. As we will see in chapter 5 of this book, much of echo soundings analysis can be interpreted as the *adjoint* of seismogram modeling. Here we use the word "**adjoint**" in the mathematical sense to mean the complex conjugate of the matrix transpose. Not all processes can be accurately characterized as the adjoint to seismogram modeling, but many can, including normal moveout, stacking, migration, dip moveout, and more. Since these are the heavyweights of the industry, the simple word "processing" can almost be understood to stand for "processing by adjoint modeling." As we will see, such processing applied to perfect data generally gives an imperfect

result. This imperfection leads thoughtful people to the concept of inversion.

Inversion defined

Principles of physics allow us to calculate synthetic data from earth models. Such calculations are said to solve "forward" problems. In real life we are generally interested in the reverse calculation, i.e., computing earth models from data. This reverse calculation is called "**inversion**." The word "inversion" is derived from "matrix inversion." Despite its association with the well-known and well-defined mathematical task of matrix inversion, echo sounding inversion is not simple and is often ill defined. Inversion promises to give us an earth model from our data despite the likelihood that our data is inaccurate and incomplete. This promise goes too far. Inversion applied to perfect data, however, can give a perfect result, which makes inversion more appealing academically than processing by adjoint modeling.

Processing versus inversion

Practical people often regard inversion theorists with suspicion, much as one might regard those gripped by an exotic religion. There is not one theory of inversion of seismic data, but many—maybe more theories than theoreticians. The inventors of these theories are all ingenious, and some are illustrious, but many ignore the others' work. How can this be science or engineering? The diversity of viewpoint arises from the many practical problems that need to be solved, from the various ways that noise can be modeled, from the incompleteness of data, and above all, from the many approaches to simplifying the underlying model.

Practitioners too are a diverse group of shrewd and talented people, many illustrious in their highly competitive industry. As a group they have the advantage of the "real world" as a helpful arbitrator. Why do they prefer a adjoint operator when the correct answer, almost by definition, stems from the inverse? Adjoint processing requires no more than the data one has actually collected. It requires no noise model, never uses divisions so cannot divide by zero, and often uses only additions (no subtractions) so cannot amplify small differences. Anyone taking the first step beyond adjoint processing loses these supports. Unfortunately, adjoint operators handle missing data as if it were zero-valued data. This is obviously wrong and is

known to limit resolution.

> I hope to illuminate the gaps between theory and practice which are the heart and soul of exploration seismology, as they are of any living science.

Fortunately there is a middle way between adjoint processing and inversion, and this book is a guide to it. Adjoint processing and inversion stand at opposite ends of the spectrum of philosophies of data processing, but, as we will see in chapter 6, adjoint processing is also the *first* step of inversion. Whether the *second* and any subsequent steps are worthwhile depends on circumstances.

The theme of this book is not developed in an abstract way but instead is drawn from the details of many examples: normal moveout, stacking, velocity analysis, several kinds of migration, missing data, tomography, deconvolution, and weighted deconvolution. Knowing how processing relates to inversion suggests different opportunities in each case.

Linear inverse theory

In mathematical statistics is a well-established theory called "**linear inverse theory**." "**Geophysical inverse theory**" is similar, with the additions that (1) variables can be sample points from a continuum, and (2) physical problems are often intractable without linearization. Once I imagined a book that would derive techniques used in industry from general geophysical inverse theory. After thirty years of experience I can report to you that very few techniques in routine practical use arise directly from the general theory! There are many reasons for this, and I have chosen to sprinkle them throughout discussion of the applications themselves rather than attempt a revision to the general theory. I summarize here as follows: the computing requirements of the general theory are typically unrealistic since they are proportional to the cube of a huge number of variables, which are sample values representing a continuum. Equally important, the great diversity of spatial and temporal aspects of data and residuals (statistical nonstationarity) is impractical to characterize in general terms.

Our route

Centrally, this book teaches how to recognize adjoint operators in physical processes (chapter 5), and how to use those adjoints in model fitting (inversion) using least-squares optimization and the technique of conjugate gradients (chapter 6).

First, however, we review convolution and spectra (chapter 1) discrete Fourier transforms (chapter 9), and causality and the complex $Z = e^{i\omega}$ plane (chapter 3), where poles are the mathematically forbidden points of zero division. In chapter 3 we travel widely, from the heaven of theoretically perfect results through a life of practical results including poor results, sinking to the purgatory of instability, and finally arriving at the "big bang" of zero division. Chapter 4 is a collection of solved problems with a *single unknown* that illustrates the pitfalls and opportunities that arise from weighting functions, zero division, and nonstationarity. Thus we are prepared for the keystone chapter, chapter 5, where we learn to recognize the relation of the linear operators we studied in chapters 1–3 to their adjoints, and to see how computation of these adjoints is a straightforward adjunct to direct computation. Also included in chapter 5 are interpolation, smoothing, and most of the many operators that populate the world of exploration seismology. Thus further prepared, we pass easily through the central theoretical concepts of least-squares optimization,

basic NMO stack, and deconvolution applications in chapter 6.

In chapter 7 we see the formulation and solution of many problems in time-series analysis, prediction, and interpolation and learn more about mathematical formulations that control stability. Chapter 8 shows how missing data can be estimated. Of particular interest is a nonstationary world model where, locally in time and space, the wave field fits the model of a small number of plane waves. Here we find "magical" results: data that is apparently undersampled (spatially aliased) is recovered.

Hyperbolas are the reflection seismologist's delight. My book *Imaging the Earth's Interior* could almost have been named *Hyperbolas and the Earth.* That book includes many techniques for representing and deforming hyperbolas, especially using various representations of the wave equation. Here I repeat a minimal part of that lore in chapter **??**. My goal is now to marry hyperbolas to the conjugate-gradient model-fitting theme of this book.

Having covered a wide range of practical problems, we turn at last to more theoretical ones: spectra and phase (chapter 10), and sample spectra of random numbers (chapter 11). I have begun revising three theoretical chapters from my first book, *Fundamentals of Geophysical Data Processing* (hereinafter referred to

as **FGDP**), which is still in print. Since these revisions are not yet very extensive, I am excluding the revised chapters from the current copy of this book. (My 1985 book, *Imaging the Earth's Interior* (hereinafter referred to as **IEI**), deserves revision in the light of the conjugacy methods developed here, but that too lies in the future.)

Finally, every academic is entitled to some idiosyncrasies, and I find Jensen inequalities fascinating. These have an unproved relationship to practical echo analysis, but I include them anyway in a brief concluding chapter.

0.1. References

Claerbout, J.F., 1985, Fundamentals of geophysical data processing: Blackwell Scientific Publications.

Claerbout, J.F., 1985, Imaging the earth's interior: Blackwell Scientific Publications.

Press, W.H. et al., 1989, Numerical recipes: the art of scientific computing: Cambridge University Press.

Yilmaz, O., 1987, Seismic data processing: Society of Exploration Geophysicists.

Chapter 1

Convolution and Spectra

In human events, the word “convoluted” implies complexity. In science and engineering, “convolution” refers to a combining equation for signals, waves, or images. Although the combination may be complex, the **convolution** equation is an

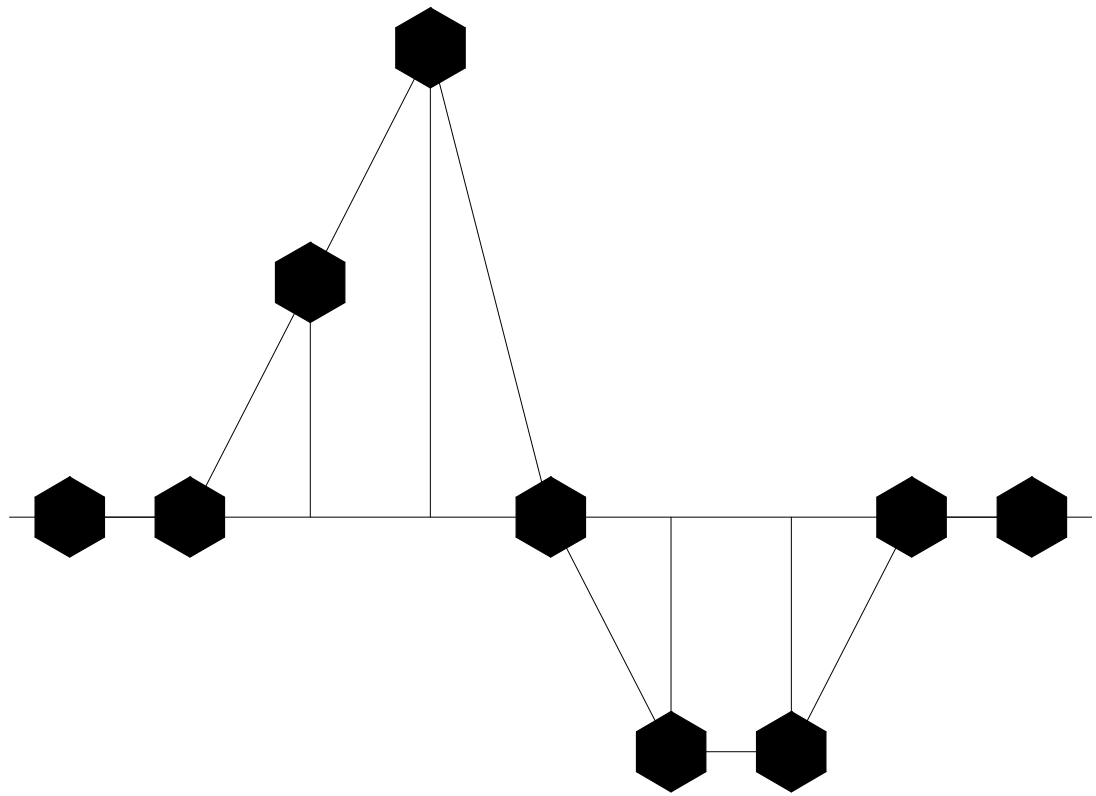
elementary one, ideally suited to be presented at the beginning of my long book on dissecting observations. Intimately connected to convolution are the concepts of pure tones and Fourier analysis.

Time and space are ordinarily thought of as continuous, but for the purposes of computer analysis we must discretize these axes. This is also called "**sampling**" or "**digitizing**." You might worry that discretization is a practical evil that muddies all later theoretical analysis. Actually, physical concepts have representations that are exact in the world of discrete mathematics. In the first part of this book I will review basic concepts of convolution, spectra, and causality, while using and teaching techniques of discrete mathematics. By the time we finish with chapter 3, I think you will agree with me that many subtle concepts are easier in the discrete world than in the continuum.

1.1. SAMPLED DATA AND Z-TRANSFORMS

Consider the idealized and simplified signal in Figure 1.1. To analyze such an observed signal in a computer, it is necessary to approximate it in some way by a list of numbers. The usual way to do this is to evaluate or observe $b(t)$ at a uniform spac-

Figure 1.1: A continuous signal sampled at uniform time intervals. (Press button for trivial interaction with plot.) cs-triv1 [ER]



ing of points in time, call this discretized signal b_t. For Figure 1.1, such a discrete approximation to the continuous function could be denoted by the vector

$$b_t \quad = \quad (\ldots\ 0, 0, 1, 2, 0, -1, -1, 0, 0, \ldots) \tag{1.1}$$

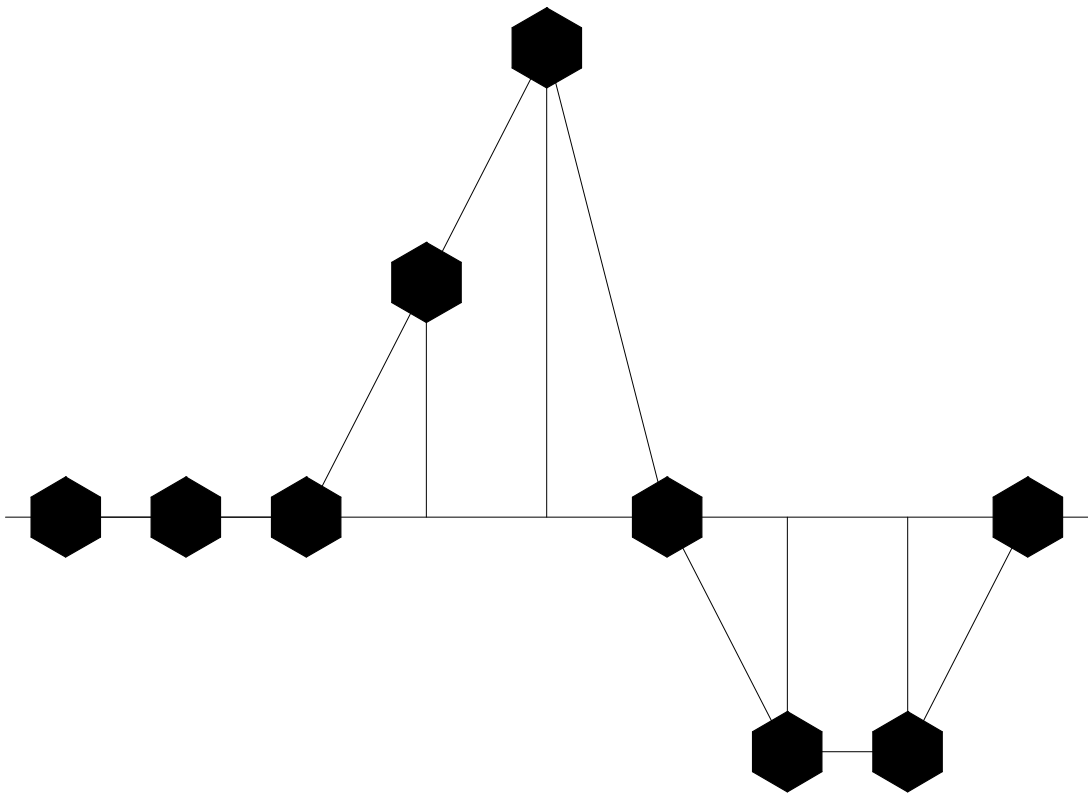
Naturally, if time points were closer together, the approximation would be more accurate. What we have done, then, is represent a signal by an abstract n-dimensional vector.

Another way to represent a signal is as a polynomial, where the *coefficients* of the polynomial represent the value of b_t at successive times. For example,

$$B(Z) \quad = \quad 1 + 2Z + 0Z^2 - Z^3 - Z^4 \tag{1.2}$$

This polynomial is called a "Z**-transform**." What is the meaning of Z here? Z should not take on some numerical value; it is instead the **unit-delay operator**. For example, the coefficients of $ZB(Z) = Z + 2Z^2 - Z^4 - Z^5$ are plotted in Figure 1.2. Figure 1.2 shows the same waveform as Figure 1.1, but now the waveform has been delayed. So the signal b_t is delayed n time units by multiplying $B(Z)$ by Z^n. The delay operator Z is important in analyzing waves simply because waves take a certain amount of time to move from place to place.

Figure 1.2: The coefficients of $ZB(Z)$ are the shifted version of the coefficients of $B(Z)$. cs-triv2 [ER]

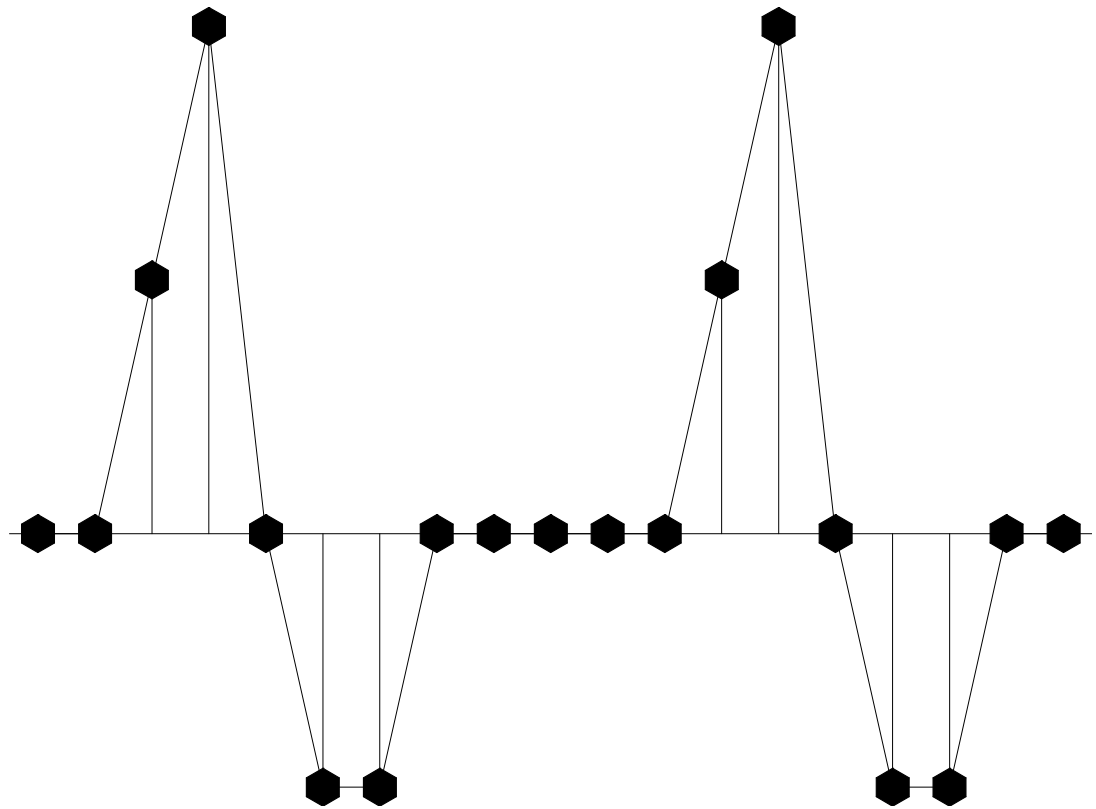


Another value of the delay operator is that it may be used to build up more complicated signals from simpler ones. Suppose b_t represents the acoustic pressure function or the seismogram observed after a distant explosion. Then b_t is called the "**impulse response**." If another explosion occurred at $t = 10$ time units after the first, we would expect the pressure function $y(t)$ depicted in Figure 1.3. In terms of Z-transforms, this pressure function would be expressed as $Y(Z) = B(Z) + Z^{10}B(Z)$.

1.1.1. Linear superposition

If the first explosion were followed by an implosion of half-strength, we would have $B(Z) - \frac{1}{2}Z^{10}B(Z)$. If pulses overlapped one another in time (as would be the case if $B(Z)$ had degree greater than 10), the waveforms would simply add together in the region of overlap. The supposition that they would just add together without any interaction is called the "**linearity**" property. In seismology we find that—although the earth is a heterogeneous conglomeration of rocks of different shapes and types—when seismic waves travel through the earth, they do not interfere with one another. They satisfy linear **superposition**. The plague of **nonlinearity** arises from large

Figure 1.3: Response to two explosions. cs-triv3 [ER]



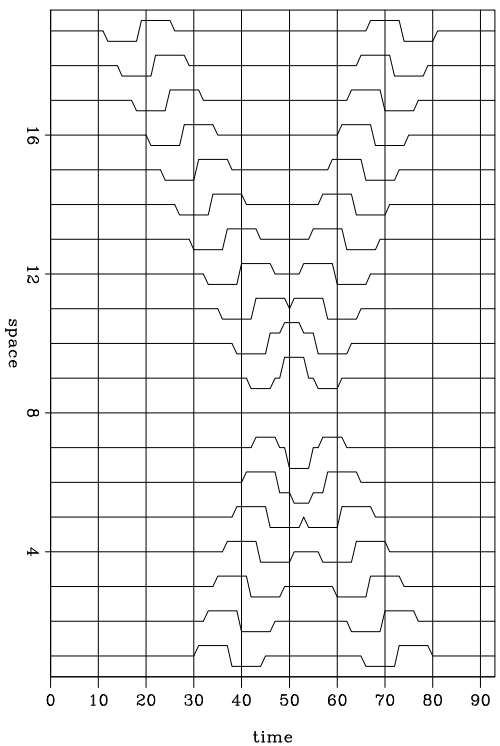
amplitude disturbances. Nonlinearity is a dominating feature in hydrodynamics, where flow velocities are a noticeable fraction of the wave velocity. Nonlinearity is absent from reflection seismology except within a few meters from the source. Nonlinearity does not arise from geometrical complications in the propagation path. An example of two **plane wave**s superposing is shown in Figure 1.4.

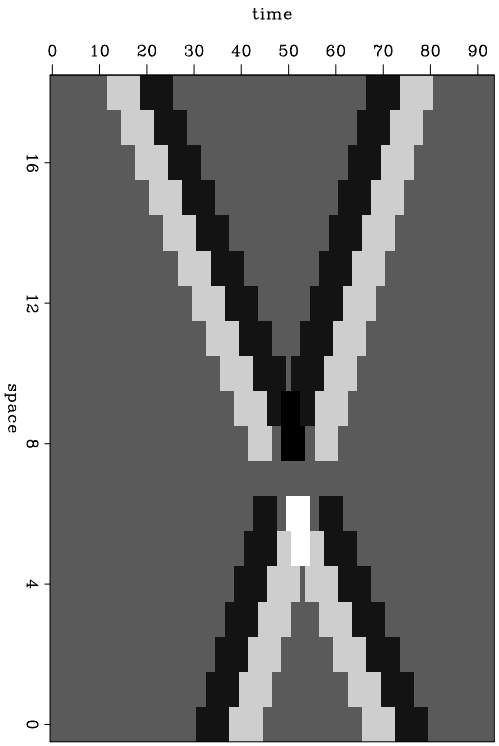
1.1.2. Convolution with Z-transform

Now suppose there was an explosion at $t = 0$, a half-strength implosion at $t = 1$, and another, quarter-strength explosion at $t = 3$. This sequence of events determines a "source" time series, $x_t = (1, -\frac{1}{2}, 0, \frac{1}{4})$. The Z-transform of the source is $X(Z) = 1 - \frac{1}{2}Z + \frac{1}{4}Z^3$. The observed y_t for this sequence of explosions and implosions through the seismometer has a Z-transform $Y(Z)$, given by

$$\begin{aligned} Y(Z) &= B(Z) - \frac{Z}{2}B(Z) + \frac{Z^3}{4}B(Z) \\ &= \left(1 - \frac{Z}{2} + \frac{Z^3}{4}\right) B(Z) \end{aligned}$$

Figure 1.4: Crossing plane waves superposing viewed on the left as “wiggle traces” and on the right as “raster.” cs-super [ER]





$$= \qquad X(Z)\,B(Z) \tag{1.3}$$

The last equation shows **polynomial multiplication** as the underlying basis of time-invariant linear-system theory, namely that the output $Y(Z)$ can be expressed as the input $X(Z)$ times the impulse-response **filter** $B(Z)$. When signal values are insignificant except in a "small" region on the time axis, the signals are called "**wavelet**s."

There are many examples of linear systems. The one of most interest to us is wave propagation in the earth. A simpler example, around which a vast literature exists, is electronic filters. A **cascade of filters** is formed by taking the output of one filter and plugging it into the input of another. Suppose we have two linear filters characterized by $B(Z)$ and $C(Z)$. Then the question arises, illustrated in Figure 1.5, as to whether the two combined filters are equivalent.

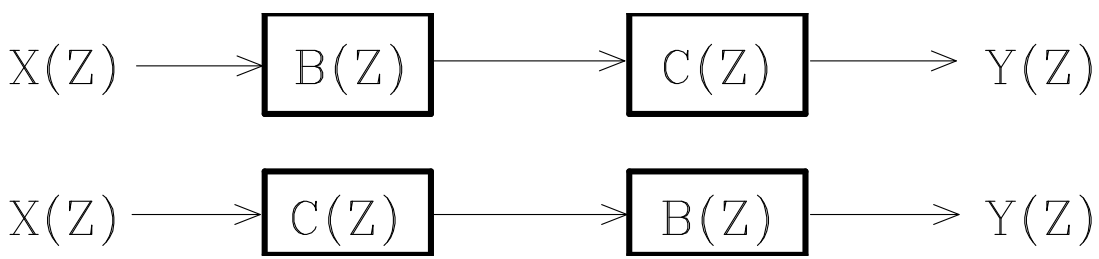


Figure 1.5: Two equivalent filtering systems. cs-commute [NR]

The use of Z-transforms makes it obvious that these two systems are equivalent,

since products of polynomials **commute**, i.e.,

$$\begin{array}{ccccccc} Y_1(Z) & = & [X(Z)B(Z)]C(Z) & = & XBC & & \\ Y_2(Z) & = & [X(Z)C(Z)]B(Z) & = & XCB & = & XBC \end{array} \qquad (1.4)$$

1.1.3. Dissecting systems by factoring

Consider a system with an impulse response $(2,-1,-1)$. Its Z-transform is $B(Z) = 2 - Z - Z^2$. This polynomial can be **factor**ed into $2 - Z - Z^2 = (2+Z)(1-Z)$. Thus our original filter could be thought of as a cascade of two filters, $(2,1)$ and $(1,-1)$. Either of the two filters could be applied first and the other second: the output would be the same. Since any polynomial can be factored, any impulse response can be simulated by a cascade of two-term filters (impulse responses whose Z-transforms are linear in Z).

1.1.4. Convolution equation and program

What do we actually do in a computer when we multiply two Z-transforms together? The filter $2+Z$ would be represented in a computer by the storage in memory of

the coefficients $(2,1)$. Likewise, for $1-Z$, the numbers $(1,-1)$ would be stored. The polynomial multiplication program should take these inputs and produce the sequence $(2,-1,-1)$. Let us see how the computation proceeds in a general case, say

$$X(Z)\,B(Z) \quad = \quad Y(Z) \tag{1.5}$$

$$(x_0+x_1Z+x_2Z^2+\cdots)(b_0+b_1Z+b_2Z^2) \quad = \quad y_0+y_1Z+y_2Z^2+\cdots \tag{1.6}$$

Identifying coefficients of successive powers of Z, we get

$$\begin{aligned} y_0 &= x_0b_0 \\ y_1 &= x_1b_0+x_0b_1 \\ y_2 &= x_2b_0+x_1b_1+x_0b_2 \\ y_3 &= x_3b_0+x_2b_1+x_1b_2 \\ y_4 &= x_4b_0+x_3b_1+x_2b_2 \\ &= \cdots\cdots\cdots\cdots\cdots \end{aligned} \tag{1.7}$$

In matrix form this looks like

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} = \begin{bmatrix} x_0 & 0 & 0 \\ x_1 & x_0 & 0 \\ x_2 & x_1 & x_0 \\ x_3 & x_2 & x_1 \\ x_4 & x_3 & x_2 \\ 0 & x_4 & x_3 \\ 0 & 0 & x_4 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} \tag{1.8}$$

The following equation, called the "convolution equation," carries the spirit of the group shown in (1.7):

$$y_k = \sum_{i=0}^{N_b} x_{k-i} b_i \tag{1.9}$$

To be correct in detail when we associate equation (1.9) with the group (1.7), we should also assert that either the input x_k vanishes before $k = 0$ or N_b must be adjusted so that the sum does not extend before x_0. These end conditions are expressed

more conveniently by defining $j = k - i$ in equation (1.9) and eliminating k getting

$$y_{j+i} = \sum_{i=0}^{N_b} x_j b_i \tag{1.10}$$

A convolution program based on equation (1.10) including end effects on both ends, is `convolve()`. convolve Some details of the Ratfor programming language are given in an appendix, along with the subroutine `zero()` /prog:zero, which erases the space for the output.

1.1.5. Negative time

Notice that $X(Z)$ and $Y(Z)$ need not strictly be polynomials; they may contain both positive and negative powers of Z, such as

$$X(Z) = \cdots + \frac{x_{-2}}{Z^2} + \frac{x_{-1}}{Z} + x_0 + x_1 Z + \cdots \tag{1.11}$$

$$Y(Z) = \cdots + \frac{y_{-2}}{Z^2} + \frac{y_{-1}}{Z} + y_0 + y_1 Z + \cdots \tag{1.12}$$

```
#       convolution:    Y(Z) = X(Z) * B(Z)
#
subroutine convolve( nb, bb, nx, xx, yy )
integer nb      # number of coefficients in filter
integer nx      # number of coefficients in input
                # number of coefficients in output will be nx+nb-1
real    bb(nb)  # filter coefficients
real    xx(nx)  # input trace
real    yy(1)   # output trace
integer ib, ix, iy, ny
ny = nx + nb -1
call null( yy, ny)
do ib= 1, nb
        do ix= 1, nx
                yy( ix+ib-1) = yy( ix+ib-1) + xx(ix) * bb(ib)
return; end
```

Back

The negative powers of Z in $X(Z)$ and $Y(Z)$ show that the *data* is defined before $t = 0$. The effect of using negative powers of Z in the *filter* is different. Inspection of (1.9) shows that the output y_k that occurs at time k is a linear combination of current and previous inputs; that is, $(x_i, i \leq k)$. If the filter $B(Z)$ had included a term like b_{-1}/Z, then the output y_k at time k would be a linear combination of current and previous inputs and x_{k+1}, an input that really has not arrived at time k. Such a filter is called a "**nonrealizable**" filter, because it could not operate in the real world where nothing can respond now to an excitation that has not yet occurred. However, nonrealizable filters are occasionally useful in computer simulations where all the data is prerecorded.

1.2. FOURIER SUMS

The world is filled with sines and cosines. The coordinates of a point on a spinning wheel are $(x, y) = (\cos(\omega t + \phi), \sin(\omega t + \phi))$, where ω is the angular frequency of revolution and ϕ is the phase angle. The purest tones and the purest colors are sinusoidal. The movement of a pendulum is nearly sinusoidal, the approximation going to perfection in the limit of small amplitude motions. The sum of all the tones

in any signal is its "spectrum."

Small amplitude signals are widespread in nature, from the vibrations of atoms to the sound vibrations we create and observe in the earth. Sound typically compresses air by a volume fraction of 10^{-3} to 10^{-6}. In water or solid, the compression is typically 10^{-6} to 10^{-9}. A mathematical reason why sinusoids are so common in nature is that laws of nature are typically expressible as partial differential equations. Whenever the coefficients of the differentials (which are functions of material properties) are constant in time and space, the equations have exponential and sinusoidal solutions that correspond to waves propagating in all directions.

1.2.1. Superposition of sinusoids

Fourier analysis is built from the complex exponential

$$e^{-i\omega t} = \cos\omega t - i\,\sin\omega t \tag{1.13}$$

A Fourier component of a time signal is a complex number, a sum of real and imaginary parts, say

$$B = \Re B + i\Im B \tag{1.14}$$

which is attached to some frequency. Let j be an integer and ω_j be a set of frequencies. A signal $b(t)$ can be manufactured by adding a collection of complex exponential signals, each complex exponential being scaled by a complex coefficient B_j, namely,

$$b(t) \quad = \quad \sum_j B_j \; e^{-i\omega_j t} \tag{1.15}$$

This manufactures a **complex-valued signal**. How do we arrange for $b(t)$ to be real? We can throw away the imaginary part, which is like adding $b(t)$ to its complex conjugate $\overline{b}(t)$, and then dividing by two:

$$\Re\, b(t) \quad = \quad \frac{1}{2} \sum_j (B_j \; e^{-i\omega_j t} + \overline{B}_j \; e^{i\omega_j t}) \tag{1.16}$$

In other words, for each positive ω_j with amplitude B_j, we add a negative $-\omega_j$ with amplitude $\overline{B}_j$ (likewise, for every negative ω_j ...). The B_j are called the "frequency function," or the "Fourier transform." Loosely, the B_j are called the "**spectrum**," though technically, and in this book, the word "spectrum" should be reserved for the product $\overline{B}_j \, B_j$. The words "**amplitude spectrum**" universally mean $\sqrt{\overline{B}_j \, B_j}$.

In practice, the collection of frequencies is almost always evenly spaced. Let j be an integer $\omega = j\ \Delta\omega$ so that

$$b(t) \quad = \quad \sum_j B_j\, e^{-i(j\ \Delta\omega)t} \tag{1.17}$$

Representing a signal by a sum of sinusoids is technically known as "inverse Fourier transformation." An example of this is shown in Figure 1.6.

1.2.2. Sampled time and Nyquist frequency

In the world of computers, time is generally mapped into integers too, say $t = n\Delta t$. This is called "discretizing" or "sampling." The highest possible frequency expressible on a **mesh** is $(\cdots, 1, -1, +1, -1, +1, -1, \cdots)$, which is the same as $e^{i\pi n}$. Setting $e^{i\omega_{\max} t} = e^{i\pi n}$, we see that the maximum frequency is

$$\omega_{\max} \quad = \quad \frac{\pi}{\Delta t} \tag{1.18}$$

Time is commonly given in either seconds or sample units, which are the same when $\Delta t = 1$. In applications, frequency is usually expressed in cycles per second,

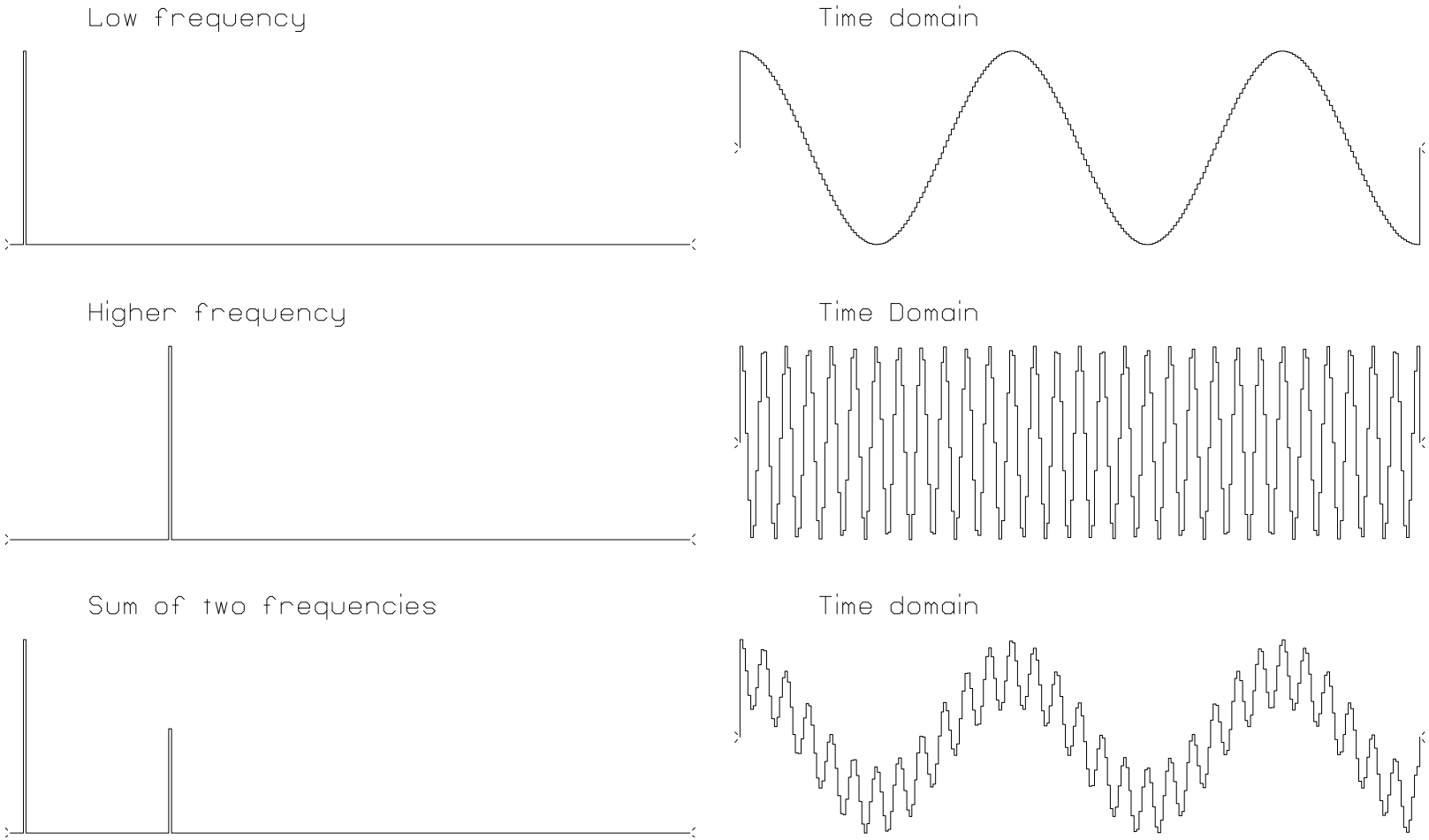


Figure 1.6: Superposition of two sinusoids. (Press button to activate program `ed1D`. See appendix for details.) cs-cosines [NR]

which is the same as **Hertz**, abbreviated **Hz**. In computer work, frequency is usually specified in cycles per sample. In theoretical work, frequency is usually expressed in **radian**s where the relation between radians and cycles is $\omega = 2\pi f$. We use radians because, otherwise, equations are filled with 2π's. When time is given in sample units, the maximum frequency has a name: it is the "**Nyquist frequency**," which is π radians or $1/2$ cycle per sample.

1.2.3. Fourier sum

In the previous section we superposed uniformly spaced frequencies. Now we will **superpose** delayed impulses. The frequency function of a delayed impulse at time delay t_0 is $e^{i\omega t_0}$. Adding some pulses yields the "**Fourier sum**":

$$B(\omega) \quad = \quad \sum_n b_n \, e^{i\omega t_n} \quad = \quad \sum_n b_n \, e^{i\omega n \Delta t} \tag{1.19}$$

The Fourier sum transforms the signal b_t to the frequency function $B(\omega)$. Time will often be denoted by t, even though its units are sample units instead of physical units. Thus we often see b_t in equations like (1.19) instead of b_n, resulting in an implied $\Delta t = 1$.

1.3. FOURIER AND Z-TRANSFORM

The frequency function of a pulse at time $t_n = n\Delta t$ is $e^{i\omega n\Delta t} = (e^{i\omega\Delta t})^n$. The factor $e^{i\omega\Delta t}$ occurs so often in applied work that it has a name:

$$Z \quad = \quad e^{i\omega\,\Delta t} \tag{1.20}$$

With this Z, the pulse at time t_n is compactly represented as Z^n. The variable Z makes **Fourier transform**s look like polynomials, the subject of a literature called "Z**-transform**s." The Z-transform is a variant form of the Fourier transform that is particularly useful for time-discretized (sampled) functions.

From the definition (1.20), we have $Z^2 = e^{i\omega 2\Delta t}$, $Z^3 = e^{i\omega 3\Delta t}$, etc. Using these equivalencies, equation (1.19) becomes

$$B(\omega) \quad = \quad B(\omega(Z)) \quad = \quad \sum_n b_n\, Z^n \tag{1.21}$$

1.3.1. Unit circle

In this chapter, ω is a real variable, so $Z = e^{i\omega\Delta t} = \cos\omega\Delta t + i\,\sin\omega\Delta t$ is a complex variable. It has unit magnitude because $\sin^2 + \cos^2 = 1$. As ω ranges on the real

axis, Z ranges on the unit circle $|Z| = 1$. In chapter 3 we will see how the definition (1.20) also applies for complex values of ω.

1.3.2. Differentiator

A particularly interesting factor is $(1 - Z)$, because the filter $(1, -1)$ is like a time derivative. The time-derivative filter destroys **zero frequency** in the input signal. The zero frequency is $(\cdots, 1, 1, 1, \cdots)$ with a Z-transform $(\cdots + Z^2 + Z^3 + Z^4 + \cdots)$. To see that the filter $(1 - Z)$ destroys zero frequency, notice that $(1 - Z)(\cdots + Z^2 + Z^3 + Z^4 + \cdots) = 0$. More formally, consider output $Y(Z) = (1 - Z)X(Z)$ made from the filter $(1 - Z)$ and any input $X(Z)$. Since $(1 - Z)$ vanishes at $Z = 1$, then likewise $Y(Z)$ must vanish at $Z = 1$. Vanishing at $Z = 1$ is vanishing at frequency $\omega = 0$ because $Z = \exp(i\omega\Delta t)$ from (1.20). Now we can recognize that multiplication of two functions of Z or of ω is the equivalent of convolving the associated time functions.

> Multiplication in the frequency domain is **convolution** in the time domain.

A popular mathematical abbreviation for the convolution operator is an asterisk:

equation (1.9), for example, could be denoted by $y_t = x_t * b_t$. I do not disagree with asterisk notation, but I prefer the equivalent expression $Y(Z) = X(Z)B(Z)$, which simultaneously exhibits the time domain and the frequency domain.

The filter $(1 - Z)$ is often called a "**differentiator**." It is displayed in Figure 1.7.

The letter "z" plotted at the origin in Figure 1.7 denotes the **root** of $1 - Z$ at $Z = 1$, where $\omega = 0$. Another interesting filter is $1 + Z$, which destroys the highest possible frequency $(1, -1, 1, -1, \cdots)$, where $\omega = \pi$.

A root is a numerical value for which a polynomial vanishes. For example, $2 - Z - Z^2 = (2 + Z)(1 - Z)$ vanishes whenever $Z = -2$ or $Z = 1$. Such a root is also called a "**zero**." The fundamental theorem of algebra says that if the highest power of Z in a polynomial is Z^N, then the polynomial has exactly N roots, not necessarily distinct. As N gets large, finding these roots requires a sophisticated computer program. Another complication is that complex numbers can arise. We will soon see that complex roots are exactly what we need to design filters that destroy any frequency.

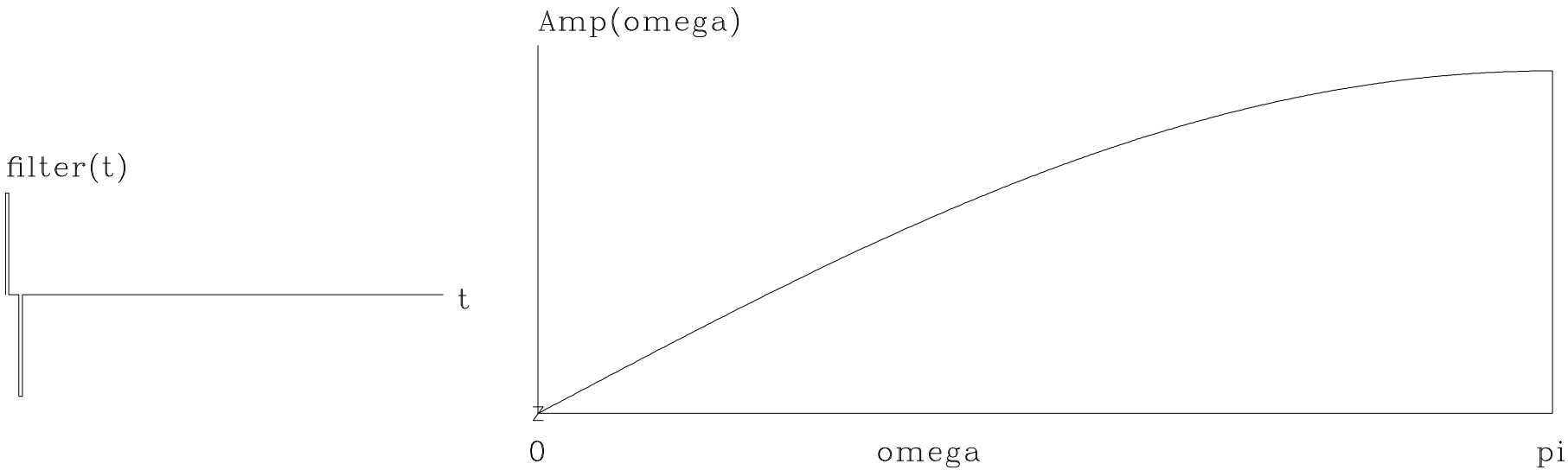


Figure 1.7: A discrete representation of the first-derivative operator. The filter $(1, -1)$ is plotted on the left, and on the right is an amplitude response, i.e., $|1 - Z|$ versus ω. (Press button to activate program `Zplane`. See appendix for details.) cs-ddt [NR]

1.3.3. Gaussian examples

The filter $(1+Z)/2$ is a running average of two adjacent time points. Applying this filter N times yields the filter $(1+Z)^N/2^N$. The coefficients of the filter $(1+Z)^N$ are generally known as **Pascal's triangle**. For large N the coefficients tend to a mathematical limit known as a **Gaussian** function, $\exp(-\alpha(t-t_0)^2)$, where α and t_0 are constants that we will determine in chapter 11. We will not prove it here, but this Gaussian-shaped signal has a Fourier transform that also has a Gaussian shape, $\exp(-\beta\omega^2)$. The Gaussian shape is often called a "bell shape." Figure 1.8 shows an example for $N \approx 15$. Note that, except for the rounded ends, the bell shape seems a good fit to a triangle function. Curiously, the filter $(.75+.25Z)^N$ also tends to the same Gaussian but with a different t_0. A mathematical theorem (discussed in chapter 11) says that almost any polynomial raised to the N-th power yields a Gaussian.

In seismology we generally fail to observe the **zero frequency**. Thus the idealized seismic pulse cannot be a Gaussian. An analytic waveform of longstanding popularity in seismology is the second derivative of a Gaussian, also known as a "**Ricker wavelet**." Starting from the Gaussian and putting two more zeros at the origin with $(1-Z)^2 = 1-2Z+Z^2$ produces this old, favorite wavelet, shown in

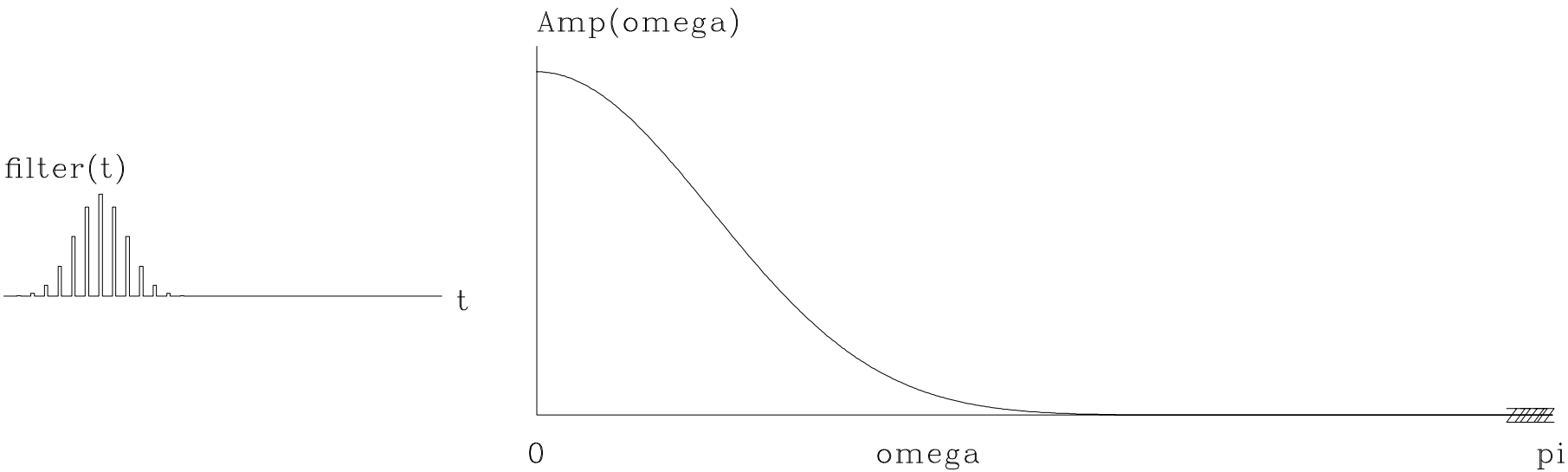


Figure 1.8: A Gaussian approximated by many powers of $(1+Z)$. cs-gauss [NR]

Figure 1.9.

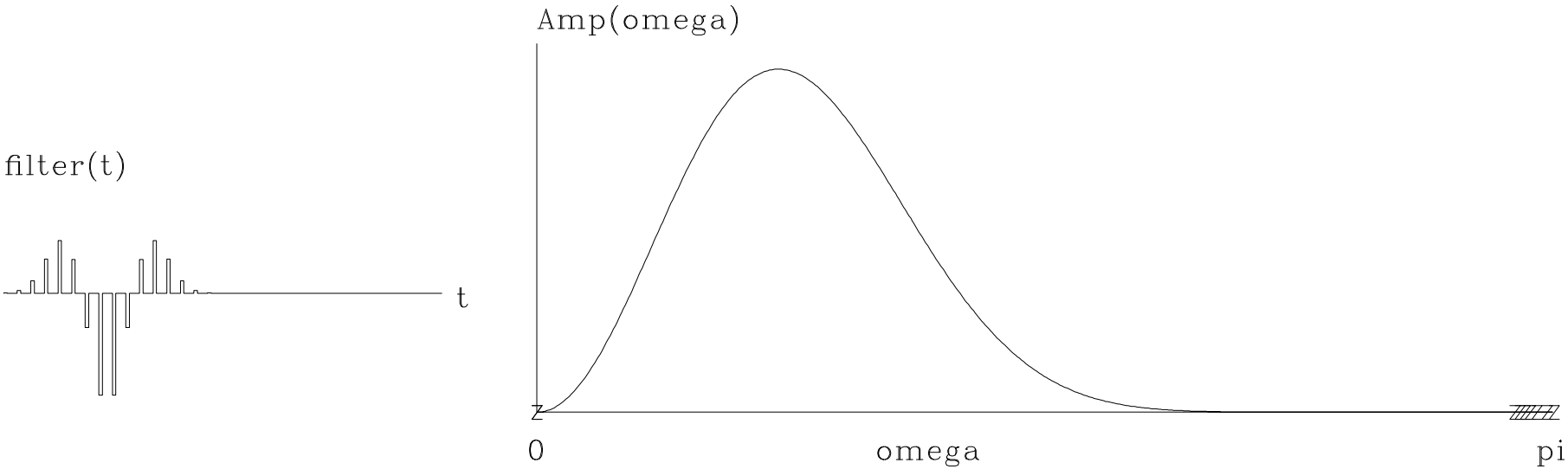


Figure 1.9: Ricker wavelet. cs-ricker [NR]

1.3.4. Complex roots

We have seen how a simple two-term filter can destroy the zero frequency or the Nyquist frequency. When we try to destroy any other frequency, we run into a new

difficulty—we will see **complex-valued signal**s. Let Z_0 take the complex value $Z_0 = e^{i\omega_0}$, where ω_0 is real. Further, choose $\omega_0 = \pi/2$ and as a result $Z_0 = i$. So the filter $(1 - Z/Z_0) = (1 + iZ)$ has the complex coefficients $(1, i)$, and its output is a complex-valued signal. Naturally this is annoying, because we usually prefer a real output signal.

The way to avoid complex-valued signals is to handle **negative frequency** $-\omega_0$ the same way we handle ω_0. To do this we use a filter with **two roots**, one at ω_0 and one at $-\omega_0$. The filter $(1 + iZ)(1 - iZ) = 1 + Z^2$ has real-valued time-domain coefficients, namely, $(1, 0, 1)$. The factor $(1 + iZ)$ vanishes when $Z = i$ or $\omega = \pi/2$, and $(1 - iZ)$ vanishes at $\omega = -\pi/2$. Notice what happens when the filter $(1, 0, 1)$ is convolved with the time series $b_t = (\cdots 1, 0, -1, 0, 1, 0, -1, \cdots)$: the output is zero at all times. This is because b_t is a sinusoid at the half-Nyquist frequency $\pi/2$, and the filter $(1, 0, 1)$ has zeros at plus and minus half-Nyquist.

Let us work out the general case for a root anywhere in the **complex plane**. Let the root Z_0 be decomposed into its real and imaginary parts:

$$Z_0 \quad = \quad x + iy \quad = \quad \Re Z_0 + i\Im Z_0 \tag{1.22}$$

and let the root be written in a polar form:

$$Z_0 \quad = \quad \frac{e^{i\omega_0}}{\rho} \tag{1.23}$$

where ω_0 and ρ are constants that can be derived from the constants $\Re Z_0$ and $\Im Z_0$ and vice versa. The conjugate root is $\overline{Z}_0 = e^{-i\omega_0}/\rho$. The combined filter is

$$\left(1-\frac{Z}{Z_0}\right)\left(1-\frac{Z}{\overline{Z}_0}\right) \quad = \quad 1-\left(\frac{1}{Z_0}+\frac{1}{\overline{Z}_0}\right)Z \;+\; \frac{Z^2}{Z_0\overline{Z}_0} \tag{1.24}$$

$$= \quad 1 \;-\; 2\rho\cos\omega_0\, Z \;+\; \rho^2 Z^2 \tag{1.25}$$

So the convolutional coefficients of this filter are the real values $(1, -2\rho\cos\omega_0, \rho^2)$. Taking $\rho = 1$, the filter completely destroys energy at frequency ω_0. Other values of ρ near unity suppress nearby frequencies without completely destroying them.

Recall that to keep the filter response real, any root on the positive ω-axis must have a twin on the negative ω-axis. In the figures I show here, the negative axis is not plotted, so we must remember the twin. Figure 1.10 shows a discrete approximation to the second derivative. It is like $(1-Z)^2$, but since both its roots are in

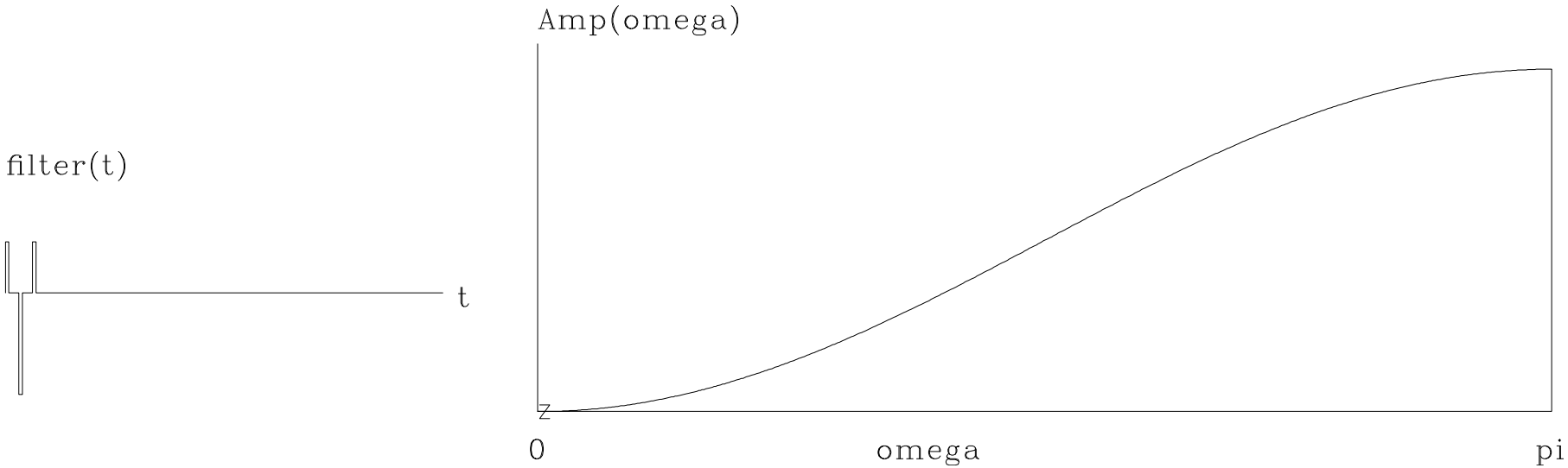


Figure 1.10: Approximation to the second difference operator $(1, -2, 1)$. cs-ddt2 [NR]

the same place at $Z = 1$, I pushed them a little distance apart, one going to positive frequencies and one to negative.

1.3.5. Inverse Z-transform

Fourier analysis is widely used in mathematics, physics, and engineering as a **Fourier integral** transformation pair:

$$B(\omega) = \int_{-\infty}^{+\infty} b(t) e^{i\omega t}\, dt \tag{1.26}$$

$$\bar{b}(t) = \int_{-\infty}^{+\infty} B(\omega) e^{-i\omega t}\, d\omega \tag{1.27}$$

These integrals correspond to the sums we are working with here except for some minor details. Books in electrical engineering redefine $e^{i\omega t}$ as $e^{-i\omega t}$. That is like switching ω to $-\omega$. Instead, we have chosen the **sign convention** of physics, which is better for wave-propagation studies (as explained in IEI). The infinite limits on the integrals result from expressing the **Nyquist frequency** in **radian**s/second as $\pi/\Delta t$. Thus, as Δt tends to zero, the **Fourier sum** tends to the integral. When we

reach equation (1.31) we will see that if a scaling divisor of 2π is introduced into either (1.26) or (1.27), then $b(t)$ will equal $b(t)$.

The Z-transform is always easy to make, but the Fourier integral could be difficult to perform, which is paradoxical, because the transforms are really the same. To make a Z-transform, we merely attach powers of Z to successive data points. When we have $B(Z)$, we can refer to it either as a time function or a frequency function. If we graph the polynomial coefficients, then it is a time function. It is a frequency function if we evaluate and graph the polynomial $B(Z = e^{i\omega})$ for various frequencies ω.

If the Z-transform amounts to attaching powers of Z to successive points of a time function, then the **inverse Z-transform** must be merely identifying coefficients of various powers of Z with different points in time. How can this mere "identification of coefficients" be the same as the apparently more complicated operation of inverse Fourier integration? Let us see. The **inverse Fourier integral** (1.27) for integer values of time is

$$b_t \quad = \quad \frac{1}{2\pi} \int_{-\pi}^{+\pi} B(\omega) e^{-i\omega t} \, d\omega \tag{1.28}$$

Substituting (1.21) into (1.28), we get

$$b_t \quad = \quad \frac{1}{2\pi}\int_{-\pi}^{\pi}(\cdots + b_{-1}e^{-i\omega} + b_0 + b_1 e^{+i\omega} + \cdots)e^{-i\omega t}\,d\omega \tag{1.29}$$

Since sinusoids have as much area above the axis as below, the integration of $e^{in\omega}$ over $-\pi \le \omega < +\pi$ gives zero unless $n = 0$, that is,

$$\begin{aligned}\frac{1}{2\pi}\int_{-\pi}^{\pi} e^{in\omega}\,d\omega &= \frac{1}{2\pi}\int_{-\pi}^{\pi}(\cos n\omega + i\sin n\omega)\,d\omega \\ &= \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{if } n = \text{non-zero integer} \end{cases}\end{aligned} \tag{1.30}$$

Of all the terms in the integrand (1.29), we see from (1.30) that only the term with b_t will contribute to the integral; all the rest oscillate and cancel. In other words, it is only the coefficient of Z to the zero power that contributes to the integral, so (1.29) reduces to

$$b_t \quad = \quad \frac{1}{2\pi}\int_{-\pi}^{+\pi} b_t\, e^{-i0}\,d\omega \tag{1.31}$$

This shows how inverse Fourier transformation is just like identifying coefficients of powers of Z. It also shows why the scale factor in equation (1.28) is 2π.

EXERCISES:

1 Let $B(Z) = 1 + Z + Z^2 + Z^3 + Z^4$. Graph the coefficients of $B(Z)$ as a function of the powers of Z. Graph the coefficients of $[B(Z)]^2$.

2 As ω moves from zero to positive frequencies, where is Z and which way does it rotate around the unit circle, clockwise or counterclockwise?

3 Identify locations on the unit circle of the following frequencies: (1) the zero frequency, (2) the Nyquist frequency, (3) negative frequencies, and (4) a frequency sampled at 10 points per wavelength.

4 Given numerical constants $\Re Z_0$ and $\Im Z_0$, derive ω_0 and ρ.

5 Sketch the amplitude spectrum of Figure 1.9 from 0 to 4π.

1.4. CORRELATION AND SPECTRA

The spectrum of a signal is a positive function of frequency that says how much of each tone is present. The Fourier transform of a spectrum yields an interesting function called an "**autocorrelation**," which measures the similarity of a signal to itself shifted.

1.4.1. Spectra in terms of Z-transforms

Let us look at spectra in terms of Z-transforms. Let a **spectrum** be denoted $S(\omega)$, where

$$S(\omega) \quad = \quad |B(\omega)|^2 \quad = \quad \overline{B(\omega)}B(\omega) \tag{1.32}$$

Expressing this in terms of a three-point Z-transform, we have

$$S(\omega) \quad = \quad (\bar{b}_0 + \bar{b}_1 e^{-i\omega} + \bar{b}_2 e^{-i2\omega})(b_0 + b_1 e^{i\omega} + b_2 e^{i2\omega}) \tag{1.33}$$

$$S(Z) \quad = \quad \left(\bar{b}_0 + \frac{\bar{b}_1}{Z} + \frac{\bar{b}_2}{Z^2}\right)(b_0 + b_1 Z + b_2 Z^2) \tag{1.34}$$

$$S(Z) \quad = \quad \overline{B}\left(\frac{1}{Z}\right) B(Z) \tag{1.35}$$

It is interesting to multiply out the polynomial $\overline{B}(1/Z)$ with $B(Z)$ in order to examine the coefficients of $S(Z)$:

$$\begin{aligned} S(Z) &= \frac{\overline{b}_2 b_0}{Z^2} + \frac{(\overline{b}_1 b_0 + \overline{b}_2 b_1)}{Z} + (\overline{b}_0 b_0 + \overline{b}_1 b_1 + \overline{b}_2 b_2) + (\overline{b}_0 b_1 + \overline{b}_1 b_2) Z + \overline{b}_0 b_2 Z^2 \\ S(Z) &= \frac{s_{-2}}{Z^2} + \frac{s_{-1}}{Z} + s_0 + s_1 Z + s_2 Z^2 \end{aligned} \tag{1.36}$$

The coefficient s_k of Z^k is given by

$$s_k = \sum_i \overline{b}_i b_{i+k} \tag{1.37}$$

Equation (1.37) is the **autocorrelation** formula. The autocorrelation value s_k at lag 10 is s_{10}. It is a measure of the similarity of b_i with itself shifted 10 units in time. In the most frequently occurring case, b_i is real; then, by inspection of (1.37), we see that the autocorrelation coefficients are real, and $s_k = s_{-k}$.

Specializing to a real time series gives

$$S(Z) = s_0 + s_1 \left(Z + \frac{1}{Z} \right) + s_2 \left(Z^2 + \frac{1}{Z^2} \right) \tag{1.38}$$

$$S(Z(\omega)) = s_0 + s_1(e^{i\omega} + e^{-i\omega}) + s_2(e^{i2\omega} + e^{-i2\omega}) \quad (1.39)$$

$$S(\omega) = s_0 + 2s_1 \cos\omega + 2s_2 \cos 2\omega \quad (1.40)$$

$$S(\omega) = \sum_k s_k \cos k\omega \quad (1.41)$$

$$S(\omega) = \text{cosine transform of } \; s_k \quad (1.42)$$

This proves a classic theorem that for real-valued signals can be simply stated as follows:

> For any real signal, the cosine transform of the **autocorrelation** equals the magnitude squared of the Fourier transform.

1.4.2. Two ways to compute a spectrum

There are two computationally distinct methods by which we can compute a spectrum: (1) compute all the s_k coefficients from (1.37) and then form the cosine sum (1.41) for each ω; and alternately, (2) evaluate $B(Z)$ for some value of Z on the unit circle, and multiply the resulting number by its complex conjugate. Repeat for

many values of Z on the unit circle. When there are more than about twenty lags, method (2) is cheaper, because the fast Fourier transform discussed in chapter 9 can be used.

1.4.3. Common signals

Figure 1.11 shows some common signals and their **autocorrelation**s. Figure 1.12 shows the cosine transforms of the autocorrelations. Cosine transform takes us from time to frequency and it also takes us from frequency to time. Thus, transform pairs in Figure 1.12 are sometimes more comprehensible if you interchange time and frequency. The various signals are given names in the figures, and a description of each follows:

cos The theoretical spectrum of a sinusoid is an impulse, but the sinusoid was truncated (multiplied by a rectangle function). The autocorrelation is a sinusoid under a triangle, and its spectrum is a broadened impulse (which can be shown to be a narrow sinc-squared function).

sinc The **sinc** function is $\sin(\omega_0 t)/(\omega_0 t)$. Its autocorrelation is another sinc function, and its spectrum is a rectangle function. Here the rectangle is corrupted

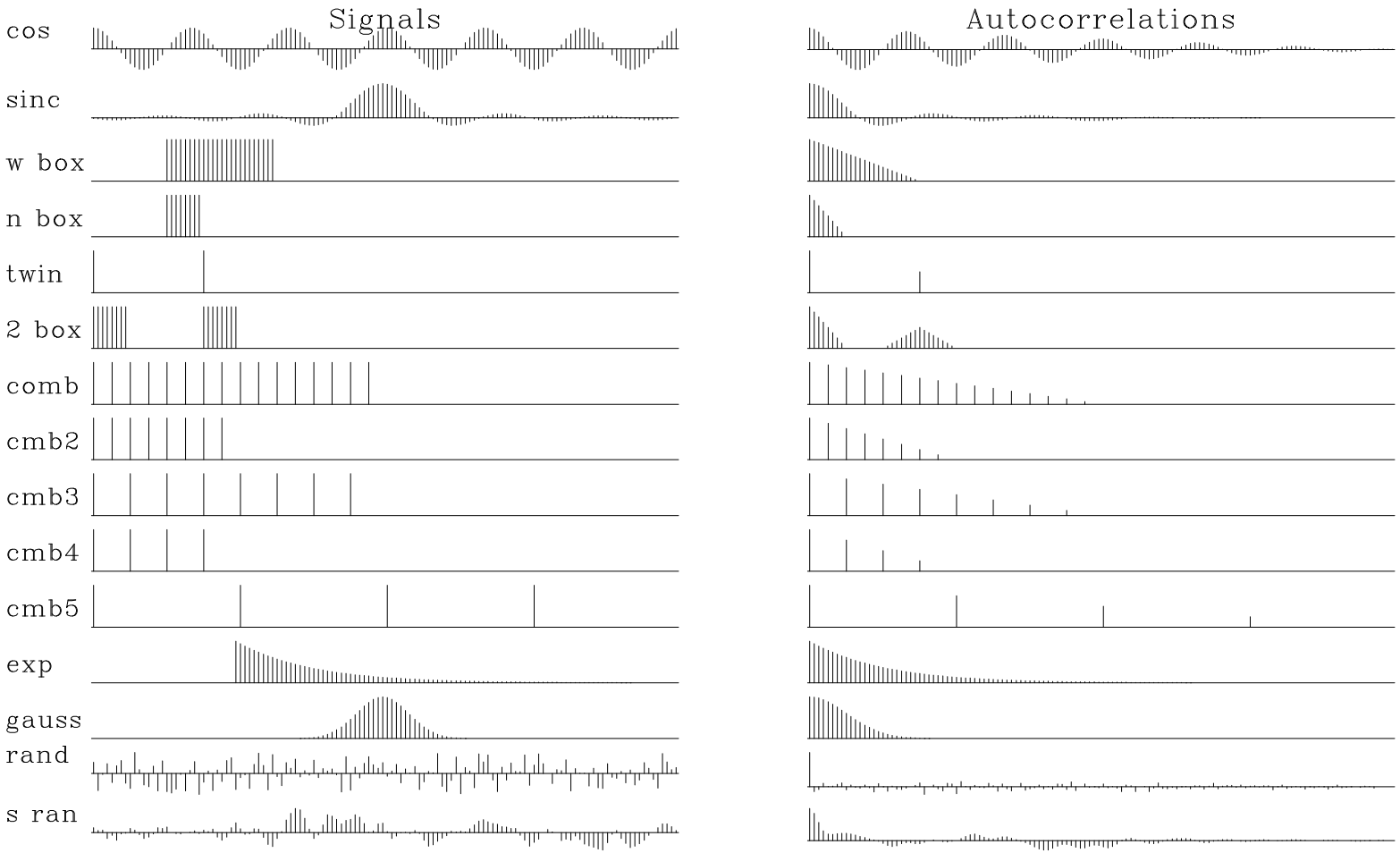


Figure 1.11: Common signals and one side of their autocorrelations. cs-autocor [ER]

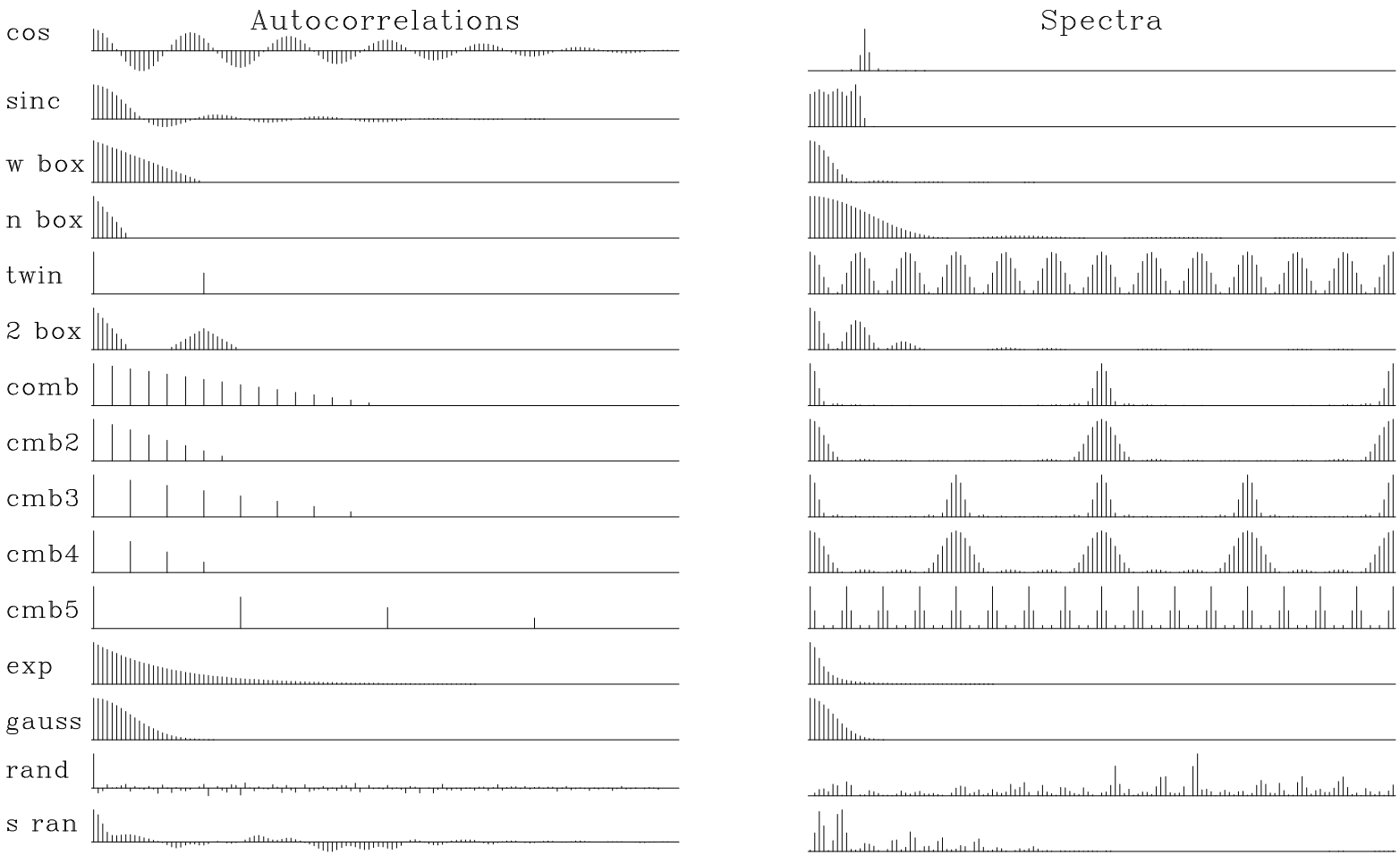


Figure 1.12: Autocorrelations and their cosine transforms, i.e., the (energy) spectra of the common signals. cs-spectra [ER]

slightly by "**Gibbs sidelobes**," which result from the time truncation of the original sinc.

wide box A wide **rectangle function** has a wide triangle function for an autocorrelation and a narrow sinc-squared spectrum.

narrow box A narrow rectangle has a wide sinc-squared spectrum.

twin Two pulses.

2 boxes Two separated narrow boxes have the spectrum of one of them, but this spectrum is modulated (multiplied) by a sinusoidal function of frequency, where the modulation frequency measures the time separation of the narrow boxes. (An oscillation seen in the frequency domain is sometimes called a "**quefrency**.")

comb Fine-toothed-**comb** functions are like rectangle functions with a lower Nyquist frequency. Coarse-toothed-comb functions have a spectrum which is a fine-toothed comb.

exponential The autocorrelation of a transient **exponential** function is a **double-sided exponential** function. The spectrum (energy) is a Cauchy function,

$1/(\omega^2+\omega_0^2)$. The curious thing about the **Cauchy function** is that the amplitude spectrum diminishes inversely with frequency to the *first* power; hence, over an infinite frequency axis, the function has infinite integral. The sharp edge at the onset of the transient exponential has much high-frequency energy.

Gauss The autocorrelation of a **Gaussian** function is another Gaussian, and the spectrum is also a Gaussian.

random **Random** numbers have an autocorrelation that is an impulse surrounded by some short grass. The spectrum is positive random numbers. For more about random signals, see chapter 11.

smoothed random Smoothed random numbers are much the same as random numbers, but their spectral bandwidth is limited.

1.4.4. Spectra of complex-valued signals

The **spectrum** of a signal is the magnitude squared of the Fourier transform of the function. Consider the real signal that is a delayed impulse. Its Z-transform is sim-

ply Z; so the real part is $\cos\omega$, and the imaginary part is $\sin\omega$. The real part is thus an **even function** of frequency and the imaginary part an **odd function** of frequency. This is also true of Z^2 and any sum of powers (weighted by real numbers), and thus it is true of any time function. For any real signal, therefore, the Fourier transform has an even real part RE and an imaginary odd part IO. Taking the squared magnitude gives $(\mathrm{RE}+i\,\mathrm{IO})(\mathrm{RE}-i\,\mathrm{IO}) = (\mathrm{RE})^2 + (\mathrm{IO})^2$. The square of an even function is obviously even, and the square of an odd function is also even. Thus, because the spectrum of a real-time function is even, its values at plus frequencies are the same as its values at minus frequencies. In other words, no special meaning should be attached to negative frequencies. This is not so of complex-valued signals.

Although most signals which arise in applications are real signals, a discussion of correlation and spectra is not mathematically complete without considering **complex-valued signal**s. Furthermore, complex-valued signals arise in many different contexts. In seismology, they arise in imaging studies when the space axis is Fourier transformed, i.e., when a two-dimensional function $p(t,x)$ is Fourier transformed over space to $P(t,k_x)$. More generally, complex-valued signals arise where rotation occurs. For example, consider two vector-component wind-speed indicators: one pointing north, recording n_t, and the other pointing west, recording w_t.

Now, if we make a complex-valued time series $v_t = n_t + i\,w_t$, the magnitude and phase angle of the complex numbers have an obvious physical interpretation: $+\omega$ corresponds to rotation in one direction (counterclockwise), and $(-\omega)$ to rotation in the other direction. To see why, suppose $n_t = \cos(\omega_0 t + \phi)$ and $w_t = -\sin(\omega_0 t + \phi)$. Then $v_t = e^{-i(\omega_0 t + \phi)}$. The Fourier transform is

$$V(\omega) \quad = \quad \int_{-\infty}^{+\infty} e^{-i(\omega_0 t + \phi)} e^{i\omega t}\, dt \tag{1.43}$$

The integrand oscillates and averages out to zero, except for the frequency $\omega = \omega_0$. So the frequency function is a pulse at $\omega = \omega_0$:

$$V(\omega) \quad = \quad \delta(\omega - \omega_0) e^{-i\phi} \tag{1.44}$$

Conversely, if w_t were $\sin(\omega_0 t + \phi)$, then the frequency function would be a pulse at $-\omega_0$, meaning that the wind velocity vector is rotating the other way.

1.4.5. Time-domain conjugate

A **complex-valued signal** such as $e^{i\omega_0 t}$ can be imagined as a **corkscrew**, where the real and imaginary parts are plotted on the x- and y-axes, and time t runs down

the axis of the screw. The complex conjugate of this signal reverses the y-axis and gives the screw an opposite handedness. In Z-transform notation, the **time-domain conjugate** is written

$$\overline{B}(Z) \quad = \quad \overline{b_0} + \overline{b_1} e^{i\omega} + \overline{b_2} e^{i2\omega} + \cdots \tag{1.45}$$

Now consider the complex conjugate of a frequency function. In Z-transform notation this is written

$$\overline{B(\omega)} \quad = \quad \overline{B}\left(\frac{1}{Z}\right) \quad = \quad \overline{b_0} + \overline{b_1} e^{-i\omega} + \overline{b_2} e^{-i2\omega} + \cdots \tag{1.46}$$

To see that it makes a difference in which domain we take a conjugate, contrast the two equations (1.45) and (1.46). The function $\overline{B}(\frac{1}{Z})B(Z)$ is a spectrum, whereas the function $\overline{b_t}\, b_t$ is called an "**envelope** function."

For example, given complex-valued b_t vanishing for $t < 0$, the composite filter $B(Z)\overline{B}(Z)$ is a causal filter with a real time function, whereas the filter $B(Z)\overline{B}(1/Z)$ is noncausal and also a real-valued function of time. (The latter filter would turn out to be symmetric in time only if all b_t were real.)

You might be tempted to think that $\overline{Z} = 1/Z$, but that is true only if ω is real, and often it is not. Chapter 3 is largely devoted to exploring the meaning of complex

frequency.

1.4.6. Spectral transfer function

Filters are often used to change the spectra of given data. With input $X(Z)$, filters $B(Z)$, and output $Y(Z)$, we have $Y(Z) = B(Z)X(Z)$ and the Fourier conjugate $\overline{Y}(1/Z) = \overline{B}(1/Z)\overline{X}(1/Z)$. Multiplying these two relations together, we get

$$\overline{Y}Y \quad = \quad (\overline{B}B)(\overline{X}X) \tag{1.47}$$

which says that the spectrum of the input times the spectrum of the filter equals the spectrum of the output. Filters are often characterized by the shape of their spectra; this shape is the same as the **spectral ratio** of the output over the input:

$$\overline{B}B \quad = \quad \frac{\overline{Y}Y}{\overline{X}X} \tag{1.48}$$

1.4.7. Crosscorrelation

The concept of **autocorrelation** and spectra is easily generalized to **crosscorrelation** and **cross-spectra**. Consider two Z-transforms $X(Z)$ and $Y(Z)$. The cross-

spectrum $C(Z)$ is defined by

$$C(Z) \quad = \quad \overline{X}\left(\frac{1}{Z}\right) Y(Z) \tag{1.49}$$

The crosscorrelation function is the coefficients c_k. If some particular coefficient c_k in $C(Z)$ is greater than any of the others, then it is said that the waveform x_t most resembles the waveform y_t if either x_t or y_t is delayed k time units with respect to the other.

1.4.8. Matched filtering

Figure 1.13 shows a **deep-water seismogram** where the bottom is unusually hard. The second signal is the wavelet that results from windowing about the first water-bottom reflection. Notice that the wavelet has a comparatively simple spectrum, its principal feature being that it vanishes at low frequencies and high frequencies. The input has a spectrum that is like that of the wavelet, but multiplied by a fine-toothed **comb** reminiscent of "cmb5" in Figure 1.12.

"**Matched filtering**" is crosscorrelating with a wavelet. Equivalently, it is convolving with the time-reversed wavelet. Matched filtering uses $Y(Z) = F(1/Z)X(Z)$

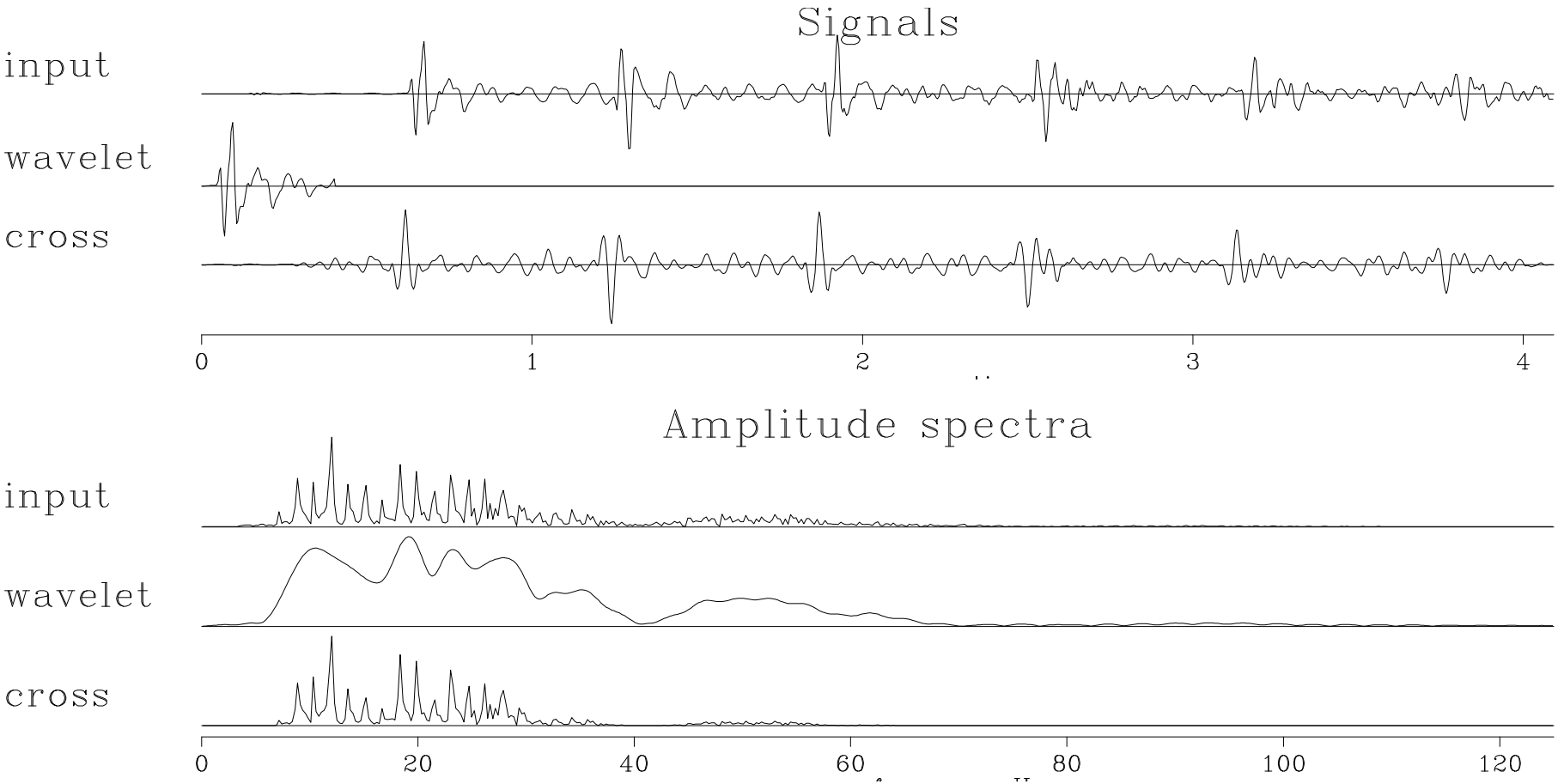


Figure 1.13: Example of matched filtering with water-bottom reflection. Top shows signals and bottom shows corresponding spectra. The result was time shifted to best align with the input. cs-match [ER]

instead of $Y(Z) = F(Z)X(Z)$. The third signal in Figure 1.13 shows the data cross-correlated with the sea-floor reflection. Notice that the output sea-floor reflection is symmetric like an **autocorrelation** function. Later bounces are **crosscorrelations**, but they resemble the autocorrelation. Ideally, alternate water-bottom reflections have alternating **polarities**. From the figure you can see that matched filtering makes this idealization more apparent. An annoying feature of the matched filter is that it is noncausal, i.e., there is an output before there is an input. You can see this in Figure 1.13 just before the water-bottom reflection.

EXERCISES:

1 Suppose a wavelet is made up of complex numbers. Is the autocorrelation relation $s_k = s_{-k}$ true? Is s_k real or complex? Is $S(\omega)$ real or complex?

2 If concepts of time and frequency are interchanged, what does the meaning of spectrum become?

3 Suggest a reason why the spectrum of the wavelet in Figure 1.13 contains more low-frequency energy than the whole seismogram.

4 Suggest a reason why the spectrum of the wavelet in Figure 1.13 contains more high-frequency energy than the whole seismogram.

Chapter 2

Discrete Fourier transform

Happily, Fourier sums are exactly invertible: given the output, the input can be quickly found. Because signals can be transformed to the frequency domain, manipulated there, and then returned to the time domain, convolution and correlation

can be done faster. Time derivatives can also be computed with more accuracy in the frequency domain than in the time domain. Signals can be shifted a fraction of the time sample, and they can be shifted back again exactly. In this chapter we will see how many operations we associate with the time domain can often be done better in the frequency domain. We will also examine some two-dimensional Fourier transforms.

2.1. FT AS AN INVERTIBLE MATRIX

A **Fourier sum** may be written

$$B(\omega) \quad = \quad \sum_t b_t \; e^{i\omega t} \quad = \quad \sum_t b_t \; Z^t \tag{2.1}$$

where the complex value Z is related to the real frequency ω by $Z = e^{i\omega}$. This Fourier sum is a way of building a continuous function of ω from discrete signal values b_t in the time domain. In this chapter we will study the computational tricks associated with specifying both time and frequency domains by a set of points. Begin with an example of a signal that is nonzero at four successive instants, (b_0, b_1, b_2, b_3).

The transform is

$$B(\omega) \quad = \quad b_0 + b_1 Z + b_2 Z^2 + b_3 Z^3 \tag{2.2}$$

The evaluation of this polynomial can be organized as a matrix times a vector, such as

$$\begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & W & W^2 & W^3 \\ 1 & W^2 & W^4 & W^6 \\ 1 & W^3 & W^6 & W^9 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{2.3}$$

Observe that the top row of the matrix evaluates the polynomial at $Z = 1$, a point where also $\omega = 0$. The second row evaluates $B_1 = B(Z = W = e^{i\omega_0})$, where ω_0 is some base frequency. The third row evaluates the Fourier transform for $2\omega_0$, and the bottom row for $3\omega_0$. The matrix could have more than four rows for more frequencies and more columns for more time points. I have made the matrix square in order to show you next how we can find the inverse matrix. The size of the matrix in (2.3) is $N = 4$. If we choose the base frequency ω_0 and hence W correctly, the

inverse matrix will be

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = 1/N \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1/W & 1/W^2 & 1/W^3 \\ 1 & 1/W^2 & 1/W^4 & 1/W^6 \\ 1 & 1/W^3 & 1/W^6 & 1/W^9 \end{bmatrix} \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} \tag{2.4}$$

Multiplying the matrix of (2.4) with that of (2.3), we first see that the diagonals are +1 as desired. To have the off diagonals vanish, we need various sums, such as $1 + W + W^2 + W^3$ and $1 + W^2 + W^4 + W^6$, to vanish. Every element (W^6, for example, or $1/W^9$) is a unit vector in the complex plane. In order for the sums of the unit vectors to vanish, we must ensure that the vectors pull symmetrically away from the origin. A uniform distribution of directions meets this requirement. In other words, W should be the N-th root of unity, i.e.,

$$W = \sqrt[N]{1} = e^{2\pi i/N} \tag{2.5}$$

The lowest frequency is zero, corresponding to the top row of (2.3). The next-to-the-lowest frequency we find by setting W in (2.5) to $Z = e^{i\omega_0}$. So $\omega_0 = 2\pi/N$;

and for (2.4) to be inverse to (2.3), the frequencies required are

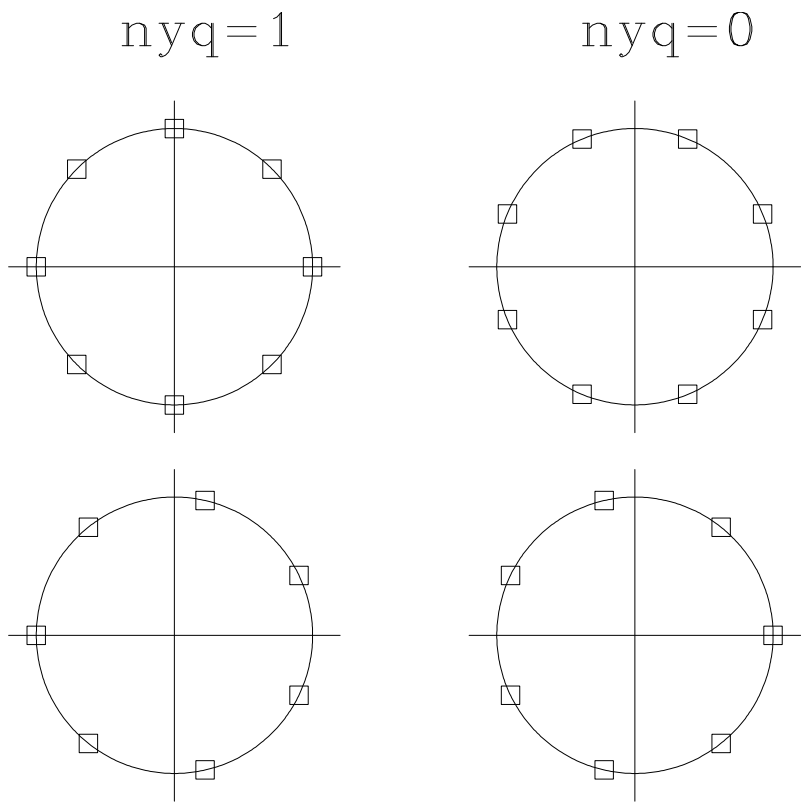
$$\omega_k \quad = \quad \frac{(0,1,2,\ldots,N-1)\,2\pi}{N} \tag{2.6}$$

2.1.1. The Nyquist frequency

The highest frequency in equation (2.6), $\omega = 2\pi(N-1)/N$, is almost 2π. This frequency is twice as high as the Nyquist frequency $\omega = \pi$. The **Nyquist frequency** is normally thought of as the "highest possible" frequency, because $e^{i\pi t}$, for integer t, plots as $(\cdots,1,-1,1,-1,1,-1,\cdots)$. The double Nyquist frequency function, $e^{i2\pi t}$, for integer t, plots as $(\cdots,1,1,1,1,1,\cdots)$. So this frequency above the highest frequency is really zero frequency! We need to recall that $B(\omega) = B(\omega - 2\pi)$. Thus, all the frequencies near the upper end of the range (2.6) are really small negative frequencies. Negative frequencies on the interval $(-\pi,0)$ were moved to interval $(\pi,2\pi)$ by the matrix form of Fourier summation.

Figure 2.1 shows possible arrangements for distributing points uniformly around the **unit circle**. Those circles labeled "even" and "odd" have even and odd numbers of points on their perimeters. Zero frequency is the right edge of the circles, and

Figure 2.1: Possible arrangements of uniformly spaced frequencies. Nyquist frequency is at the left edge of the circles and zero frequency at the right edge. dft-circles [ER]



Nyquist frequency is the left edge. Those circles labeled “nyq=1” have a point at the Nyquist frequency, and those labeled “nyq=0” do not.

Rewriting equations (2.3) and (2.4) with different even values of N leads to arrangements like the upper left circle in Figure 2.1. Rewriting with odd values of N leads to arrangements like the lower right circle. Although the “industry standard” is the upper-left arrangement, the two right-side arrangements are appealing for two reasons: the Nyquist frequency is absent, and its time-domain equivalent, the jump from large positive time to large negative time (a philosophical absurdity), is also absent. We will be testing and evaluating all four arrangements in Figure 2.5.

2.1.2. Laying out a mesh

In theoretical work and in programs, the definition $Z = e^{i\omega\Delta t}$ is often simplified to $\Delta t = 1$, leaving us with $Z = e^{i\omega}$. How do we know whether ω is given in radians per second or radians per sample? We may not invoke a cosine or an exponential unless the argument has no physical dimensions. So where we see ω without Δt, we know it is in units of radians per sample.

In practical work, frequency is typically given in cycles or **Hertz**, f, rather

than radians, ω (where $\omega = 2\pi f$). Here we will now switch to f. We will design a computer **mesh** on a physical object (such as a waveform or a function of space). We often take the mesh to begin at $t = 0$, and continue till the end t_{max} of the object, so the time range $t_{\text{range}} = t_{\text{max}}$. Then we decide how many points we want to use. This will be the N used in the discrete Fourier-transform program. Dividing the range by the number gives a mesh interval Δt.

Now let us see what this choice implies in the frequency domain. We customarily take the maximum frequency to be the Nyquist, either $f_{\text{max}} = .5/\Delta t$ Hz or $\omega_{\text{max}} = \pi/\Delta t$ radians/sec. The frequency range f_{range} goes from $-.5/\Delta t$ to $.5/\Delta t$. In summary:

- $\Delta t = t_{\text{range}}/N$ is time **resolution**.
- $f_{\text{range}} = 1/\Delta t = N/t_{\text{range}}$ is frequency range.
- $\Delta f = f_{\text{range}}/N = 1/t_{\text{range}}$ is frequency **resolution**.

In principle, we can always increase N to refine the calculation. Notice that increasing N sharpens the time resolution (makes Δt smaller) but does not sharpen the frequency resolution Δf, which remains fixed. Increasing N increases the frequency *range*, but not the frequency *resolution.*

What if we want to increase the frequency resolution? Then we need to choose t_{range} larger than required to cover our object of interest. Thus we either record data over a larger range, or we assert that such measurements would be zero. Three equations summarize the facts:

$$\Delta t \; f_{\text{range}} = 1 \tag{2.7}$$

$$\Delta f \; t_{\text{range}} = 1 \tag{2.8}$$

$$\Delta f \; \Delta t = \frac{1}{N} \tag{2.9}$$

Increasing *range* in the time domain increases *resolution* in the frequency domain and vice versa. Increasing **resolution** in one domain does not increase **resolution** in the other.

2.1.3. The comb function

Consider a constant function of time. In the frequency domain, it is an impulse at zero frequency. The **comb** function is defined to be zero at alternate time points.

Multiply this constant function by the comb function. The resulting signal contains equal amounts of two frequencies; half is zero frequency, and half is Nyquist frequency. We see this in the second row in Figure 2.2, where the Nyquist energy is in the middle of the frequency axis. In the third row, 3 out of 4 points are zeroed by another comb. We now see something like a new Nyquist frequency at half the Nyquist frequency visible on the second row.

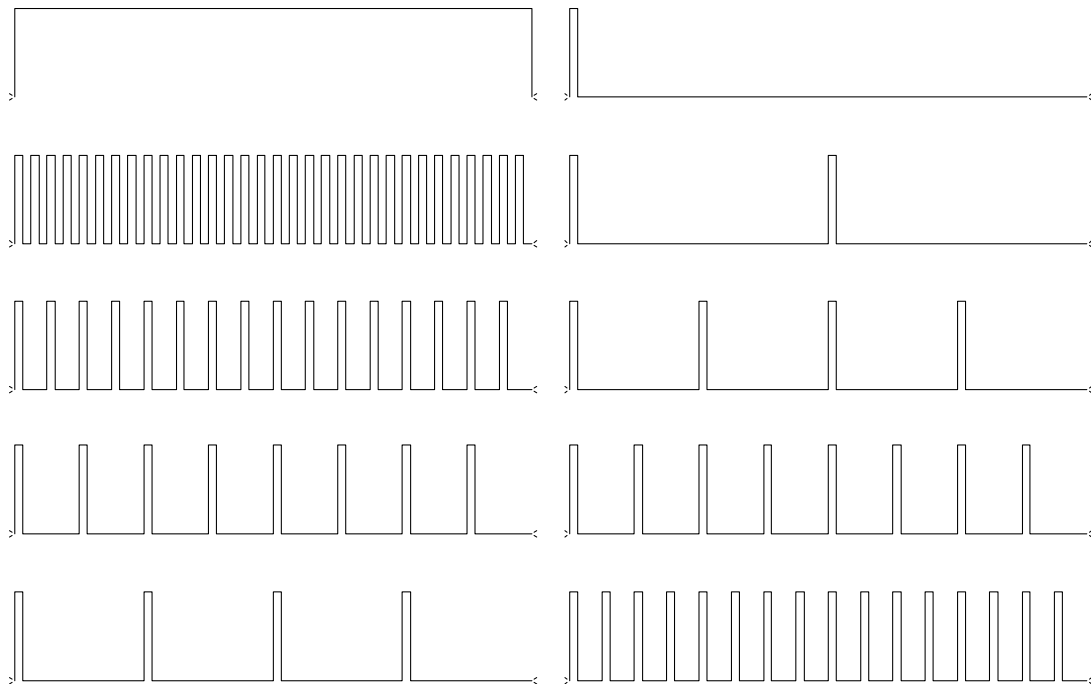


Figure 2.2: A zero-frequency function and its cosine transform. Successive rows show increasingly sparse sampling of the zero-frequency function. dft-comb [NR]

2.1.4. Undersampled field data

Figure 2.3 shows a recording of an **airgun** along with its spectrum. The original data is sampled at an interval of 4 milliseconds, which is 250 times per second. Thus, the **Nyquist frequency** $1/(2\Delta t)$ is 125 Hz. Negative frequencies are not shown, since the amplitude spectrum at negative frequency is identical with that at positive frequency. Think of extending the top row of spectra in Figure 2.3 to range from minus 125 Hz to plus 125 Hz. Imagine the even function of frequency centered at zero frequency—we will soon see it. In the second row of the plot, I decimated the data to 8 ms. This drops the Nyquist frequency to 62.5 Hz. Energy that was at -10 Hz appears at $125 - 10$ Hz in the *second* row spectrum. The appearance of what were formerly small negative frequencies near the Nyquist frequency is called "**fold**ing" of the spectrum. In the next row the data is sampled at 16 ms intervals, and in the last row at 32 ms intervals. The 8 ms sampling seems OK, whereas the 32 ms sampling looks poor. Study how the spectrum changes from one row to the next.

The spectrum suffers no visible harm in the drop from 4 ms to 8 ms. The 8 ms data could be used to construct the original 4 ms data by transforming the 8 ms data to the frequency domain, replacing values at frequencies above 125/2 Hz by zero,

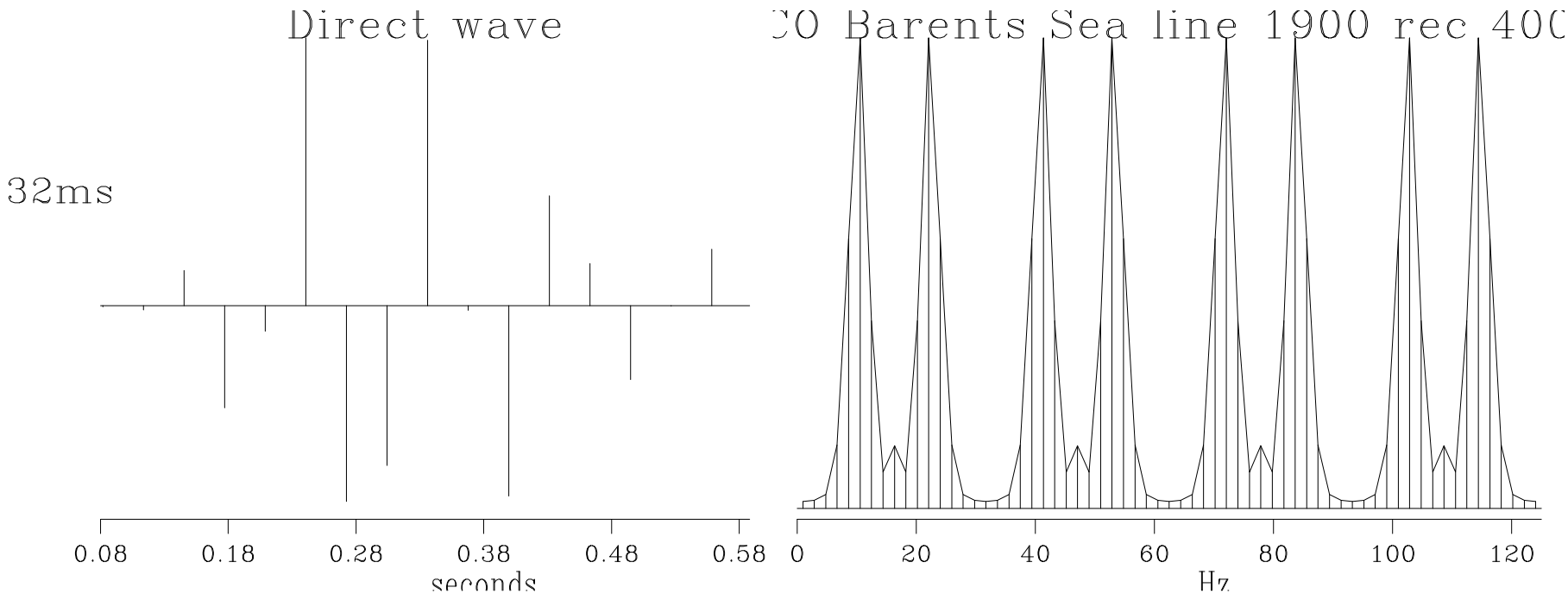


Figure 2.3: Raw data is shown on the top left, of about a half-second duration. Right shows amplitude spectra (magnitude of FT). In successive rows the data is sampled less densely. dft-undersample [ER]

and then inverse transforming to the time domain.

(Airguns usually have a higher frequency content than we see here. Some high-frequency energy was removed by the recording geometry, and I also removed some when preparing the data.)

2.2. INVERTIBLE SLOW FT PROGRAM

Because Fourier sums are exactly invertible, some other things we often require can be done exactly by doing them in the frequency domain.

Typically, signals are real valued. But the programs in this chapter are for complex-valued signals. In order to use these programs, copy the real-valued signal into a complex array, where the signal goes into the real part of the complex numbers; the imaginary parts are then automatically set to zero.

There is no universally correct choice of **scale factor** in Fourier transform: choice of scale is a matter of convenience. Equations (2.3) and (2.4) mimic the Z-transform, so their scaling factors are convenient for the convolution theorem—that a product in the frequency domain is a convolution in the time domain. Obviously, the scaling factors of equations (2.3) and (2.4) will need to be interchanged for the

complementary theorem that a convolution in the frequency domain is a product in the time domain. I like to use a scale factor that keeps the sums of squares the same in the time domain as in the frequency domain. Since I almost never need the scale factor, it simplifies life to omit it from the subroutine argument list. When a scaling program is desired, we can use a simple one like `scale()` /prog:scale. Complex-valued data can be scaled with `scale()` merely by doubling the value of `n`.

Fourier transform is just one of many transforms discussed in this book. In the case of most other transforms, the number of output values is different than the number of inputs. In addition, inverse transforms (and conjugate transforms), which will also be represented in code included in this book, transform in reverse, outputs to inputs. Finally, we will eventually combine transformations by addition or concatenation (one occurring after the other). All these considerations are expressed in the simple program `adjnull()`, which erases output before we begin. `adjnull()` may seem like too trivial a function to put in a library routine, but at last count, 15 other routines in this book use it. adjnull

```
subroutine adjnull( adj, add, x, nx,  y, ny )
integer ix, iy,     adj, add,    nx,     ny
real                             x( nx), y( ny )
if( add == 0 )
        if( adj == 0 )
                do iy= 1, ny
                        y(iy) = 0.
        else
                do ix= 1, nx
                        x(ix) = 0.
return; end
```

Back

```
subroutine slowft( adj, add, nyq, t0,dt,nt,tt,  f0,df,  nf,ff)
integer it,ie,     adj, add, nyq,       nt,              nf
complex cexp, cmplx,                    tt(nt),          ff(nf)
real pi2, freq, time, scale,       t0,dt,        f0,df
call adjnull(      adj, add,            tt,2*nt,         ff,2*nf)
                        pi2= 2. * 3.14159265;           scale = 1./sqrt( 1.*nt)
                        df = (1./dt) / nf
                 if( nyq>0)
                        f0 =   - .5/dt
                 else
                        f0 =   - .5/dt + df/2.
do ie = 1, nf {   freq= f0 + df*(ie-1)
do it = 1, nt {   time= t0 + dt*(it-1)
         if( adj == 0 )
                 ff(ie)= ff(ie) + tt(it) * cexp(cmplx(0., pi2*freq*time)) * scale
         else
                 tt(it)= tt(it) + ff(ie) * cexp(cmplx(0.,-pi2*freq*time)) * scale
         }}
return; end
```

Back

2.2.1. The slow FT code

The `slowft()` routine exhibits features found in many physics and engineering programs. For example, the time-domain signal (which I call "`tt()`"), has `nt` values subscripted, from `tt(1)` to `tt(nt)`. The first value of this signal `tt(1)` is located in real physical time at `t0`. The time interval between values is `dt`. The value of `tt(it)` is at time `t0+(it-1)*dt`. I do not use "`if`" as a pointer on the frequency axis because `if` is a keyword in most programming languages. Instead, I count along the frequency axis with a variable named `ie`. slowft The total frequency band is 2π radians per sample unit or $1/\Delta t$ Hz. Dividing the total interval by the number of points `nf` gives Δf. We could choose the frequencies to run from 0 to 2π radians/sample. That would work well for many applications, but it would be a nuisance for applications such as differentiation in the frequency domain, which require multiplication by $-i\omega$ including the **negative frequencies** as well as the positive. So it seems more natural to begin at the most negative frequency and step forward to the most positive frequency. Next, we must make a confusing choice.

Refer to Figure 2.1. We could begin the frequency axis at the negative Nyquist, $-.5/\Delta t$ Hz; then we would finish one point short of the positive Nyquist. This is shown on the left two circles in Figure 2.1. Alternately, for the right two circles

we could shift by half a mesh interval, so the points would **straddle** the **Nyquist frequency**. To do this, the most negative frequency would have to be $-.5/\Delta t + \Delta f/2$ Hz. In routine `slowft()` and in the test results, "`nyq=1`" is a logical statement that the Nyquist frequency is in the dataset. Oppositely, if the Nyquist frequency is interlaced by the given frequencies, then `nyq=0`. Finally, the heart of the program is to compute either a Fourier sum, or its inverse, which uses the complex conjugate.

The routine `ftlagslow()` below simply transforms a signal to the Fourier domain, multiplies by $\exp(i\omega t_0)$, where t_0 is some desired time `lag`, and then inverse transforms to the time domain. Notice that if the negative Nyquist frequency is present, it is treated as the average of the negative and positive Nyquist frequencies. If we do not take special care to do this, we will be disappointed to find that the time derivative of a real-time function develops an imaginary part. ftlagslow

Figure 2.4 shows what happens when an impulse is shifted by various fractions of a sample unit with subroutine `ftlagslow()`. Notice that during the delay, the edges of the signals **ripple**—this is sometimes called the "**Gibbs ripple**." You might find these ripples annoying, but it is not easy to try to represent an impulse halfway between two mesh points. You might think of doing so with $(.5, .5)$, but that lacks the high frequencies of an ideal impulse.

```
subroutine ftlagslow( nyq, lag, t0,dt, n1, ctt)
integer nyq, n1, ie
real lag, t0, dt, f0, df, freq
complex ctt(n1), cexp, cmplx
temporary complex cff(n1)

call slowft( 0, 0, nyq, t0, dt, n1, ctt, f0, df, n1, cff)

do ie= 1, n1 {  freq= f0 + (ie-1)*df
        if( ie==1  &&  nyq > 0)
                cff(1)  = cff(1)  * cos(            2.*3.14159265 * freq * lag )
        else
                cff(ie) = cff(ie) * cexp( cmplx(0., 2.*3.14159265 * freq * lag))
        }
call slowft( 1, 0, nyq, t0, dt, n1, ctt, f0, df, n1, cff)

return; end
```

Back

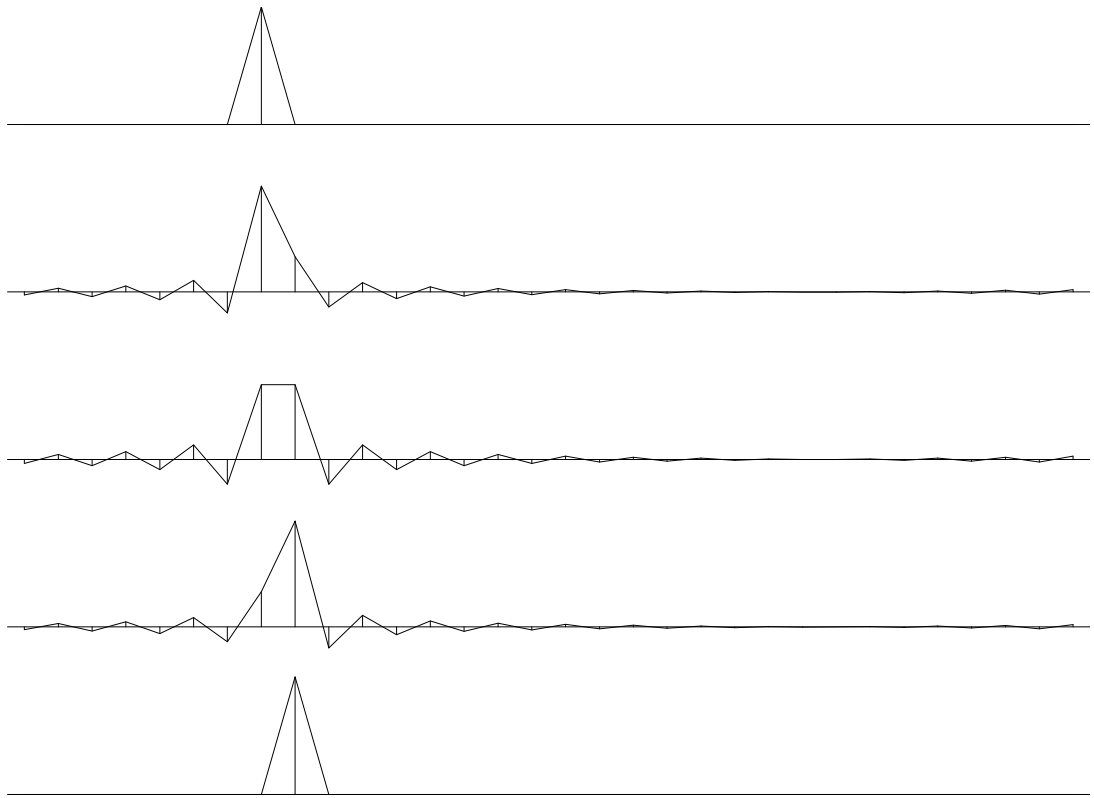


Figure 2.4: An impulse function delayed various fractions of a mesh point. Pushbutton for interaction (experimental). dft-delay [ER]

```
subroutine ftderivslow( nyq, t0,dt, ntf, ctt, cdd)
integer nyq, ntf, ie
real t0,dt,f0,df, freq
complex ctt(ntf), cdd(ntf), cmplx
temporary complex cff(ntf)
call slowft( 0, 0, nyq, t0, dt, ntf, ctt, f0, df, ntf, cff)

do ie= 1, ntf {  freq= f0+(ie-1)*df
        cff(ie) = cff(ie) *  cmplx( 0., - 2. * 3.141549265 * freq )
        }
if( nyq > 0 )                                           #  if( omega0 == -pi/dt)
        cff(1) = 0.

call slowft( 1, 0, nyq, t0, dt, ntf, cdd, f0, df, ntf, cff)
return; end
```

Back

The routine `ftderivslow()` below is the Fourier-domain routine for computing a time derivative by multiplying in the frequency domain by $-i\omega$. ftderivslow

2.2.2. Truncation problems

When real signals are transformed to the frequency domain, manipulated there, and then transformed back to the time domain, they will no longer be completely real. There will be a tiny noise in the imaginary part due to numerical roundoff. The size of the imaginary part, theoretically zero, is typically about 10^{-6} of the real part. This is also about the size of the error on the real part of a signal after inverse transform. It is almost always much smaller than experimental errors and is of little consequence. As a check, I viewed these near-zero imaginary parts, but I do not show them here.

A more serious error is a relative one of about $1/N$ on an N-point signal. This arises from insufficient care in numerical analysis, especially associated with the ends of the time or frequency axis. To show **end effect**s, I will print some numbers resulting from processing very short signals with `slowft()` /prog:slowft. Below I show first the result that a transform followed by an inverse transform gives the

original signal. I display this for both even and odd lengths of data, and for the two Nyquist arrangements as well.

```
Inversion: You should see (2,1,0,0)
nyq=0   2.00   1.00   0.00   0.00
nyq=1   2.00   1.00   0.00   0.00
nyq=0   2.00   1.00   0.00   0.00   0.00
nyq=1   2.00   1.00   0.00   0.00   0.00
```

Second, I display the result of a test of the convolution theorem by convolving $(2,1)$ with $(1,-1)$. We see that the scale factor varies with the data size because we are using the energy-conserving FT, instead of equations (2.3) and (2.4). No problems yet.

```
Convolution theorem: Proportional to (0,2,-1,-1,0,0,0,0)
nyq=0   0.00   0.89  -0.45  -0.45   0.00
nyq=1   0.00   0.89  -0.45  -0.45   0.00
nyq=0   0.00   0.82  -0.41  -0.41   0.00   0.00
nyq=1   0.00   0.82  -0.41  -0.41   0.00   0.00
```

The third test is **delay**ing a signal by two samples using `ftlagslow()` /prog:ftlagslov

Here the interesting question is what will happen at the ends of the data sample. Sometimes what shifts off one end shifts back in the other end: then the signal space is like the perimeter of a circle. Surprisingly, another aggravating possibility exists. What shifts off one end can return in the other end *with opposite* ***polarity***. When this happens, a figure like 2.4 looks much rougher because of the discontinuity at the ends. Even if there is no physical signal at the ends, the ripple we see in Figure 2.4 reaches the ends and worsens. (Recall that `nyq=1` means the Nyquist frequency is included in the spectrum, and that `nyq=0` means it is interlaced.)

```
Delay tests:
In                  11.0   12.0   13.0   14.0   15.0   16.0   17.0
Out  n=7 nyq=0      16.0   17.0   11.0   12.0   13.0   14.0   15.0
Out  n=7 nyq=1     -16.0  -17.0   11.0   12.0   13.0   14.0   15.0
Out  n=6 nyq=0     -15.0  -16.0   11.0   12.0   13.0   14.0
Out  n=6 nyq=1      15.0   16.0   11.0   12.0   13.0   14.0
```

The fourth test is to do a time derivative in the frequency domain with subroutine `ftderivslow()` /prog:ftderivslow. Here we do not have quite so clear an idea of what to expect. The natural position for a time derivative is to interlace the original data points. When we make the time derivative by multiplying in the frequency

domain by $-i\omega$, however, the derivative does not interlace the original **mesh**, but is on the same mesh. The time derivative of the small pulse we see here is the expected doublet aligned on the original mesh, and it has some unexpected high-frequency ripple that drops off slowly. The ripple resembles that on a pulse shifted half a mesh point, as in Figure 2.4. It happens that this rippling signal is an accurate representation of the derivative in many examples where such mesh alignment is needed, so (as with time shift) the ripple is worth having. Here again, we notice that there is an unfortunate **transient** on the ends of the data on two of the tests. But in two of the four tests below, the transient is so huge that it overwhelms the derivative of the small pulse in the middle of the signal.

```
Derivative tests:
In                10.0  10.0  10.0  10.0  12.0  10.0  10.0  10.0  10.0
Out  n=9 nyq=0  -0.7   0.8  -1.1   2.0   0.0  -2.0   1.1  -0.8   0.7
Out  n=9 nyq=1  13.5  -5.1   2.0   0.7   0.0  -0.7  -2.0   5.1 -13.5
Out  n=8 nyq=0  13.2  -5.7   3.5  -1.9   3.9  -6.6   7.6 -14.8
Out  n=8 nyq=1   0.0   0.3  -0.8   1.9   0.0  -1.9   0.8  -0.3
```

Examining all the tests, we conclude that if the data has an even number of points, it is best to include the **Nyquist frequency** in the frequency-domain repre-

sentation. If the data has an odd number of points, it is better to exclude the Nyquist frequency by interlacing it. A more positive way of summarizing our results is that the zero frequency should always be present. Given this conclusion, the next question is whether we should choose to use an even or an odd number of points.

The disadvantage of an even number of data values is that the programs that do frequency-domain manipulations will always need to handle Nyquist as a special case. The value at the Nyquist frequency must be handled as if half of it were at plus Nyquist and the other half at minus Nyquist. The Nyquist aggravation will get worse in two dimensions, where we have corners as well as edges. Figure 2.5 reproduces the four arrangements in Figure 2.1 along with a one-word summary of the suitability of each arrangement: “standard" for the standard arrangement, “risky" for arrangements that have end effects that are likely to be undesirable, and “best" for the arrangement that involves no risky end effects and no pesky Nyquist frequency.

Later in this chapter we will see the importance of using a *fast* FT program—one which is orders of magnitude faster than `slowft()` /prog:slowft. Unfortunately, among fast FT programs, I could not find one for an **odd-length transform** that is suitable for printing here, since odd-length FT programs seem to be many

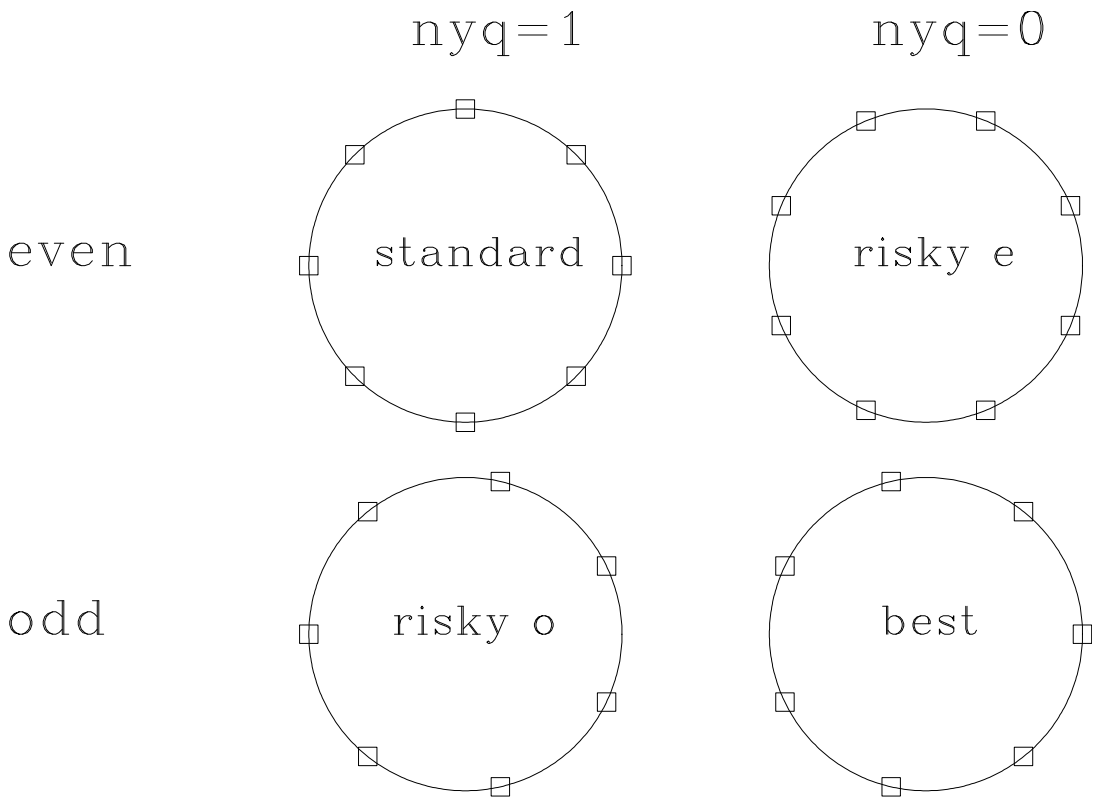


Figure 2.5: Evaluation of various arrangements of frequencies. dft-circeval [ER]

pages in length. So further applications in this book will use the even-length program. As a result, we will always need to fuss with the Nyquist frequency, making use of the frequency arrangement labeled “standard" and not that labeled “best."

A discrete Fourier-transform program designed for an *odd* number of points would make applications somewhat simpler. Alas, there seems to be no program for odd-length transforms that is both simple and fast.

2.2.3. FT by Z-transform

The program `slowft()` is unnecessarily slow, requiring us to compute a complex exponential at each step. By reorganizing easily using the Z-transform, the computational load can be reduced by about a factor of five (from a complex exponential to a complex multiply) at every step.

For simplicity we consider a signal that is only four points long:

$$B(\omega) \quad = \quad b_0 + b_1 Z + b_2 Z^2 + b_3 Z^3 \tag{2.10}$$

Reorganizing the polynomial (2.10) by nesting gives

$$B(\omega) \quad = \quad b_0 + Z(b_1 + Z(b_2 + Z(b_3))) \tag{2.11}$$

A subroutine for evaluating $B(\omega)$ in this way is `polyft()`. polyft

2.3. SYMMETRIES

Next we examine odd/even **symmetries** to see how they are affected in Fourier transform. The even part e_t of a signal b_t is defined as

$$e_t \quad = \quad \frac{b_t + b_{-t}}{2} \tag{2.12}$$

The odd part is

$$o_t \quad = \quad \frac{b_t - b_{-t}}{2} \tag{2.13}$$

By adding (2.12) and (2.13), we see that a function is the sum of its even and odd parts:

$$b_t \quad = \quad e_t + o_t \tag{2.14}$$

```
# Fourier transform by polynomial evaluation.
subroutine polyft( nt,tt, nw,cww )
integer nt      # number of points in the time domain
integer nw      # number of points in the fourier transform
real    tt(nt)  # sampled function of time
complex cww(nw) # sampled fourier transform
        integer it, iw
        real omega
        complex cz, cw
        do iw= 1, nw {
                omega =  3.14159265 * (iw-1.) / ( nw-1.)
                cz = cexp( cmplx( 0., omega ) )
                cw = tt(nt)
                do it= nt-1, 1, -1      # loop runs backwards
                        cw = cw * cz + tt(it)
                cww(iw) = cw
                }
        return; end
```

Back

Consider a simple, real, even signal such as $(b_{-1}, b_0, b_1) = (1, 0, 1)$. Its transform $Z + 1/Z = e^{i\omega} + e^{-i\omega} = 2\cos\omega$ is an even function of ω, since $\cos\omega = \cos(-\omega)$.

Consider the real, odd signal $(b_{-1}, b_0, b_1) = (-1, 0, 1)$. Its transform $Z - 1/Z = 2i\sin\omega$ is imaginary and odd, since $\sin\omega = -\sin(-\omega)$.

Likewise, the transform of the imaginary even function $(i, 0, i)$ is the imaginary even function $i2\cos\omega$. Finally, the transform of the imaginary odd function $(-i, 0, i)$ is real and odd.

Let r and i refer to real and imaginary, e and o to even and odd, and lower-case and upper-case letters to time and frequency functions. A summary of the symmetries of Fourier transform is shown in Figure 2.6.

More elaborate signals can be made by adding together the three-point functions we have considered. Since sums of even functions are even, and so on, the diagram in Figure 2.6 applies to all signals. An arbitrary signal is made from these four parts only, i.e., the function has the form $b_t = (\mathrm{re} + \mathrm{ro})_t + i(\mathrm{ie} + \mathrm{io})_t$. On transformation of b_t, each of the four individual parts transforms according to the table.

Most "industry standard" methods of Fourier transform set the zero frequency as the first element in the vector array holding the transformed signal, as implied

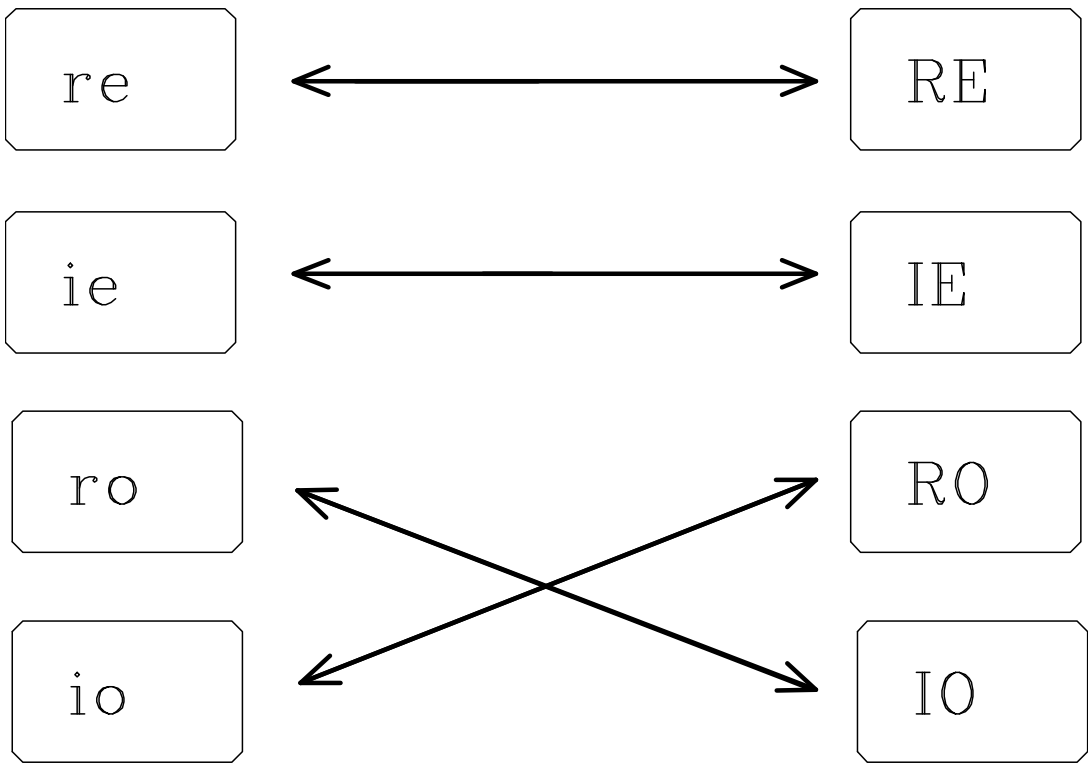


Figure 2.6: Odd functions swap real and imaginary. Even functions do not get mixed up with complex numbers. dft-reRE [NR]

by equation (2.3). This is a little inconvenient, as we saw a few pages back. The Nyquist frequency is then the first point past the middle of the even-length array, and the negative frequencies lie beyond. Figure 2.7 shows an example of an **even function** as it is customarily stored.

2.3.1. Plot interpretation

Now we will get away from the ends and think about what is in the middle of signals. Figure 2.7 shows even functions in both time and frequency domains. This figure was computed with the matrix equations (2.3) and (2.4). Displaying both the left and right halves of each function wastes half the paper; equivalently, for a fixed amount of paper, it wastes half the resolution. Typically, only the left half of each function is displayed. Accepting this form of display, we receive a bonus: each figure can be interpreted in two more ways.

Since imaginary parts are not shown, they are arbitrary. If you see only half of an axis, you cannot tell whether the function is even or odd or neither. A frequently occurring function is the "**causal**" function, i.e., the function that vanishes for $t < 0$. Its even part cancels its odd part on $t < 0$. The ro transforms to an IO, which, being

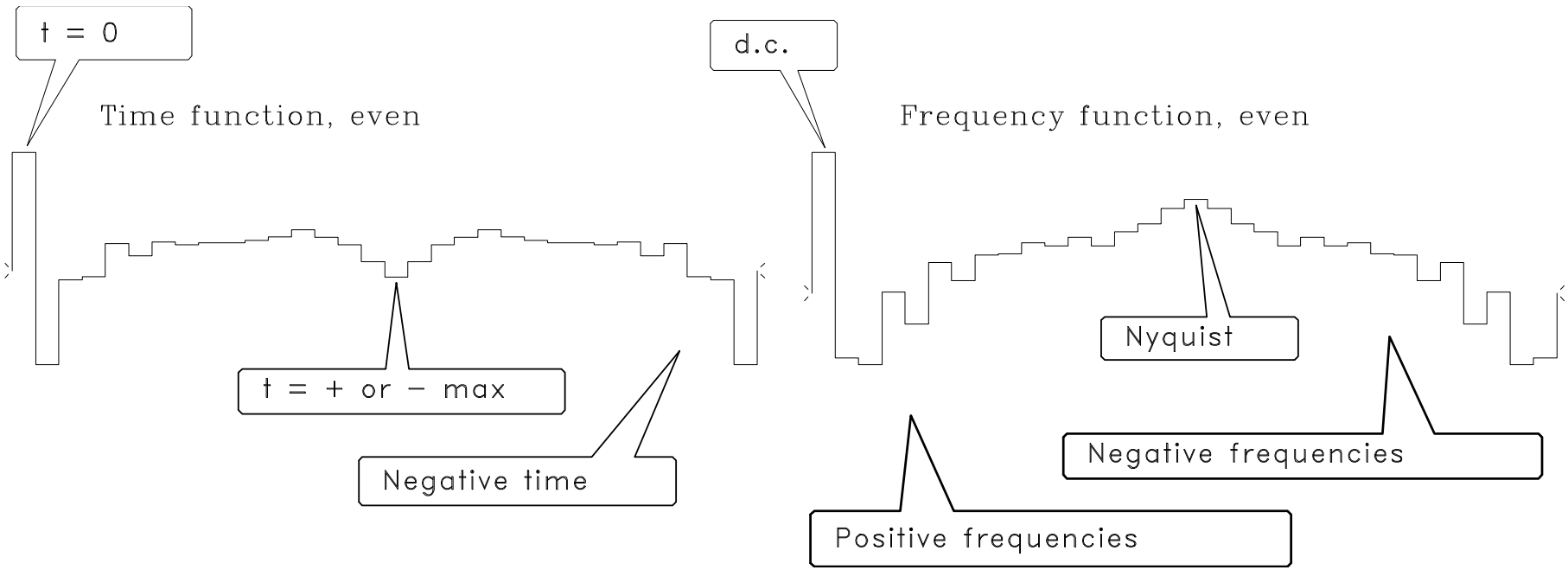


Figure 2.7: Even functions as customarily stored by “industry standard” FT programs. dft-even [NR]

imaginary, is not shown.

The third interpretation of these displays is that the frequency function is one-sided, and the time signal is complex. Such signals are called "**analytic signal**s." For analytic signals, RE extinguishes RO at negative ω, and the imaginary even part, ie, is not displayed.

In summary, plots that show only half the axes can be correctly interpreted in three ways:

left side	**right side**
even[$\Re f(t)$]	even[$\Re F(\omega)$]
$\Re$ causal(t)	$\Re F(\omega)$
$\Re f(t)$	$\Re$ OneSided(ω)

How can we compute these invisible imaginary parts? Their computation is called "**Hilbert transform**." Briefly, the Hilbert transform takes a *cosinusoidal* signal (like the real part of the FT of a delayed impulse, i.e., $\Re e^{i\omega t_0}$) and converts it to a *sinusoidal* signal of the same amplitude (like the imaginary part of a delayed impulse, $\Im e^{i\omega t_0}$).

2.3.2. Convolution in the frequency domain

Let $Y(Z) = X(Z)\,B(Z)$. The coefficients y_t can be found from the coefficients x_t and b_t by convolution in the time domain or by multiplication in the frequency domain. For the latter, we would evaluate both $X(Z)$ and $B(Z)$ at uniform locations around the unit circle, i.e., compute Fourier sums X_k and B_k from x_t and b_t. Then we would form $C_k = X_k B_k$ for all k, and inverse Fourier transform to y_t. The values y_t come out the same as by the time-domain convolution method, roughly that of our calculation **precision** (typically four-byte arithmetic or about one part in 10^{-6}). The only way in which you need to be cautious is to use **zero pad**ding greater than the combined lengths of x_t and b_t.

An example is shown in Figure 2.8. It is the result of a Fourier-domain computation which shows that the convolution of a rectangle function with itself gives a triangle. Notice that the triangle is clean—there are no unexpected end effects.

Because of the fast method of Fourier transform described next, the frequency-domain calculation is quicker when both $X(Z)$ and $B(Z)$ have more than roughly 20 coefficients. If either $X(Z)$ or $B(Z)$ has less than roughly 20 coefficients, then the time-domain calculation is quicker.

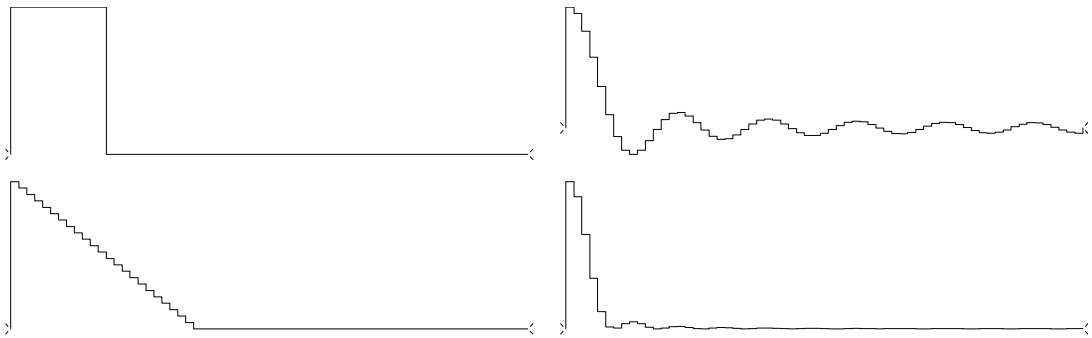


Figure 2.8: Top shows a rectangle transformed to a sinc. Bottom shows the sinc squared, back transformed to a triangle. dft-box2triangle [NR]

2.4. SETTING UP THE FAST FOURIER TRANSFORM

Typically we **Fourier transform** seismograms about a thousand points long. Under these conditions another Fourier summation method works about a hundred times faster than those already given. Unfortunately, the faster Fourier transform program is not so transparently clear as the programs given earlier. Also, it is slightly less flexible. The speedup is so overwhelming, however, that the fast program is always used in routine work.

Flexibility may be lost because the basic fast program works with complex-

```
integer function pad2( n )
integer n
pad2 = 1
while( pad2 < n )
        pad2 = pad2 * 2
return; end
```

Back

valued signals, so we ordinarily convert our real signals to complex ones (by adding a zero imaginary part). More flexibility is lost because typical fast FT programs require the data length to be an integral power of 2. Thus geophysical datasets often have zeros appended (a process called "**zero pad**ding") until the data length is a power of 2. From time to time I notice clumsy computer code written to deduce a number that is a power of 2 and is larger than the length of a dataset. An answer is found by rounding up the logarithm to base 2. The more obvious and the quicker way to get the desired value, however, is with the simple Fortran function `pad2()`.

pad2

How fast is the fast Fourier transform method? The answer depends on the size of the data. The matrix times vector operation in (2.3) requires N^2 multiplications and additions. That determines the speed of the slow transform. For the fast method the number of adds and multiplies is proportional to $N \log_2 N$. Since $2^{10} = 1024$, the speed ratio is typically 1024/10 or about 100. In reality, the fast method is not quite that fast, depending on certain details of overhead and implementation. In 1987 I tested the three programs on a 1024-point real signal and found times

`slowft`	153s
`polyft`	36s

`ftu` .7s

Below is `ftu()`, a version of the **fast Fourier transform** program. There are many versions of the program—I have chosen this one for its simplicity. Considering the complexity of the task, it is remarkable that no auxiliary memory vectors are required; indeed, the output vector lies on top of the input vector. To run this program, your first step might be to copy your real-valued signal into a complex-valued array. Then append enough zeros to fill in the remaining space. ftu

The following two lines serve to Fourier transform a vector of 1024 complex-valued points, and then to **inverse Fourier transform** them back to the original data:

```
call ftu(  1., 1024, cx)
call ftu( -1., 1024, cx)
```

An engineering reference given at the end of this chapter contains many other versions of the FFT program. One version transforms real-valued signals to complex-valued frequency functions in the interval $0 \leq \omega < \pi$. Others that do not transform data on top of itself may be faster with specialized computer architectures.

```
subroutine ftu( signi, nx, cx )
#   complex fourier transform with unitary scaling
#
#               1         nx          signi*2*pi*i*(j-1)*(k-1)/nx
#   cx(k)  =  -------- * sum cx(j) * e
#             sqrt(nx)   j=1             for k=1,2,...,nx=2**integer
#
integer nx, i, j, k, m, istep, pad2
real    signi, scale, arg
complex cx(nx), cmplx, cw, cdel, ct
if( nx != pad2(nx) )    call erexit('ftu: nx not a power of 2')
scale = 1. / sqrt( 1.*nx)
do i= 1, nx
        cx(i) = cx(i) * scale
j = 1;  k = 1
do i= 1, nx {
        if (i<=j) { ct = cx(j); cx(j) = cx(i); cx(i) = ct }
        m = nx/2
        while (j>m && m>1) { j = j-m; m = m/2 }         # "&&" means .AND.
        j = j+m
        }
repeat {
        istep = 2*k;   cw = 1.;   arg = signi*3.14159265/k
        cdel = cmplx( cos(arg), sin(arg))
        do m= 1, k {
                do i= m, nx, istep
                        { ct=cw*cx(i+k);  cx(i+k)=cx(i)-ct;  cx(i)=cx(i)+ct }
                cw = cw * cdel
                }
        k = istep
        if(k>=nx) break
        }
return; end
```

Back

EXERCISES:

1 Consider an even time function that is constant for all frequencies less than ω_0 and zero for all frequencies above ω_0. What is the rate of decay of amplitude with time for this function?

2 Waves spreading from a point source decay in energy as the area on a sphere. The amplitude decays as the square root of energy. This implies a certain decay in time. The time-decay rate is the same if the waves reflect from planar interfaces. To what power of time t do the signal amplitudes decay? For waves backscattered to the source from point reflectors, energy decays as distance to the minus fourth power. What is the associated decay with time?

2.4.1. Shifted spectra

Customarily, FT programs store frequencies in the interval $0 \leq \omega < 2\pi$. In some applications the interval $-\pi \leq \omega < \pi$ is preferable, and here we will see how this shift in one domain can be expressed as a product in the other domain. First we examine shifting by matrix multiplication. A single unit shift, wrapping the end

value around to the beginning, is

$$\begin{bmatrix} B_3 \\ B_0 \\ B_1 \\ B_2 \end{bmatrix} = \begin{bmatrix} . & . & . & 1 \\ 1 & . & . & . \\ . & 1 & . & . \\ . & . & 1 & . \end{bmatrix} \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} \tag{2.15}$$

You might recognize that equation (2.15) convolves a wavelet with a delayed impulse, where the bottom of the matrix is wrapped back in to the top to keep the output the same length as the input. For this 4×4 matrix, shifting one more point does the job of switching the high and low frequencies:

$$\begin{bmatrix} B_2 \\ B_3 \\ B_0 \\ B_1 \end{bmatrix} = \begin{bmatrix} . & . & 1 & . \\ . & . & . & 1 \\ 1 & . & . & . \\ . & 1 & . & . \end{bmatrix} \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} \tag{2.16}$$

We are motivated to seek an algebraic identity for the 4×4 matrix which represents the fact that convolution in the time domain is multiplication in the frequency domain. To this end we will look at the converse theorem, that multiplication in the time domain does shifting in the frequency domain. On the left of equation (2.17) is

the operation that first transforms from time to frequency and then swaps high and low frequencies. On the right is the operation that weights in the time domain, and then Fourier transforms. To verify the equation, multiply the matrices and simplify with $W^4 = 1$ to throw out all powers greater than 3.

$$\begin{bmatrix} . & . & 1 & . \\ . & . & . & 1 \\ 1 & . & . & . \\ . & 1 & . & . \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & W & W^2 & W^3 \\ 1 & W^2 & W^4 & W^6 \\ 1 & W^3 & W^6 & W^9 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & W & W^2 & W^3 \\ 1 & W^2 & W^4 & W^6 \\ 1 & W^3 & W^6 & W^9 \end{bmatrix} \begin{bmatrix} 1 & . \\ . & W^2 \\ . & . \\ . & . \end{bmatrix} \quad (2.17)$$

For an FT matrix of arbitrary size N, the desired shift is $N/2$, so values at alternate points in the time axis are multiplied by -1. A subroutine for that purpose is `fth()`. fth To Fourier transform a 1024-point complex vector `cx(1024)` and then inverse transform it, you would

```
call fth( 0, 1., 1, 1024, 1, cx)
call fth( 1, 1., 1, 1024, 1, cx)
```

You might wonder about the apparent redundancy of using both the argument `conj` and the argument `sign`. Having two arguments instead of one allows us to define the

```
# FT a vector in a matrix, with first omega = - pi
#
subroutine  fth( adj,sign, m1, n12, cx)
integer i,       adj,      m1, n12
real sign
complex cx(m1,n12)
temporary complex temp(n12)
do i= 1, n12
        temp(i) = cx(1,i)
if( adj == 0)   { do i= 2, n12, 2
                        temp(i) =  -temp(i)
                  call ftu(  sign, n12, temp)
                }
else            { call ftu( -sign, n12, temp)
                  do i= 2, n12, 2
                        temp(i) =  -temp(i)
                }
do i= 1, n12
        cx(1,i) = temp(i)
return; end
```

Back

forward transform for a *time* axis with the opposite sign as the forward transform for a *space* axis. The subroutine `fth()` is somewhat cluttered by the inclusion of a frequently needed practical feature—namely, the facility to extract vectors from a matrix, transform the vectors, and then restore them into the matrix.

2.5. TWO-DIMENSIONAL FT

The program `fth()` is set up so that the vectors transformed can be either rows or columns of a two-dimensional array. To see how this works, recall that in **Fortran** a matrix allocated as `(n1,n2)` can be subscripted as a matrix `(i1,i2)` or as a long vector `(i1 + n1*(i2-1),1)`, and `call sub(x(i1,i2))` passes the subroutine a pointer to the `(i1,i2)` element. To transform an entire axis, the subroutines `ft1axis()` and `ft2axis()` are given. For a two-dimensional FT, we simply call both `ft1axis()` and `ft2axis()` in either order. ft1axis ft2axis

I confess that there are faster ways to do things than those I have shown you above. When we are doing many FTs, for example, the overhead calculations done the first time should be saved for use on subsequent FTs, as in the subroutine `rocca()` included in IEI. Further, manufacturers of computers for heavy numeri-

```
# 1D Fourier transform on a 2D data set along the 1-axis
#
subroutine  ft1axis( adj, sign1, n1,n2, cx)
integer i2,          adj,        n1,n2
complex cx(n1,n2)
real sign1
do i2= 1, n2
                call fth( adj, sign1, 1,n1, cx(1,i2))
return; end
```

Back

```
# 1D Fourier transform on a 2D data set along the 2-axis
#
subroutine  ft2axis( adj, sign2, n1,n2, cx)
integer i1,          adj,        n1,n2
complex cx(n1,n2)
real    sign2
do i1= 1, n1
                call fth( adj, sign2, n1,n2, cx(i1,1))
return; end
```

Back

cal use generally design special FT codes for their architecture. Although the basic fast FT used here ingeniously stores its output on top of its input, that feature is not compatible with vectorizing architectures.

2.5.1. Basics of two-dimensional Fourier transform

Before going any further, let us review some basic facts about **two-dimensional Fourier transform**. A two-dimensional function is represented in a computer as numerical values in a matrix, whereas a one-dimensional Fourier transform in a computer is an operation on a vector. A 2-D Fourier transform can be computed by a sequence of 1-D Fourier transforms. We can first transform each column vector of the matrix and then each row vector of the matrix. Alternately, we can first do the

rows and later do the columns. This is diagrammed as follows:

$$
\begin{array}{ccc}
p(t,\ x) & \longleftrightarrow & P(t,\ k_x) \\
\updownarrow & & \updownarrow \\
P(\omega,\ x) & \longleftrightarrow & P(\omega,\ k_x)
\end{array}
$$

The diagram has the notational problem that we cannot maintain the usual convention of using a lower-case letter for the domain of physical space and an upper-case letter for the Fourier domain, because that convention cannot include the mixed objects $P(t,k_x)$ and $P(\omega,x)$. Rather than invent some new notation, it seems best to let the reader rely on the context: the arguments of the function must help name the function.

An example of **two-dimensional Fourier transform**s on typical deep-ocean data is shown in Figure 2.9. In the deep ocean, sediments are fine-grained and deposit slowly in flat, regular, horizontal beds. The lack of permeable rocks such as sandstone severely reduces the potential for petroleum production from the deep ocean. The fine-grained shales overlay irregular, igneous, **basement rock**s. In the

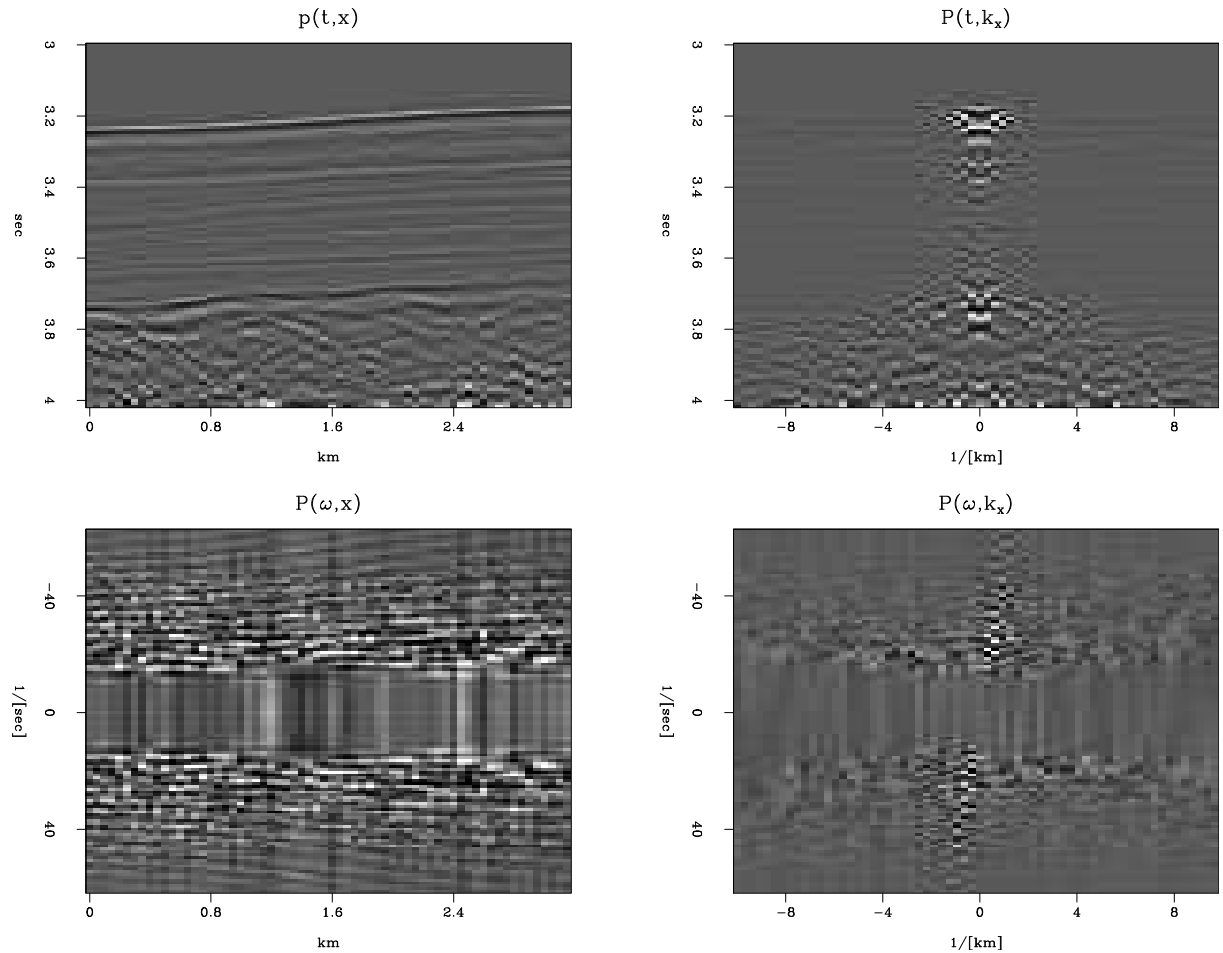


Figure 2.9: A deep-marine dataset $p(t,x)$ from Alaska (U.S. Geological Survey) and the *real* part of various Fourier transforms of it. Because of the long traveltime through the water, the time axis does not begin at $t = 0$. dft-plane4 [ER]

plot of $P(t,k_x)$, the lateral continuity of the sediments is shown by the strong spectrum at low k_x. The igneous rocks show a k_x spectrum extending to such large k_x that the deep data may be somewhat **spatially aliased** (sampled too coarsely). The plot of $P(\omega,x)$ shows that the data contains no low-frequency energy. The dip of the sea floor shows up in (ω,k_x)-space as the energy crossing the origin at an angle.

Altogether, the **two-dimensional Fourier transform** of a collection of seismograms involves only twice as much computation as the one-dimensional Fourier transform of each seismogram. This is lucky. Let us write some equations to establish that the asserted procedure does indeed do a 2-D Fourier transform. Say first that any function of x and t may be expressed as a superposition of sinusoidal functions:

$$p(t,x) \;=\; \int\int e^{-i\omega t + i k_x x}\; P(\omega,k_x)\, d\omega\, dk_x \tag{2.18}$$

The double integration can be nested to show that the temporal transforms are done first (inside):

$$p(t,x) \quad = \quad \int e^{i\,k_x x} \left[\int e^{-i\omega t}\; P(\omega,k_x)\, d\omega \right] dk_x$$

$$= \int e^{i k_x x} \, P(t, k_x) \, dk_x$$

The quantity in brackets is a Fourier transform over ω done for each and every k_x. Alternately, the nesting could be done with the k_x-integral on the inside. That would imply rows first instead of columns (or vice versa). It is the separability of $\exp(-i\omega t + i k_x x)$ into a product of exponentials that makes the computation this easy and cheap.

2.5.2. Signs in Fourier transforms

In Fourier transforming t-, x-, and z-coordinates, we must choose a sign convention for each coordinate. Of the two alternative **sign convention**s, electrical engineers have chosen one and physicists another. While both have good reasons for their choices, our circumstances more closely resemble those of physicists, so we will use their convention. For the *inverse* Fourier transform, our choice is

$$p(t,x,z) \quad = \quad \int\int\int e^{-i\omega t + i k_x x + i k_z z} \, P(\omega, k_x, k_z) \, d\omega \, dk_x \, dk_z \qquad (2.19)$$

For the *forward* Fourier transform, the space variables carry a *negative* sign, and time carries a *positive* sign.

Let us see the reasons why electrical engineers have made the opposite choice, and why we go with the physicists. Essentially, engineers transform only the time axis, whereas physicists transform both time and space axes. Both are simplifying their lives by their choice of sign convention, but physicists complicate their time axis in order to simplify their many space axes. The engineering choice minimizes the number of minus signs associated with the time axis, because for engineers, d/dt is associated with $i\omega$ instead of, as is the case for us and for physicists, with $-i\omega$. We confirm this with equation (2.19). Physicists and geophysicists deal with many more independent variables than time. Besides the obvious three space axes are their mutual combinations, such as midpoint and offset.

You might ask, why not make *all* the signs positive in equation (2.19)? The reason is that in that case waves would not move in a positive direction along the space axes. This would be especially unnatural when the space axis was a radius. Atoms, like geophysical sources, always radiate from a point to infinity, not the other way around. Thus, in equation (2.19) the sign of the spatial frequencies must be opposite that of the temporal frequency.

The only good reason I know to choose the engineering convention is that we might compute with an array processor built and microcoded by engineers. Conflict of sign convention is not a problem for the programs that transform complex-valued time functions to complex-valued frequency functions, because there the sign convention is under the user's control. But sign conflict does make a difference when we use any program that converts real-time functions to complex frequency functions. The way to live in both worlds is to imagine that the frequencies produced by such a program do not range from 0 to $+\pi$ as the program description says, but from 0 to $-\pi$. Alternately, we could always take the complex conjugate of the transform, which would swap the sign of the ω-axis.

2.5.3. Examples of 2-D FT

An example of a **two-dimensional Fourier transform** of a pulse is shown in Figure 2.10. Notice the location of the pulse. It is closer to the time axis than the frequency axis. This will affect the real part of the FT in a certain way (see exercises). Notice the broadening of the pulse. It was an impulse smoothed over time (vertically) by convolution with (1,1) and over space (horizontally) with (1,4,6,4,1).

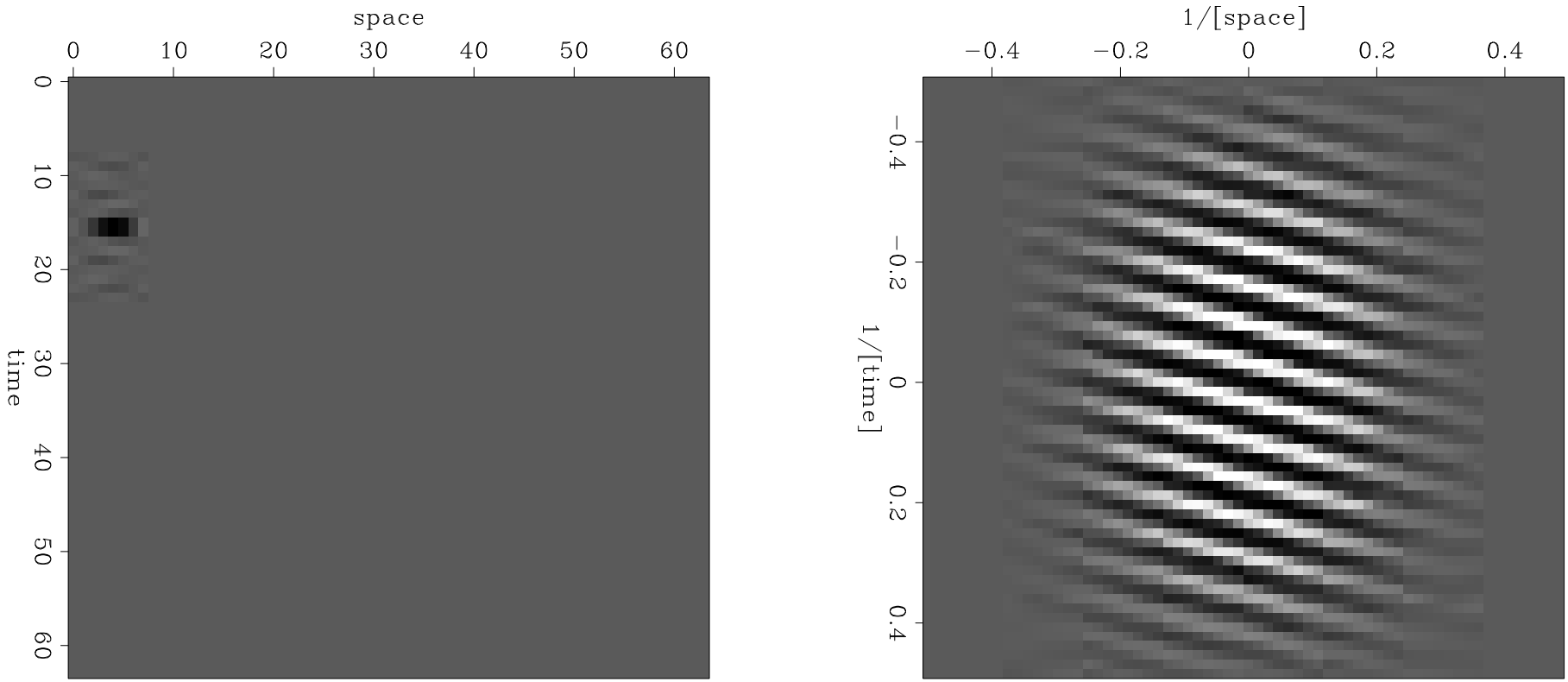


Figure 2.10: A broadened pulse (left) and the real part of its FT (right). dft-ft2dofpulse [ER]

This will affect the real part of the FT in another way.

Another example of a two-dimensional Fourier transform is given in Figure 2.11. This example simulates an impulsive air wave originating at a point on the x-axis. We see a wave propagating in each direction from the location of the source of the wave. In Fourier space there are also two lines, one for each wave. Notice that there are other lines which do not go through the origin; these lines are called "**spatial alias**es." Each actually goes through the origin of another square plane that is not shown, but which we can imagine alongside the one shown. These other planes are periodic replicas of the one shown.

EXERCISES:

1 Write `ftlag()` starting from `ftlagslow()` and `fth()`.

2 Most time functions are real. Their imaginary part is zero. Show that this means that $F(\omega, k)$ can be determined from $F(-\omega, -k)$.

3 What would change in Figure 2.10 if the pulse were moved (a) earlier on the t-axis, and (b) further on the x-axis? What would change in Figure 2.10 if instead the time axis were smoothed with (1,4,6,4,1) and the space axis with (1,1)?

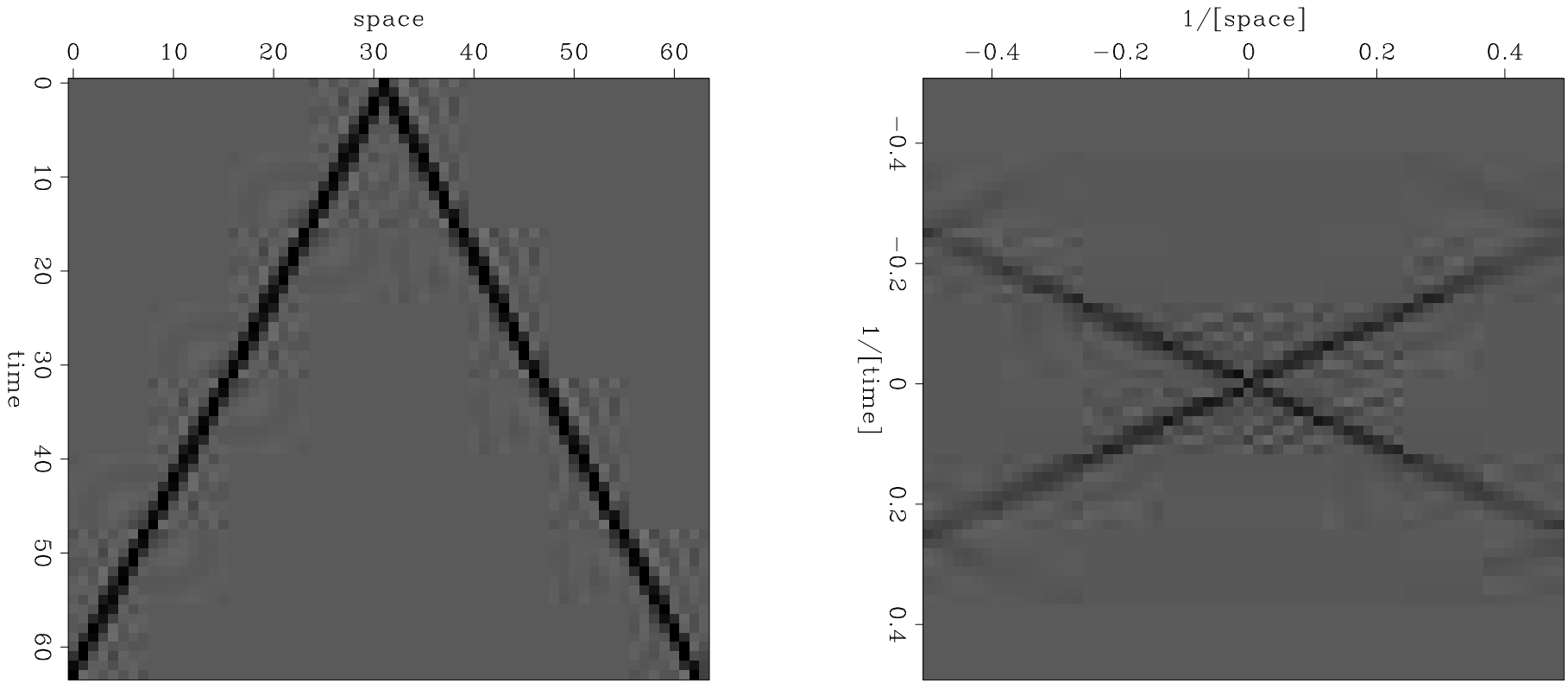


Figure 2.11: A simulated air wave (left) and the amplitude of its FT (right). dft-airwave [ER]

4 What would Figure 2.11 look like on an earth with half the earth velocity?

5 Numerically (or theoretically) compute the two-dimensional spectrum of a plane wave $[\delta(t - px)]$, where the plane wave has a randomly fluctuating amplitude: say, rand(x) is a random number between ± 1, and the randomly modulated plane wave is $[(1 + .2\,\text{rand}(x))\,\delta(t - px)]$.

6 Explain the horizontal "layering" in Figure 2.9 in the plot of $P(\omega, x)$. What determines the "layer" separation? What determines the "layer" slope?

2.6. HOW FAST FOURIER TRANSFORM WORKS

A basic building block in the **fast Fourier transform** is called "**doubling**." Given a series $(x_0, x_1, \ldots, x_{N-1})$ and its sampled Fourier transform $(X_0, X_1, \ldots, X_{N-1})$, and another series $(y_0, y_1, \ldots, y_{N-1})$ and its sampled Fourier transform $(Y_0, Y_1, \ldots, Y_{N-1})$, there is a trick to find easily the transform of the interlaced double-length series

$$z_t \quad = \quad (x_0, y_0, x_1, y_1, \ldots, x_{N-1}, y_{N-1}) \tag{2.20}$$

The process of doubling is used many times during the computing of a fast Fourier transform. As the word “doubling" might suggest, it will be convenient to suppose that N is an integer formed by raising 2 to some integer power. Suppose $N = 8 = 2^3$. We begin by dividing our eight-point series into eight separate series, each of length one. The Fourier transform of each of the one-point series is just the point. Next, we use doubling four times to get the transforms of the four different two-point series (x_0, x_4), (x_1, x_5), (x_2, x_6), and (x_3, x_7). We use doubling twice more to get the transforms of the two different four-point series (x_0, x_2, x_4, x_6) and (x_1, x_3, x_5, x_7). Finally, we use doubling once more to get the transform of the original eight-point series $(x_0, x_1, x_2, \ldots, x_7)$. It remains to look into the details of the doubling process. Let

$$V = e^{i2\pi/2N} = W^{1/2} \tag{2.21}$$

$$V^N = e^{i\pi} = -1 \tag{2.22}$$

By definition, the transforms of two N-point series are

$$X_k = \sum_{j=0}^{N-1} x_j V^{2jk} \quad (k = 0, 1, \ldots, N-1) \tag{2.23}$$

$$Y_k = \sum_{j=0}^{N-1} y_j V^{2jk} \quad (k = 0, 1, \ldots, N-1) \tag{2.24}$$

Likewise, the transform of the interlaced series $z_j = (x_0, y_0, x_1, y_1, \ldots, x_{N-1}, y_{N-1})$ is

$$Z_k = \sum_{l=0}^{2N-1} z_l V^{lk} \quad (k = 0, 1, \ldots, 2N-1) \tag{2.25}$$

To make Z_k from X_k and Y_k, we require two separate formulas, one for $k = 0$, 1, …, $N-1$, and the other for $k = N$, $N+1$, …, $2N-1$. Start from the sum

$$Z_k = \sum_{l=0}^{2N-1} z_l V^{lk} \quad (k = 0, 1, \ldots, N-1) \tag{2.26}$$

and then split the sum into two parts, noting that x_j multiplies even powers of V, and y_j multiplies odd powers:

$$Z_k = \sum_{j=0}^{N-1} x_j V^{2jk} + V^k \sum_{j=0}^{N-1} y_j V^{2jk} \tag{2.27}$$

$$= \quad X_k + V^k Y_k \tag{2.28}$$

We obtain the last half of the Z_k by

$$Z_k = \sum_{l=0}^{2N-1} z_l V^{lk} \quad (k = N, N+1, \ldots, 2N-1) \tag{2.29}$$

$$= \sum_{l=0}^{2N-1} z_l V^{l(m+N)} \quad (k - N = m = 0, 1, \ldots, N-1) \tag{2.30}$$

$$= \sum_{l=0}^{2N-1} z_l V^{lm} (V^N)^l \tag{2.31}$$

$$= \sum_{l=0}^{2N-1} z_l V^{lm} (-1)^l \tag{2.32}$$

$$= \sum_{j=0}^{N-1} x_j V^{2jm} - V^m \sum_{j=0}^{N-1} y_j V^{2jm} \tag{2.33}$$

$$= X_m - V^m Y_m \tag{2.34}$$

$$Z_k = X_{k-N} - V^{k-N} Y_{k-N} \quad (k = N, N+1, \ldots, 2N-1) \tag{2.35}$$

The subroutine `ftu()` /prog:ftu does not follow this analysis in detail.

If you would like some industrial grade FFT programs, search the web for "prime factor FFT".

2.7. References

Special issue on fast Fourier transform, June 1969: IEEE Trans. on Audio and Electroacoustics (now known as IEEE Trans. on Acoustics, Speech, and Signal Processing), **AU-17**, entire issue (66-172).

Chapter 3

Z-plane, causality, and feedback

All physical systems share the property that they do not respond before they are excited. Thus the impulse response of any physical system is a one-sided time function (it vanishes before $t = 0$). In system theory such a filter function is called

“**realizable**" or “**causal**.” In wave propagation this property is associated with *causality* in that no wave may begin to arrive before it is transmitted. The lag-time point $t = 0$ plays a peculiar and an important role. Thus many subtle matters can be more clearly understood with sampled time than with continuous time. When a filter responds at and after lag time $t = 0$, we say the filter is realizable or causal. The word “causal" is appropriate in physics, where stress causes instantaneous strain and vice versa, but one should return to the less pretentious words “realizable" or “one-sided" when using filter theory to describe economic or social systems where simultaneity is different from cause and effect.

The other new concept in this chapter is “**feedback**." Ordinarily a filter produces its output using only past *inputs*. A filter using feedback uses also its past *outputs*. After digesting the feedback concept, we will look at a wide variety of filter types, at what they are used for, and at how to implement them.

First a short review: the Z-transform of an arbitrary, time-discretized signal x_t is defined by

$$X(Z) \quad = \quad \cdots + x_{-2}\,Z^{-2} + x_{-1}\,Z^{-1} + x_0 + x_1\,Z + x_2\,Z^2 + \cdots \qquad (3.1)$$

In chapter 1 we saw that (3.1) can be understood as a Fourier sum (where $Z = e^{i\omega}$).

It is not necessary for Z to take on numerical values, however, in order for the ideas of convolution and correlation to be useful. In chapter 1 we defined Z to be the unit delay operator. Defined thus, Z^2 delays two time units. Expressions like $X(Z)\,B(Z)$ and $X(Z)\,B(1/Z)$ are useful because they imply convolution and crosscorrelation of the time-domain coefficients. Here we will be learning how to interpret $1/A(Z)$ as a feedback filter, i.e., as a filter that processes not only past inputs, but past outputs. We will see that this approach brings with it interesting opportunities as well as subtle pitfalls.

3.1. LEAKY INTEGRATION

The convolution equation (1.9)

$$y_k \quad = \quad \sum_{i=0} x_{k-i}\, b_i \tag{3.2}$$

says that the present output is created entirely from present and past values of the *input*. Now we will include past values of the *output*. The simplest example is

numerical integration, such as

$$y_t \quad = \quad y_{t-1} + x_t \tag{3.3}$$

Notice that when $x_t = (0,0,0,1,0,0,\cdots)$, $y_t = (0,0,0,1,1,1,1,\cdots)$, which shows that the integral of an impulse is a step.

A kind of deliberately imperfect integration used in numerical work is called "**leaky integration**." The name derives from the analogous situation of electrical circuits, where the voltage on a capacitor is the integral of the current: in real life, some of the current leaks away. An equation to model leaky integration is

$$y_t \quad = \quad \rho\, y_{t-1} + x_t \tag{3.4}$$

where ρ is a constant that is slightly less than plus one. Notice that if ρ were greater than unity, the output of (3.4) would grow with time instead of decaying. A program for this simple operation is `leak()`. I use this program so frequently that I wrote it so the output could be overlaid on the input. `leak()` uses a trivial subroutine, `copy()` /prog:copy, for copying. leak

Let us see what Z-transform equation is implied by (3.4). Move the y terms to the left:

$$y_t - \rho\, y_{t-1} \quad = \quad x_t \tag{3.5}$$

```
subroutine leak( rho, n, xx, yy)
integer i, n;   real xx(n), yy(n), rho
temporary real tt( n)
call     null( tt, n)
tt(1) = xx(1)
do i= 2, n
        tt(i) = rho * tt(i-1) + xx(i)
call copy( n, tt, yy)
return; end
```

Back

Given the Z-transform equation

$$(1-\rho Z)\,Y(Z) \quad = \quad X(Z) \tag{3.6}$$

notice that (3.5) can be derived from (3.6) by finding the coefficient of Z^t. Thus we can say that the output $Y(Z)$ is derived from the input $X(Z)$ by the polynomial division

$$Y(Z) \quad = \quad \frac{X(Z)}{1-\rho Z} \tag{3.7}$$

Therefore, the effective filter $B(Z)$ in $Y(Z) = B(Z)X(Z)$ is

$$B(Z) \quad = \quad \frac{1}{1-\rho Z} \quad = \quad 1+\rho Z+\rho^2 Z^2+\rho^3 Z^3+\cdots \tag{3.8}$$

The left side of Figure 3.1 shows a damped exponential function that consists of the coefficients ρ^t seen in equation (3.8). The spectrum of b_t is defined by $\overline{B}(1/Z)B(Z)$. The **amplitude spectrum** is the square root of the spectrum. It can be abbreviated by $|B(Z)|$. The amplitude spectrum is plotted on the right side of Figure 3.1. Ordinary integration has a Fourier response $1/(-i\omega)$ that blows up at $\omega = 0$. Leaky integration smooths off the infinite value at $\omega = 0$. Thus in the figure, the amplitude spectrum looks like $|1/\omega|$, except that it is not ∞ at $\omega = 0$.

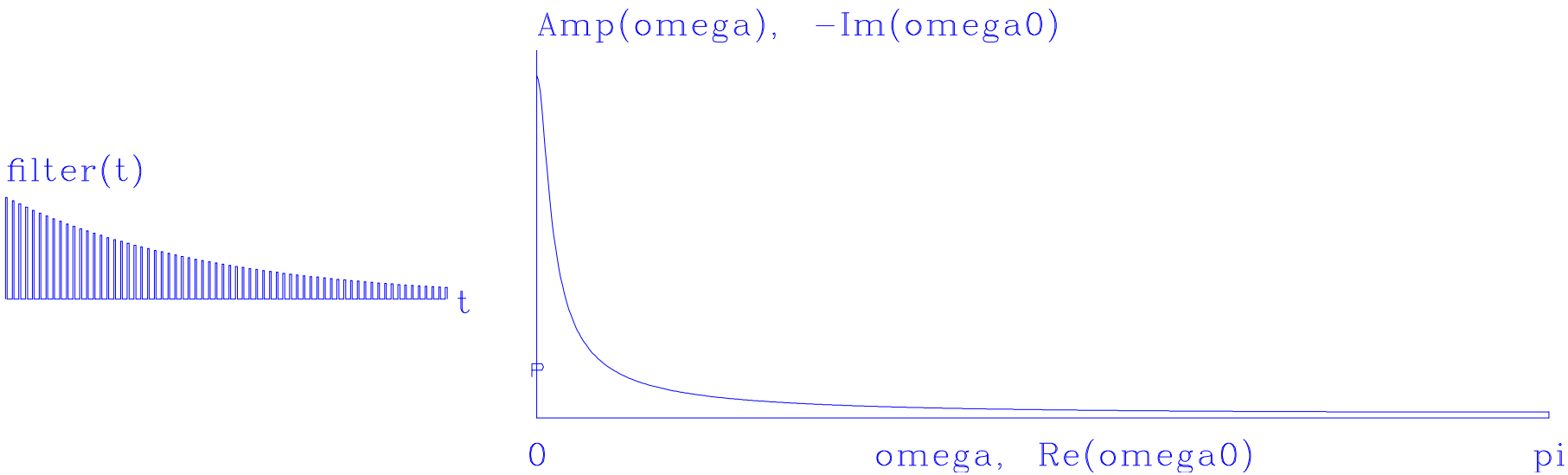


Figure 3.1: Left is the impulse response of leaky integration. Right is the amplitude $1/|1-\rho Z|$ in the Fourier domain. zp-leak [NR]

3.1.1. Plots

A **pole** is a place in the complex plane where a filter $B(Z_p)$ becomes infinity. This occurs where a denominator vanishes. For example, in equation (3.8) we see that there is one pole and it is located at $Z_p = 1/\rho$. In plots like Figure 3.1, a pole location is denoted by a “p” and a zero location by a “z." I chose to display the **pole** and **zero** locations in the ω_0-plane instead of in the Z_0-plane. Thus real frequencies run along the horizontal axis instead of around the circle of $|Z| = 1$. I further chose to superpose the complex ω_0-plane on the graph of $|F(\omega)|$ versus ω. This enables us to correlate the pole and zero locations to the spectrum. I plotted $(\Re\omega_0, -\Im\omega_0)$ in order that the ω and $\Re\omega_0$ axes would coincide. As we will see later, some poles give stable filters and some poles give unstable filters. At the risk of some confusion, I introduced the minus sign to put the stable poles atop the positive spectrum. Since we will never see a negative spectrum and we will rarely see an unstable pole, this economizes on paper (or maximizes resolution for a fixed amount of paper).

In Figure 3.1, moving the “p” down toward the horizontal axis would cause a slower time decay and a sharper frequency function.

3.1.2. Two poles

Integration twice is an operation with two **pole**s. Specifically,

$$\frac{1}{(1-Z)^2} = (1+Z+Z^2+Z^3+\cdots)(1+Z+Z^2+Z^3+\cdots) = 1+2Z+3Z^2+4Z^3+5Z^4 \tag{3.9}$$

Notice that the signal is $(1,2,3,\cdots)$, which is a discrete representation of the function $f(t) = t\,\text{step}(t)$. Figure 3.2 shows the result when the two integrations are leaky integrations. We see the signal begin as t but then drop off under some weight that looks exponential. A second time-derivative filter $(-i\omega)^2$ has an amplitude spectrum $|\omega^2|$. Likewise, a second integration has an amplitude spectrum $|1/\omega^2|$, which is about what we see in Figure 3.2, except that at $\omega = 0$ leaky integration has rounded off the ∞.

Instead of allowing two poles to sit on top of each other (which would look like just one pole), I moved the pole slightly off $\Re\omega = 0$ so that $\Re\omega > 0$. As in Figure **??**, another pole is included (but not shown) at negative frequency. This extra pole is required to keep the signal real. Of course the two poles are very close to each other. The reason I chose to split them this way is to prepare you for filters where the poles are far apart.

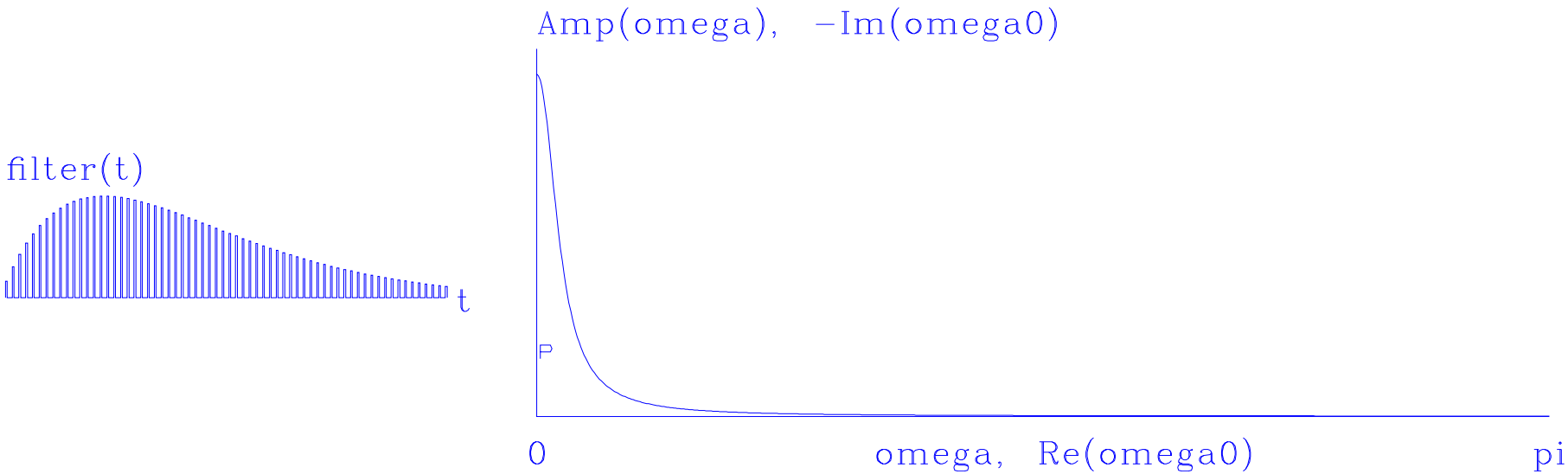


Figure 3.2: A cascade of two leaky integrators. zp-leak2 [NR]

EXERCISES:

1 Show that multiplication by $(1 - Z)$ in discretized time is analogous to time differentiation in continuous time. Show that dividing by $(1 - Z)$ is analogous to integration. What are the limits on the integral?

2 A simple feedback operation is $y_t = (1 - \epsilon)y_{t-1} + x_t$. Give a closed-form expression for the output y_t if x_t is an impulse. Estimate the decay time τ of your solution (the time it takes for y_t to drop to $e^{-1}y_0$)? For small ϵ, say = 0.1, .001, or 0.0001, what is τ?

3 Find an analytic expression for the plot on the right side of Figure 3.1 as a function of ω. Show that it is like $1/|\omega|$.

4 In continuous time, the signal analogous to that in Figure 3.2 is te^{-t}. What is the analogous frequency function?

3.2. SMOOTHING WITH BOX AND TRIANGLE

Simple "**smoothing**" is a common application of filtering. A smoothing filter is one with all positive coefficients. On the time axis, smoothing is often done with a

single-pole damped exponential function. On space axes, however, people generally prefer a symmetrical function. We will begin with rectangle and triangle functions. When the function width is chosen to be long, then the computation time can be large, but recursion can shorten it immensely.

3.2.1. Smoothing with a rectangle

The inverse of any polynomial reverberates forever, although it might drop off fast enough for any practical need. On the other hand, a rational filter can suddenly drop to zero and stay there. Let us look at a popular rational filter, the rectangle or "**box car**":

$$\frac{1-Z^5}{1-Z} \quad = \quad 1+Z+Z^2+Z^3+Z^4 \tag{3.10}$$

The filter (3.10) gives a moving average under a *rectangular* window. This is a basic smoothing filter. A clever way to apply it is to move the rectangle by adding a new value at one end while dropping an old value from the other end. This approach is formalized by the polynomial division algorithm, which can be simplified because so many coefficients are either one or zero. To find the recursion associated with

$Y(Z) = X(Z)(1 - Z^5)/(1 - Z)$, we identify the coefficient of Z^t in $(1 - Z)Y(Z) = X(Z)(1 - Z^5)$. The result is

$$y_t \quad = \quad y_{t-1} + x_t - x_{t-5} \tag{3.11}$$

This approach boils down to the program `boxconv()` /prog:boxconv, which is so fast it is almost free! boxconv Its last line scales the output by dividing by the rectangle length. With this scaling, the zero-frequency component of the input is unchanged, while other frequencies are suppressed.

Let us examine the pole and zero locations in equation (3.10). The denominator vanishes at $Z = 1$, so the filter has a pole at zero frequency. Smoothing something is like boosting frequencies near the zero frequency. The numerator vanishes at the five roots of unity, i.e., $Z = e^{i2\pi n/5}$. These five locations are uniformly spaced around the unit circle. Any sinusoid at exactly one of these frequencies is exactly destroyed by this filter, because such a sinusoid has an integer number of wavelengths under the boxcar. An exception is the zero frequency, where the root at $Z = 1$ is canceled by a pole at the same location. This cancellation is the reason the right-hand side ends at the fourth power—there is no infinite series of higher powers.

```
subroutine boxconv( nb, nx, xx, yy)
# inputs:       nx,  xx(i), i=1,nx      the data
#               nb                      the box length
# output:       yy(i),i=1,nx+nb-1       smoothed data
integer nx, ny, nb, i
real xx(nx), yy(1)
temporary real bb(nx+nb)
if( nb < 1 || nb > nx)  call erexit('boxconv')  # "||" means .OR.
ny = nx+nb-1
do i= 1, ny
        bb(i) = 0.
bb(1) = xx(1)
do i= 2, nx
                bb(i) = bb(i-1) + xx(i)         # make B(Z) = X(Z)/(1-Z)
do i= nx+1, ny
                bb(i) = bb(i-1)
do i= 1, nb
                yy(i) = bb(i)
do i= nb+1, ny
                yy(i) = bb(i) - bb(i-nb)        # make Y(Z) = B(Z)*(1-Z**nb)
do i= 1, ny
                yy(i) = yy(i) / nb
return; end
```

Back

3.2.2. Smoothing with a triangle

Triangle smoothing is rectangle smoothing done twice. For a mathematical description of the triangle filter, we simply square equation (3.10). Convolving a rectangle function with itself many times yields a result that mathematically tends towards a **Gaussian** function. Despite the sharp corner on the top of the triangle function, it has a shape that is remarkably similar to a Gaussian, as we can see by looking at Figure 11.2.

With filtering, **end effect**s can be a nuisance. Filtering increases the length of the data, but people generally want to keep input and output the same length (for various practical reasons). This is particularly true when filtering a space axis. Suppose the five-point signal $(1,1,1,1,1)$ is smoothed using the `boxconv()` program with the three-point smoothing filter $(1,1,1)/3$. The output is $5+3-1$ points long, namely, $(1,2,3,3,3,2,1)/3$. We could simply abandon the points off the ends, but I like to **fold** them back in, getting instead $(1+2,3,3,3,1+2)$. An advantage of the folding is that a constant-valued signal is unchanged by the smoothing. This is desirable since a smoothing filter is a low-pass filter which naturally should pass the lowest frequency $\omega = 0$ without distortion. The result is like a wave reflected by a **zero-slope** end condition. Impulses are smoothed into triangles except near the

boundaries. What happens near the boundaries is shown in Figure 3.3. Note that at

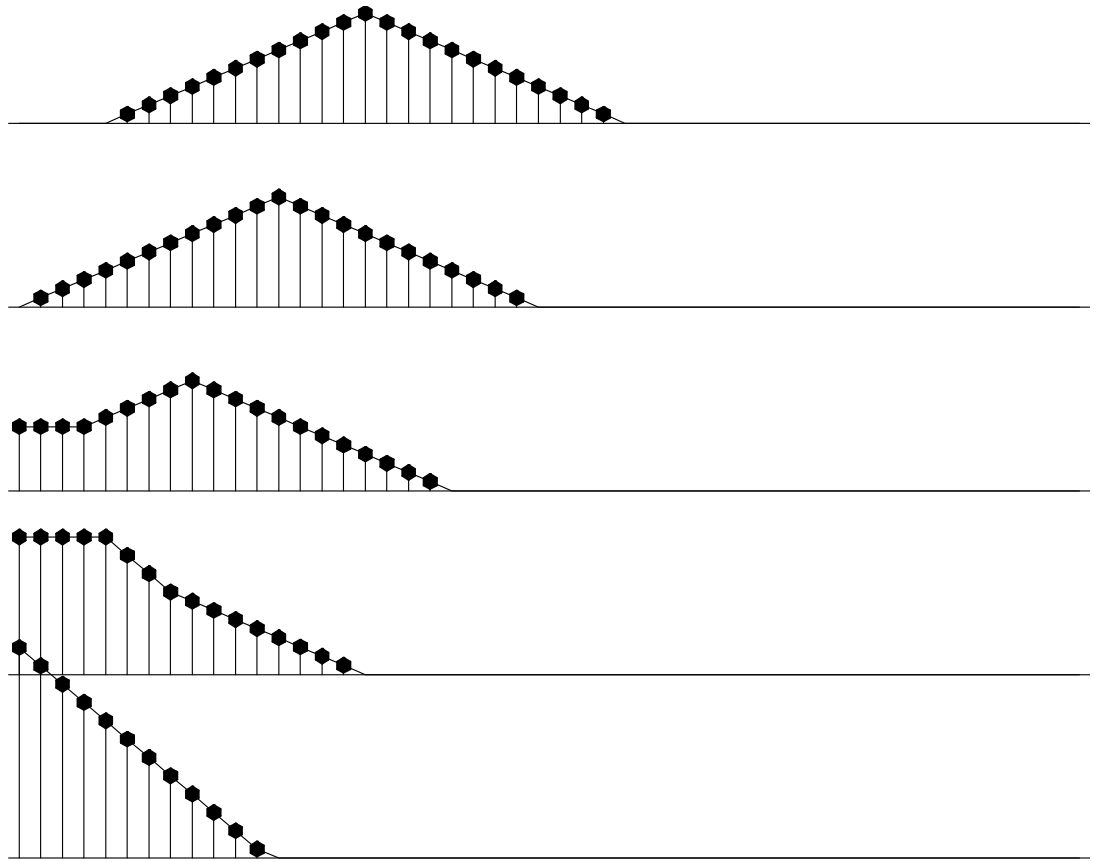


Figure 3.3: Edge effects when smoothing an impulse with a triangle function. Inputs are spikes at various distances from the edge. zp-triend [ER]

the boundary, there is necessarily only half a triangle, but it is twice as tall.

Figure 3.3 was derived from the routine `triangle()`. triangle I frequently

```
# Convolve with triangle
#
subroutine triangle( nr, m1, n12, uu, vv)
# input:        nr      rectangle width (points) (Triangle base twice as wide.)
# input:        uu(m1,i2),i2=1,n12      is a vector of data.
# output:       vv(m1,i2),i2=1,n12      may be on top of uu
integer nr,m1,n12, i,np,nq
real uu( m1, n12),  vv( m1, n12)
temporary real pp(n12+nr-1), qq(n12+nr+nr-2), tt(n12)
do i=1,n12 {    qq(i) = uu(1,i) }
if( n12 == 1 )
        call copy( n12, qq, tt)
else {
        call boxconv( nr, n12, qq, pp);         np = nr+n12-1
        call boxconv( nr, np , pp, qq);         nq = nr+np-1
        do i= 1, n12
                        tt(i) = qq(i+nr-1)
        do i= 1, nr-1                                   # fold back near end
                        tt(i) = tt(i) + qq(nr-i)
        do i= 1, nr-1                                   # fold back far end
                        tt(n12-i+1) = tt(n12-i+1) + qq(n12+(nr-1)+i)
        }
do i=1,n12 {    vv(1,i) = tt(i) }
return; end
```

Back

```
# smooth by convolving with triangle in two dimensions.
#
subroutine triangle2( rect1, rect2, n1, n2, uu, vv)
integer i1,i2,        rect1, rect2, n1, n2
real uu(n1,n2), vv(n1,n2)
temporary real ss(n1,n2)
do i1= 1, n1
                call triangle( rect2, n1, n2, uu(i1,1), ss(i1,1))
do i2= 1, n2
                call triangle( rect1,  1, n1, ss(1,i2), vv(1,i2))
return; end
```

Back

use this program, so it is cluttered with extra features. For example, the output can share the same location as the input. Further, since it is commonly necessary to smooth along the 2-axis of a two-dimensional array, there are some Fortran-style pointer manipulations to allow the user to smooth either the 1-axis or the 2-axis. For those of you unfamiliar with Fortran matrix-handling tricks, I include below another routine, `triangle2()`, that teaches how a two-dimensional array can be smoothed over both its 1-axis and its 2-axis. Some examples of two-dimensional smoothing are given in chapter 11. triangle2

EXERCISES:

1 The Fourier transform of a rectangle function is $\sin(\alpha t)/\alpha t$, also known as a "sinc" function. In terms of α, how wide is the rectangle function?

2 Express $Z^{-2} + Z^{-1} + 1 + Z + Z^2$ in the ω-domain. This is a discrete representation of a rectangle function. Identify the ways in which it is similar to and different from the sinc function.

3 Explain the signal second from the bottom in Figure 3.3.

4 Sketch the spectral response of the subroutine `triangle()` /prog:triangle.

3.3. CAUSAL INTEGRATION FILTER

Begin with a function in discretized time x_t. The Fourier transform with the substitution $Z = e^{i\omega\,\Delta t}$ is the Z-transform

$$X(Z) \quad = \quad \cdots + x_{-2}\,Z^{-2} \,+\, x_{-1}\,Z^{-1} \,+\, x_0 \,+\, x_1\,Z \,+\, x_2\,Z^2 \,+\, \cdots \qquad (3.12)$$

Define $-i\hat{\omega}$ (which will turn out to be an approximation to $-i\omega$) by

$$\frac{1}{-i\hat{\omega}\,\Delta t} \quad = \quad \frac{1}{2}\,\frac{1\,+\,Z}{1\,-\,Z} \qquad (3.13)$$

Define another signal y_t with Z-transform $Y(Z)$ by applying the operator to $X(Z)$:

$$Y(Z) \quad = \quad \frac{1}{2}\,\frac{1\,+\,Z}{1\,-\,Z}\,X(Z) \qquad (3.14)$$

Multiply both sides by $(1-Z)$:

$$(1\,-\,Z)\,Y(Z) \quad = \quad \frac{1}{2}\,(1\,+\,Z)\,X(Z) \qquad (3.15)$$

Equate the coefficient of Z^t on each side:

$$y_t \,-\, y_{t-1} \quad = \quad \frac{x_t \,+\, x_{t-1}}{2} \qquad (3.16)$$

Taking x_t to be an impulse function, we see that y_t turns out to be a step function, that is,

$$x_t = \cdots 0,0,0,0,0,1,0,0,0,0,0,0,\cdots \tag{3.17}$$

$$y_t = \cdots 0,0,0,0,0,\frac{1}{2},1,1,1,1,1,1,\cdots \tag{3.18}$$

So y_t is the discrete-domain representation of the integral of x_t from minus infinity to time t. The operator $(1+Z)/(1-Z)$ is called the "**bilinear transform**."

3.3.1. The accuracy of causal integration

The accuracy of the approximation of $\hat{\omega}$ to ω can be seen by dividing the top and bottom of equation (3.13) by $\sqrt{Z}$ and substituting $Z = e^{i\omega\Delta t}$:

$$-i\,\frac{\hat{\omega}\,\Delta t}{2} = \frac{1-Z}{1+Z} \tag{3.19}$$

$$-i\,\frac{\hat{\omega}\,\Delta t}{2} = \frac{1/\sqrt{Z}-\sqrt{Z}}{1/\sqrt{Z}+\sqrt{Z}} = -i\,\frac{\sin\frac{\omega\,\Delta t}{2}}{\cos\frac{\omega\,\Delta t}{2}} = -i\,\tan\frac{\omega\,\Delta t}{2} \tag{3.20}$$

$$\frac{\hat{\omega}\,\Delta t}{2} = \tan\frac{\omega\,\Delta t}{2} \tag{3.21}$$

$$\hat{\omega} \approx \omega \tag{3.22}$$

This is a valid approximation at low frequencies.

3.3.2. Examples of causal integration

The integration operator has a pole at $Z = 1$, which is exactly on the **unit circle** $|Z| = 1$. The implied zero division has paradoxical implications (page 166) that are easy to avoid by introducing a small positive number ϵ and defining $\rho = 1 - \epsilon$. The integration operator becomes

$$I(Z) = \frac{1}{2}\,\frac{1+\rho Z}{1-\rho Z} \tag{3.23}$$

$$I(Z) = \frac{1}{2}\,(1+\rho Z)\left[1+\rho Z+(\rho Z)^2+(\rho Z)^3+\cdots\right]$$

$$I(Z) = \frac{1}{2}+\rho Z+(\rho Z)^2+(\rho Z)^3+\cdots \tag{3.24}$$

Because ρ is less than one, this series converges for any value of Z on the unit circle. If ϵ had been slightly negative instead of positive, a converging expansion could have been carried out in negative powers of Z. A plot of $I(Z)$ is found in Figure 3.4.

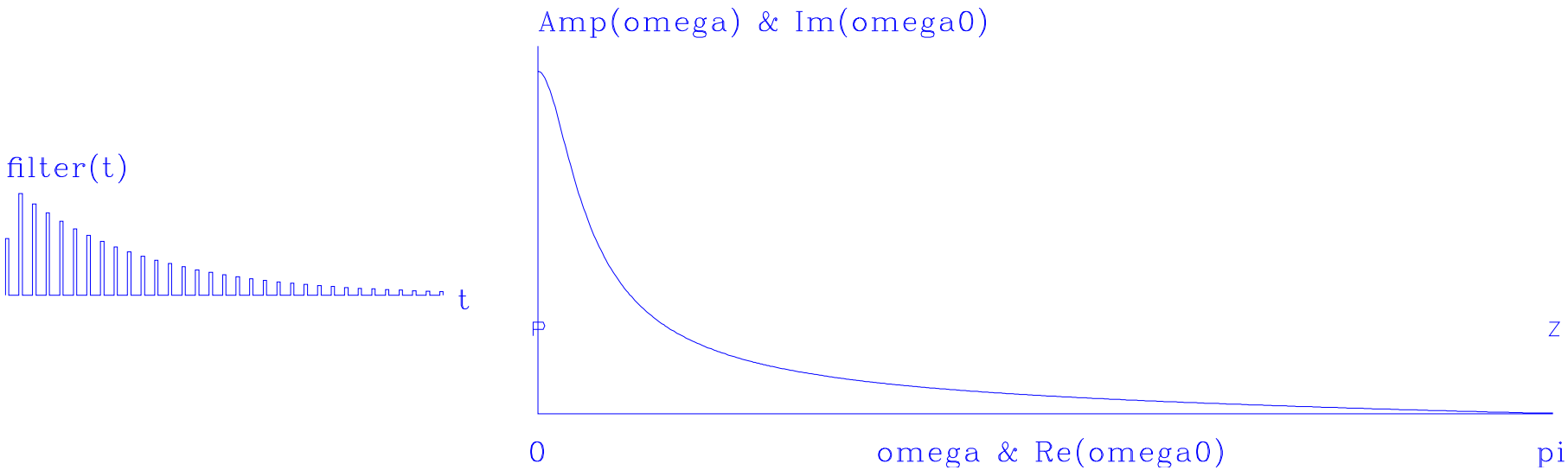


Figure 3.4: A leaky causal-integration operator I. zp-cint [NR]

Just for fun I put random noise into an integrator to see an economic simulation, shown in Figure 3.5. With $\rho = 1$, the difference between today's price and tomor-

row’s price is a random number. Thus the future price cannot be predicted from the past. This curve is called a “**random walk**."

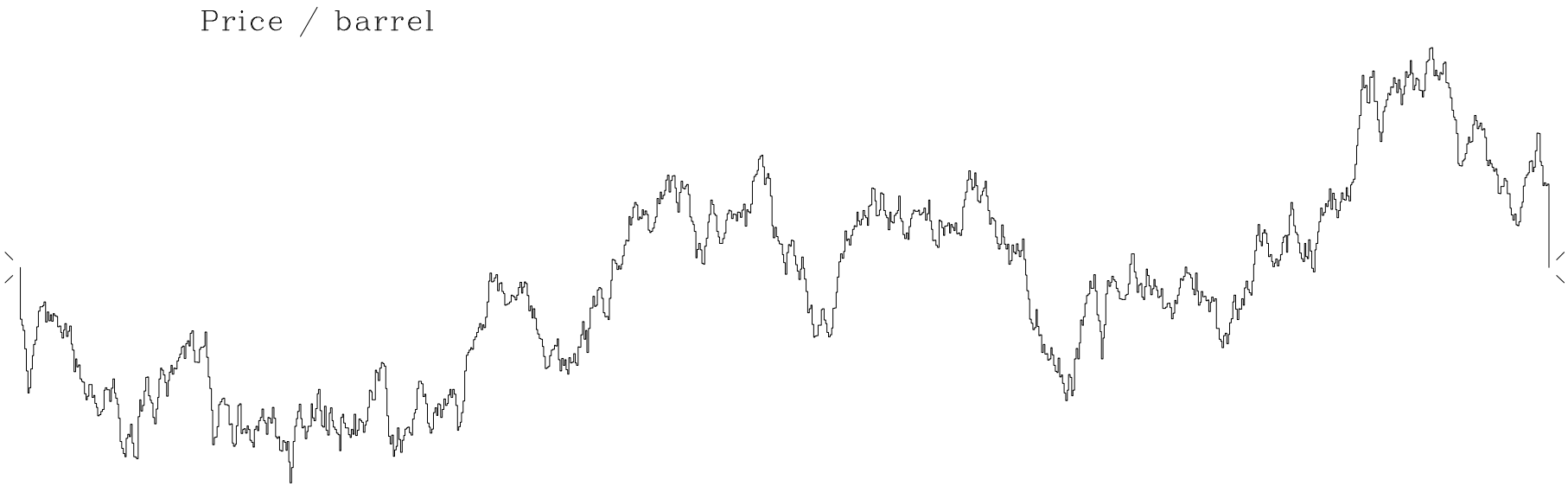


Figure 3.5: Random numbers into an integrator. zp-price [NR]

3.3.3. Symmetrical double integral

The **two-sided leaky integral** commonly arises as an even function, which is an ordinary leaky integral in one direction followed by another in the opposite direction. We will see also that the single leaky integral need not be causal; it could be an odd function.

The causal-integration operator flows one direction in time. Anticausal integration flows the other. Causal integration followed by anticausal integration makes a symmetrical smoothing operation, frequently used on the horizontal space axis. Since the idea of integration is generally associated with a long decay constant, and since data is generally limited in space, particular attention is usually given to the side boundaries. The simplest side boundaries are zero values, but these are generally rejected because people do not wish to assume data is zero beyond where it is measured. The most popular side conditions are not much more complicated, however. These are zero-slope side boundaries like those shown in Figure 3.6. I habitually smoothed with damped exponentials, but I switched to triangles after I encountered several examples where the exponential tails decreased too slowly.

The analysis for double-sided damped leaky integration with **zero-slope** boundaries is found in my previous books and elsewhere, so here I will simply state

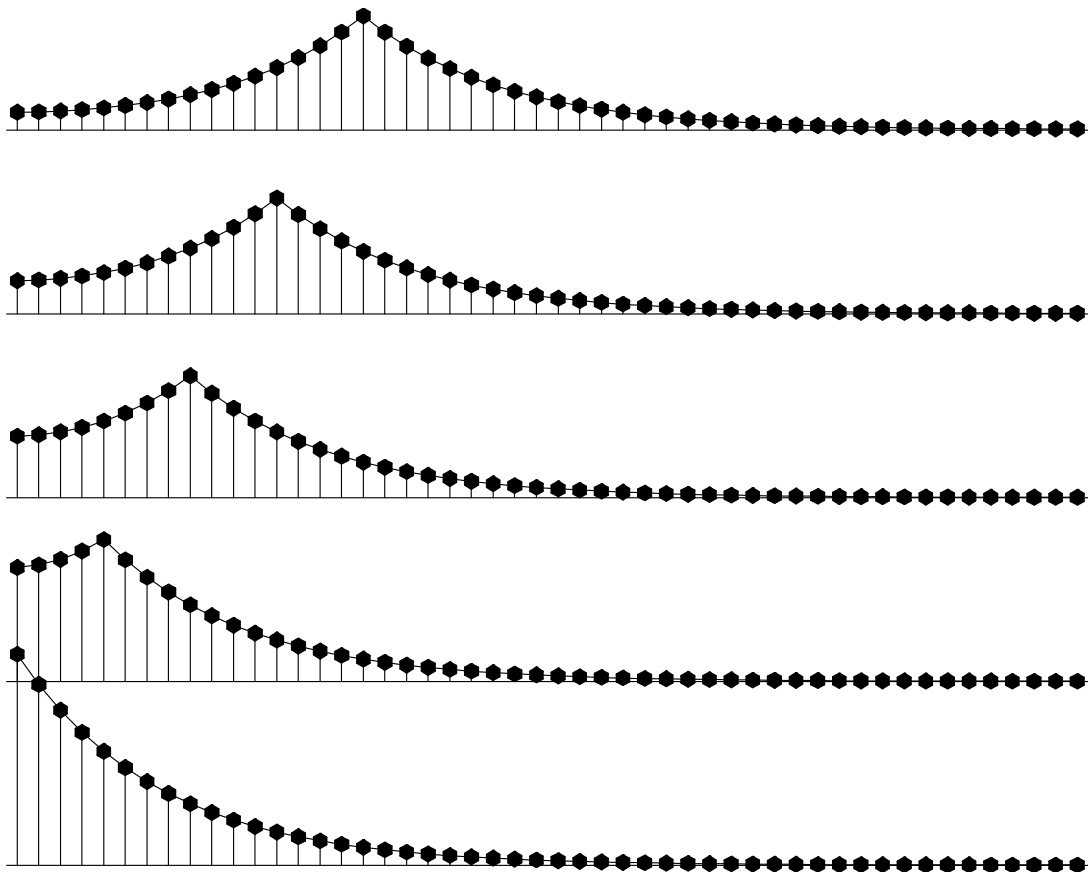


Figure 3.6: Pulses at various distances from a side boundary smoothed with two-sided leaky integration and zero-slope side conditions. Beyond the last value at the edge is a theoretical value that is the same as the edge value. zp-leakend [ER]

```
# keyword: tridiagonal smoothing on 1-axis or 2-axis
subroutine leaky( distance, m1, n12, uu, vv )
integer i,                    m1, n12
real    distance          # input:  1. < distance < infinity
real    uu(m1,n12)        # data in  is the vector (uu( 1, i), i=1,n12)
real    vv(m1,n12)        # data out is the vector (vv( 1, i), i=1,n12)
real a, b, dc, side
temporary real vecin( n12), vecout( n12)
a  = - (1.-1./distance);          b = 1.+a*a;     dc = b+a+a
a = a/dc;         b = b/dc;       side = a + b
do i= 1,n12 { vecin(i) = uu(1,i)}
if( distance<=1.|| n12==1) {call copy( n12, vecin, vecout)}
else                         {call tris( n12, side, a, b, a, side, vecin, vecout)}
do i= 1,n12  { vv(1,i) = vecout(i) }
return; end
```

Back

```
# tridiagonal simultaneous equations as in FGDP and IEI
#
subroutine tris( n, endl, a, b, c, endr, d, t )
integer i,       n
real endl, a, b, c, endr, d(n), t(n)
temporary real e(n), f(n), deni(n)
if( n == 1) {   t(1) = d(1) / b;        return }
e(1) = - a / endl
do i= 2, n-1 {
        deni(i) = 1. / ( b + c * e(i-1) )
        e(i) = - a * deni(i)
        }
f(1) = d(1) / endl
do i= 2, n-1
        f(i) = (d(i) - c * f(i-1)) * deni(i)
t(n) = ( d(n) - c * f(n-1) )  /  ( endr + c * e(n-1) )
do i= n-1, 1, -1
        t(i) = e(i) * t(i+1) + f(i)
return;  end
```

Back

the result and leave you with a working program. This kind of integration arises in the numerical solution of wave equations. Mathematically, it means solving $(\delta_{xx} - \alpha)V(x) = U(x)$ for $V(x)$ given $U(x)$. In the limit of small α, the operation is simply double integration. Nonzero α makes it **leaky integration**. The operation looks like the **Helmholtz equation** of physics but is not, because we take $\alpha > 0$ for damped solutions, whereas the Helmholtz equation typically takes $\alpha < 0$ for oscillating wave solutions. Figure 3.6 was created with `leaky()`, which performs the smoothing task using a double-sided exponential response with a decay to amplitude e^{-1} in a given `distance`. It invokes the routine `tris()`, a solver of tridiagonal simultaneous equations, which is explained in FGDP. leaky tris

It is convenient to refer to the symmetrical double *integration* operator as δ^{xx}, where the superscripts denote integration, in contrast to the usual subscripts, which denote **differentiation**. Since differentiation is widely regarded as an odd operator, it is natural also to define the odd integration operator $\delta^{x} = \delta^{xx}_{x}$.

3.3.4. Nonuniqueness of the integration operator

Integration can be thought of as $1/(-i\omega)$. The implied division by zero at $\omega = 0$ warns us that this filter is not quite normal. For example, $1/(-i\omega)$ appears to be an imaginary, antisymmetric function of ω. This implies that the time function is the real antisymmetric **signum** function, namely, $\mathrm{sgn}(t) = t/|t|$. The signum is not usually thought of as an integration operator, but by adding a constant we have a step function, and that is causal integration. By subtracting a constant we have anticausal integration. We can play games with the constant because it is at zero frequency that the definition contains zero division.

EXERCISES:

1 Show that the mean of the input of `leaky()` is the mean of the output, which demonstrates that the gain of the filter is unity at zero frequency.

3.4. DAMPED OSCILLATION

In polynomial multiplication, **zero**s of filters indicate frequencies where outputs will be small. Likewise, in polynomial division, zeros indicate frequencies where

outputs will be large.

3.4.1. Narrow-band filters

It seems we can represent a sinusoid by Z-transforms by setting a pole on the unit circle. Taking $Z_p = e^{i\omega_0}$, we have the filter

$$B(Z) \quad = \quad \frac{1}{1 - Z/Z_0} \quad = \quad \frac{1}{1 - Ze^{-i\omega_0}} \quad = \quad 1 + Ze^{-i\omega_0} + Z^2 e^{-i2\omega_0} + \cdots \tag{3.25}$$

The signal b_t seems to be the complex exponential $e^{-i\omega_0 t}$, but it is not quite that because b_t is "turned on" at $t = 0$, whereas $e^{-i\omega_0 t}$ is nonzero at negative time.

Now, how can we make a *real*-valued sinusoid starting at $t = 0$? Just as with zeros, we need to complement the pole at $+\omega_p$ by one at $-\omega_p$. The resulting signal b_t is shown on the left in Figure 3.7. On the right is a graphical attempt to plot the impulse function of dividing by zero at $\omega = \omega_0$.

Next, let us look at a damped case like leaky integration. Let $Z_p = e^{i\omega_0}/\rho$ and

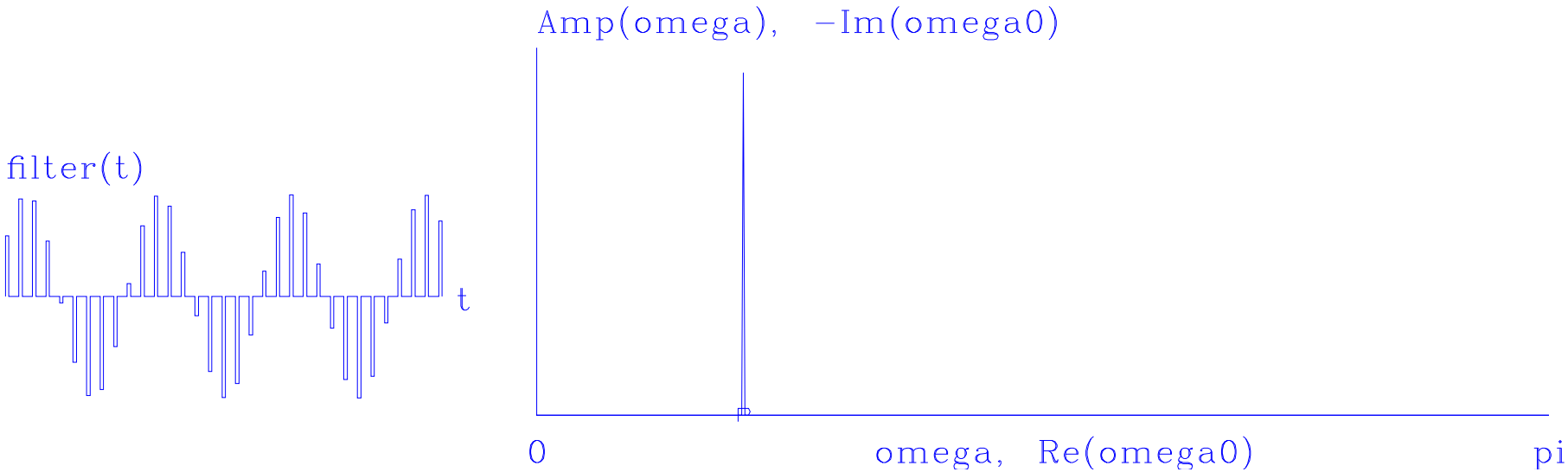


Figure 3.7: A pole on the real axis (and its mate at negative frequency) gives an impulse function at that frequency and a sinusoidal function in time. zp-sinus [NR]

$|\rho| < 1$. Then $1/Z_p = \rho e^{-i\omega_0}$. Define

$$B(Z) = \frac{1}{A(Z)} = \frac{1}{1 - Z/Z_p} = 1 + \frac{Z}{Z_p} + \left(\frac{Z}{Z_p}\right)^2 + \cdots \quad (3.26)$$

$$B(Z) = 1 + Z\rho e^{-i\omega_0} + Z^2\rho^2 e^{-i2\omega_0} + \cdots \quad (3.27)$$

The signal b_t is zero before $t = 0$ and is $\rho^t e^{-i\omega_0 t}$ after $t = 0$. It is a damped sinusoidal function with amplitude decreasing with time as ρ^t. We can readily recognize this as an exponential decay

$$\rho^t = e^{t \log \rho} \approx e^{-t(1-\rho)} \quad (3.28)$$

where the approximation is best for values of ρ near unity.

The wavelet b_t is complex. To have a real-valued time signal, we need another pole at the negative frequency, say $\overline{Z_p}$. So the composite denominator is

$$A(Z) = \left(1 - \frac{Z}{Z_p}\right)\left(1 - \frac{Z}{\overline{Z_p}}\right) = 1 - Z\rho 2\cos\omega_0 + \rho^2 Z^2 \quad (3.29)$$

Multiplying the two poles together as we did for roots results in the plots of $1/A(Z)$ in Figure 3.8. Notice the "p" in the figure. It indicates the location of the

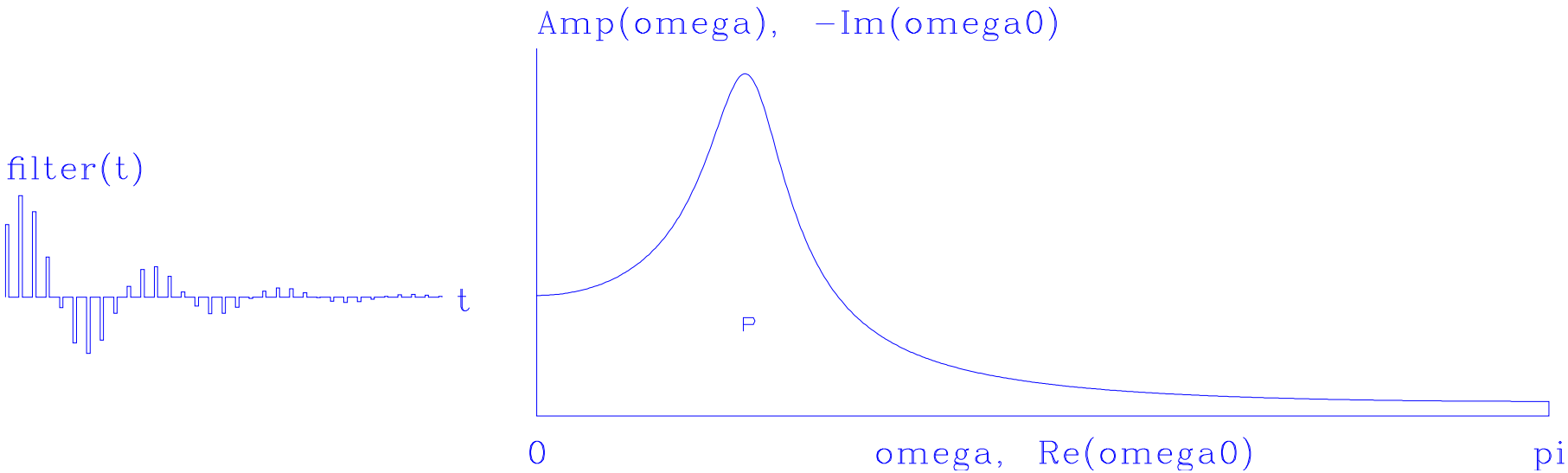


Figure 3.8: A damped sinusoidal function of time transforms to a pole near the real ω-axis, i.e., just outside the unit circle in the Z-plane. zp-dsinus [NR]

pole Z_p but is shown in the ω_0-plane, where $Z_p = e^{i\omega_0}$. Pushing the "p" left and right will lower and raise the resonant frequency. Pushing it down and up will raise and lower the duration of the resonance.

EXERCISES:

1 How far from the unit circle are the poles of $1/(1-.1Z+.9Z^2)$? What is the decay time of the filter and its resonant frequency?

3.4.2. Polynomial division

Convolution with the coefficients b_t of $B(Z) = 1/A(Z)$ is a narrow-banded filtering operation. If the pole is chosen very close to the unit circle, the filter bandpass becomes very narrow, and the coefficients of $B(Z)$ drop off very slowly. A method exists of narrow-band filtering that is much quicker than convolution with b_t. This is **polynomial division** by $A(Z)$. We have for the output $Y(Z)$:

$$Y(Z) \quad = \quad B(Z)\, X(Z) \quad = \quad \frac{X(Z)}{A(Z)} \tag{3.30}$$

Multiply both sides of (3.30) by $A(Z)$:

$$X(Z) \quad = \quad Y(Z)\, A(Z) \tag{3.31}$$

For definiteness, let us suppose that the x_t and y_t vanish before $t = 0$. Now identify coefficients of successive powers of Z to get

$$\begin{aligned} x_0 &= y_0 a_0 \\ x_1 &= y_1 a_0 + y_0 a_1 \\ x_2 &= y_2 a_0 + y_1 a_1 + y_0 a_2 \\ x_3 &= y_3 a_0 + y_2 a_1 + y_1 a_2 \\ x_4 &= y_4 a_0 + y_3 a_1 + y_2 a_2 \\ &= \cdots\cdots\cdots\cdots\cdots \end{aligned} \tag{3.32}$$

Let N_a be the highest power of Z in $A(Z)$. The k-th equation (where $k > N_a$) is

$$y_k a_0 + \sum_{i=1}^{N_a} y_{k-i} a_i \quad = \quad x_k \tag{3.33}$$

Solving for y_k, we get

$$y_k = \frac{x_k - \sum_{i=1}^{N_a} y_{k-i} a_i}{a_0} \tag{3.34}$$

Equation (3.34) may be used to solve for y_k once $y_{k-1}, y_{k-2}, \cdots$ are known. Thus the solution is **recursive**. The value of N_a is only 2, whereas N_b is technically infinite and would in practice need to be approximated by a large value. So the **feedback** operation (3.34) is much quicker than convolving with the filter $B(Z) = 1/A(Z)$. A program for the task is given below. Data lengths such as `na` in the program `polydiv()` include coefficients of all N_a powers of Z as well as $1 = Z^0$, so `na` = $N_a + 1$. polydiv

3.4.3. Spectrum of a pole

Now that we have seen the single-pole filter and the pole-pair filter in both the time domain and the frequency domain, let us find their analytical expressions. Taking

```
#       polynomial division feedback filter:    Y(Z) = X(Z) / A(Z)
#
subroutine polydiv( na, aa, nx, xx, ny, yy )
integer na      # number of coefficients of denominator
integer nx      # length of the input function
integer ny      # length of the output function
real    aa(na)  # denominator recursive filter
real    xx(nx)  # input trace
real    yy(ny)  # output trace, as long as input trace.

        integer ia, iy
        do iy= 1, ny
                if( iy <= nx)
                        yy(iy) = xx(iy)
                else
                        yy(iy) = 0.
        do iy= 1, na-1  {                                       # lead-in terms
                do ia= 2, iy
                        yy(iy) = yy(iy) - aa(ia) * yy(iy-ia+1)
                yy(iy) = yy(iy) / aa(1)
                }
        do iy= na, ny {                                         # steady state
                do ia= 2, na
                        yy(iy) = yy(iy) - aa(ia) * yy(iy-ia+1)
                yy(iy) = yy(iy) / aa(1)
                }
        return; end
```

Back

the pole to be $Z_p = e^{i\omega_0}/\rho$, we have

$$A(Z) \quad = \quad 1 - \frac{Z}{Z_p} \quad = \quad 1 - \frac{\rho}{e^{i\omega_0}}\, e^{i\omega} \quad = \quad 1 - \rho e^{i(\omega-\omega_0)} \tag{3.35}$$

The complex conjugate is

$$\overline{A}\left(\frac{1}{Z}\right) \quad = \quad 1 - \rho e^{-i(\omega-\omega_0)} \tag{3.36}$$

The spectrum of a pole filter is the inverse of

$$\begin{aligned}
\overline{A}\left(\frac{1}{Z}\right) A(Z) &= (1 - \rho e^{-i(\omega-\omega_0)})\,(1 - \rho e^{i(\omega-\omega_0)}) \\
&= 1 + \rho^2 - \rho(e^{-i(\omega-\omega_0)} + e^{i(\omega-\omega_0)}) \\
&= 1 + \rho^2 - 2\rho\cos(\omega-\omega_0) \\
&= 1 + \rho^2 - 2\rho + 2\rho[1 - \cos(\omega-\omega_0)] \\
&= (1-\rho)^2 + 4\rho\sin^2\frac{\omega-\omega_0}{2}
\end{aligned} \tag{3.37}$$

With the definition of a small $\epsilon = 1 - \rho > 0$, inverting gives

$$\overline{B}\left(\frac{1}{Z}\right) B(Z) \;\approx\; \frac{1}{\epsilon^2 + 4\sin^2\frac{\omega-\omega_0}{2}} \tag{3.38}$$

Specializing to frequencies close to ω_0, where the denominator is small and the function is large, gives

$$\overline{B}\left(\frac{1}{Z}\right) B(Z) \;\approx\; \frac{1}{\epsilon^2 + (\omega-\omega_0)^2} \tag{3.39}$$

This is called a "**narrow-band filter**" because in the Fourier domain the function is large only in a narrow band of frequencies. Setting $\overline{B}B$ to half its peak value of $1/\epsilon^2$, we find a half-bandwidth of $\Delta\omega/2 = |\omega - \omega_0| = \epsilon$. The damping time constant Δt of the damped sinusoid b_t is shown in the exercises following this section to be $\Delta t = 1/\epsilon$.

Naturally we want a real-time function, so we multiply the filter $1/(1 - Z/Z_p)$ times $1/(1 - Z/\overline{Z}_p)$. The resulting time function is real because conjugate poles are like the conjugate roots. The spectrum of the conjugate factor $1/(1 - Z/\overline{Z}_p)$ is like (3.38), except that ω_0 is replaced by $-\omega_0$. Multiplying the response (3.38) by itself with $-\omega_0$ yields the symmetric function of ω displayed on the right in Figure 3.9.

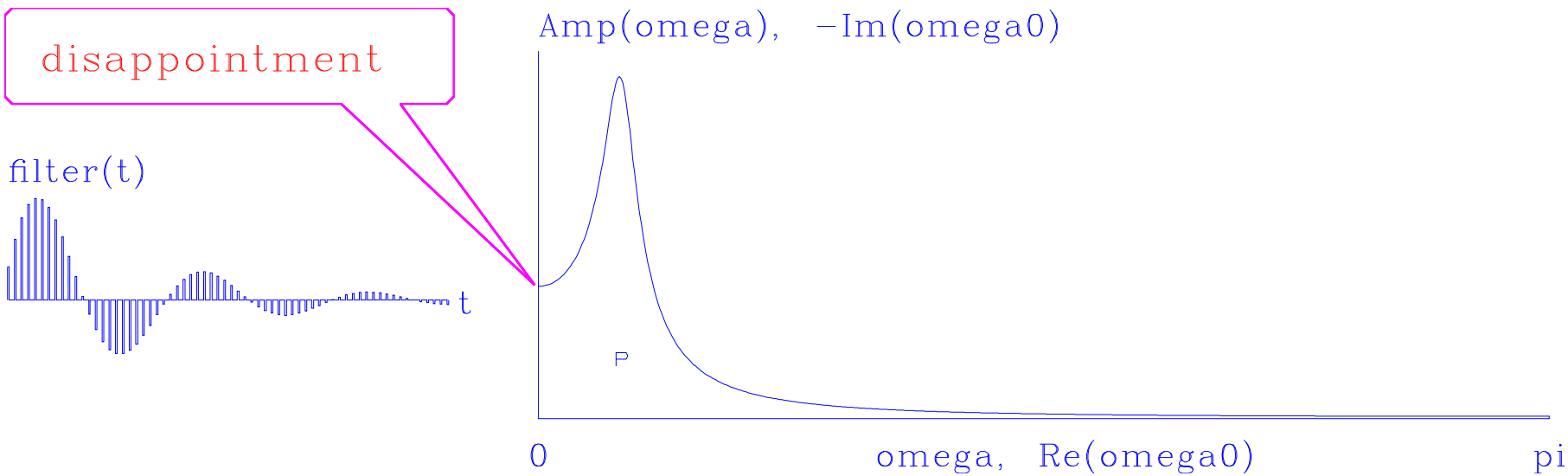


Figure 3.9: A pole near the real axis gives a damped sinusoid in time on the left. On the right is $1/|A(\omega)|$ for ω real. zp-disappoint [NR]

You might be disappointed if you intend to apply the filter of Figure 3.9 as a narrow-band filter. Notice that the passband is asymmetric and that it passes the zero frequency. Equation (3.38) is symmetric about ω_0, but taking the product with its image about $-\omega_0$ has spoiled the symmetry. Should we be concerned about this "edge effect"? The answer is yes, whenever we handle real data. For real data, Δt is usually small enough. Recall that $\omega_{\text{radians/sample}} = \omega_{\text{radians/sec}} \Delta t$. Consider a pole at a particular $\omega_{\text{radians/sec}}$: decreasing Δt pushes $\omega_{\text{radians/sample}}$ towards zero, which is where a pole and its mate at negative frequency create the asymmetrical response shown in Figure 3.9.

So in practice we might like to add a zero at zero frequency and at the Nyquist frequency, i.e., $(1 - Z)(1 + Z)$, as shown in Figure 3.10. Compare Figure 3.10 and 3.9. If the time functions were interchanged, could you tell the difference between the figures? There are two ways to distinguish them. The most obvious is that the zero-frequency component is made evident in the time domain by the sum of the filter coefficients (theoretically, $F(Z = 1)$). A more subtle clue is that the first half-cycle of the wave in Figure 3.10 is shorter than in Figure 3.9; hence, it contains extra high frequency energy, which we can see in the spectrum.

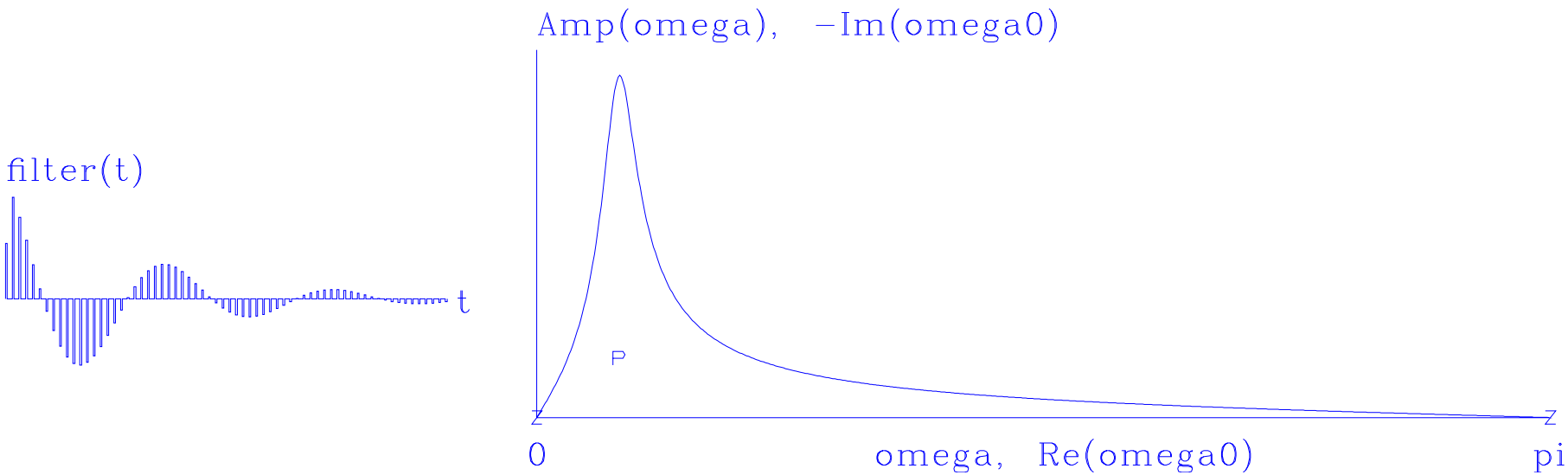


Figure 3.10: Poles at $\pm\omega_0$; a root at $\omega = 0$ and another root at $\omega = \pi$. zp-symdsin [NR]

EXERCISES:

1 Sketch the function in equation (3.38) over the range $-\pi <= \omega <= \pi$, taking care to distinguish it from Figure 3.9.

2 Figure 3.9 shows a bump around ω_0 that does not look symmetric because it is the *product* of equation (3.38) with a frequency-reversed copy. Consider the *sum* $[1/(1-Z/Z_p)]+[1/(1-Z/\overline{Z}_p)]$. Is the time filter real? Where are its poles and zeros? How will its amplitude as a function of frequency compare with the amplitude of Figure 3.9? Will the bump look more symmetric?

3.4.4. Rational filters

A general model for filtering includes both convolution (numerator Z-transforms) and feedback filtering (denominator Z-transforms):

$$Y(Z) \quad = \quad \frac{B(Z)}{A(Z)} X(Z) \tag{3.40}$$

There are a variety of ways to implement equation (3.40) in a computer. We could do the polynomial division $X(Z)/A(Z)$ first and then multiply (convolve) with $B(Z)$,

or we could do the multiplication first and the division later. Alternately, we could do them simultaneously if we identified coefficients of $A(Z)Y(Z) = B(Z)X(Z)$ and solved for recursive equations, as we did for (3.34).

The **rational filter** is more powerful than either a purely numerator filter or a purely denominator filter because, like its numerator part, the rational filter can easily destroy any frequency totally, and, like its denominator part, it can easily enhance any frequency without limit. Finite-difference solutions of differential equations often appear as rational filters.

EXERCISES:

1 Consider equation (3.40). What time-domain recurrence (analogous to equation (3.34)) is implied?

3.5. INSTABILITY

Consider the example $B(Z) = 1 - Z/2$. The inverse

$$A(Z) \quad = \quad \frac{1}{1-\frac{Z}{2}} \quad = \quad 1 + \frac{Z}{2} + \frac{Z^2}{4} + \frac{Z^3}{8} + \cdots \tag{3.41}$$

can be found by a variety of familiar techniques, such as (1) polynomial division, (2) Taylor's power-series formula, or (3) the binomial theorem. In equation (3.41) we see that there are an infinite number of filter coefficients, but because they drop off rapidly, approximation in a computer presents no difficulty.

We are not so lucky with the filter $B(Z) = 1 - 2Z$. Here we have

$$A(Z) \quad = \quad \frac{1}{1-2Z} \quad = \quad 1 + 2Z + 4Z^2 + 8Z^3 + 16Z^4 + 32Z^5 + \cdots \tag{3.42}$$

The coefficients of this series increase without bound. This is called "**instability**." The outputs of the filter $A(Z)$ depend infinitely on inputs of the infinitely distant past. (Recall that the present output of $A(Z)$ is a_0 times the present input x_1, plus a_1 times the previous input x_{t-1}, etc., so a_n represents memory of n time units earlier.) This example shows that some filters $B(Z)$ will not have useful inverses

$A(Z)$ determined by polynomial division. Two sample plots of divergence are given in Figure 3.11.

For the filter $1 - Z/Z_0$ with a single zero, the inverse filter has a single pole at the same location. We have seen a stable inverse filter when the pole $|Z_p| > 1$ exceeds unity and **instability** when the pole $|Z_p| < 1$ is less than unity. Occasionally we see **complex-valued signal**s. Stability for wavelets with complex coefficients is as follows: if the solution value Z_0 of $B(Z_0) = 0$ lies inside the **unit circle** in the complex plane, then $1/B(Z)$ will have coefficients that blow up; and if the root lies outside the unit circle, then the inverse $1/B(Z)$ will be bounded.

3.5.1. Anticausality

Luckily, unstable filters can be made stable as follows:

$$\frac{1}{1-2Z} \quad = \quad -\frac{1}{2Z}\,\frac{1}{1-\frac{1}{2Z}} \quad = \quad -\frac{1}{2Z}\left(1+\frac{1}{2Z}+\frac{1}{(2Z)^2}+\cdots\right) \tag{3.43}$$

Equation (3.43) is a series expansion in $1/Z$—in other words, a series about infinity. It converges from $|Z| = \infty$ all the way in to a circle of radius $|Z| = 1/2$. This means

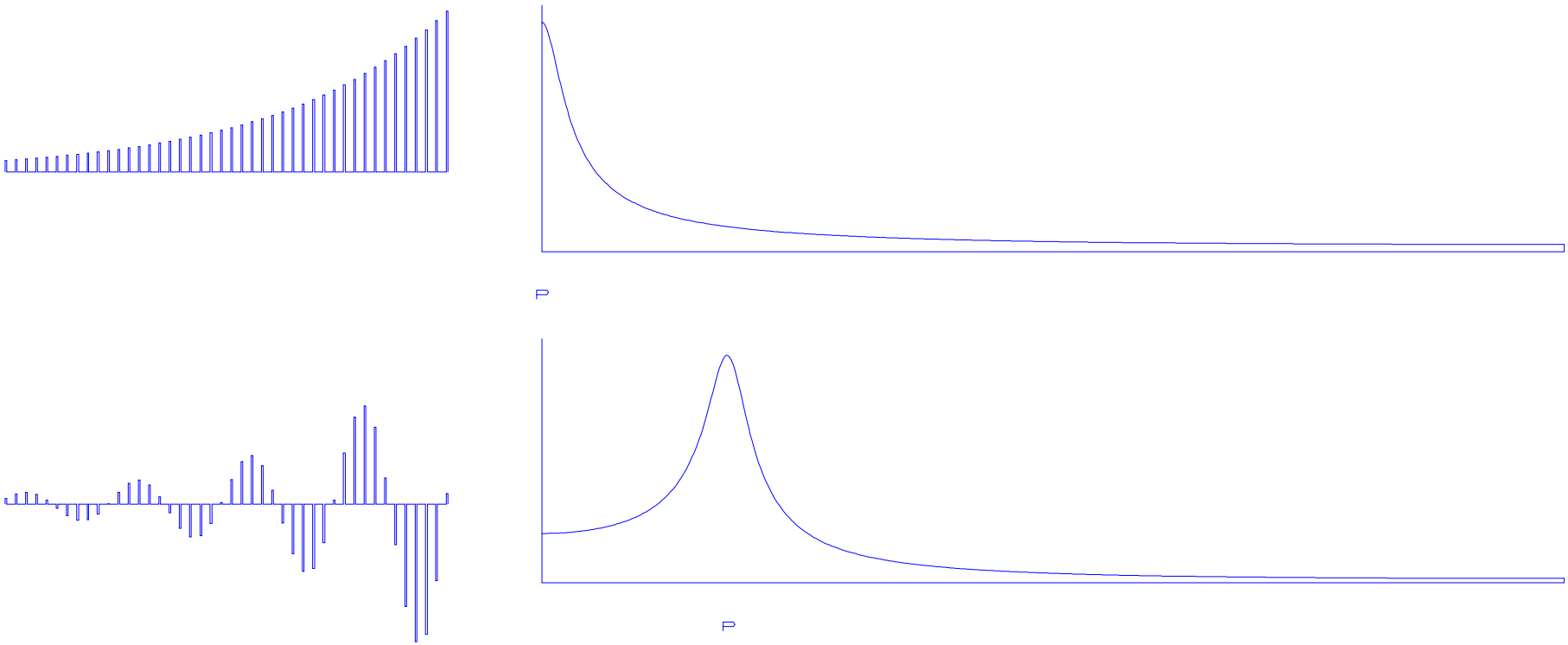


Figure 3.11: Top: the growing time function of a pole inside the unit circle at zero frequency. Bottom: at a nonzero frequency. Where the time axis is truncated, the signals are growing, and they will increase indefinitely. zp-diverge [NR]

that the inverse converges on the unit circle where it must, if the coefficients are to be bounded. In terms of filters, it means that the inverse filter must be one of those filters that responds to future inputs. Hence, although it is not physically realizable, it may be used in computer simulation.

Examining equations (3.42) and (3.43), we see that the filter $1/(1-2Z)$ can be expanded into powers of Z in (at least) two different ways. Which one is correct? The theory of complex variables shows that, given a particular numerical value of Z, only one of the sums (3.42) or (3.43) will be finite. We must use the finite one, and since we are interested in Fourier series, we want the numerical value $|Z| = 1$ for which the first series diverges and the second converges. Thus the only acceptable filter is *anticausal*.

The spectra plotted in Figure 3.11 apply to the anticausal expansion. Obviously the causal expansion, which is unbounded, has an infinite spectrum.

We saw that a polynomial $B(Z)$ of degree N may be factored into N subsystems, and that the ordering of subsystems is unimportant. Suppose we have factored $B(Z)$ and found that some of its roots lie outside the unit circle and some lie inside. We first invert the outside roots with equation (3.41) and then invert the inside roots with equation (3.43). If there are any roots exactly on the unit circle, then we have

a special case in which we can try either inverse, but neither may give a satisfactory result in practice. Implied zero division is nature's way of telling us that what we are trying to do cannot be done that way (if at all).

3.5.2. Inverse filters

Let b_t denote a filter. Then a_t is its "**inverse filter**" if the convolution of a_t with b_t is an impulse function. Filters are said to be inverse to one another if their Fourier transforms are inverse to one another. So in terms of Z-transforms, the filter $A(Z)$ is said to be inverse to the signal of $B(Z)$ if $A(Z)B(Z) = 1$. What we have seen so far is that the inverse filter can be stable or unstable depending on the location of its poles. Likewise, if $B(Z)$ is a filter, then $A(Z)$ is a usable filter inverse to $B(Z)$, if $A(Z)B(Z) = 1$ and if $A(Z)$ does not have coefficients that tend to infinity.

Another approach to inverse filters lies in the Fourier domain. There a filter inverse to b_t is the a_t made by taking the inverse Fourier transform of $1/B(Z(\omega))$. If $B(Z)$ has its zeros outside the unit circle, then a_t will be causal; otherwise not. In the Fourier domain the only danger is dividing by a zero, which would be a pole on the unit circle. In the case of Z-transforms, zeros should not only be off the

circle but also outside it. So the ω-domain seems safer than the Z-domain. Why not always use the Fourier domain? The reasons we do not always inverse filter in the ω-domain, along with many illustrations, are given in chapter 7.

3.5.3. The unit circle

What is the meaning of a pole? We will see that the location of poles determines whether filters are stable (have finite output) or unstable (have unbounded output). Considering both positive and negative values of ρ, we find that stability is associated with $|\rho| < 1$. The pole $|\rho| < 1$ happens to be real, but we will soon see that poles are complex more often than not. In the case of complex poles, the condition of stability is that they all should satisfy $|Z_p| > 1$. In the complex Z-plane, this means that all the poles should be outside a circle of unit radius, the so-called **unit circle**.

3.5.4. The mapping between Z and complex frequency

We are familiar with the fact that *real* values of ω correspond to complex values of $Z = e^{i\omega}$. Now let us look at *complex* values of ω:

$$Z \quad = \quad \Re Z + i\Im Z \quad = \quad e^{i(\Re\omega + i\Im\omega)} \quad = \quad e^{-\Im\omega}\, e^{i\Re\omega} \quad = \quad \text{amplitude}\; e^{i\,\text{phase}} \tag{3.44}$$

Thus, when $\Im\omega > 0$, $|Z| < 1$. In words, we transform the upper half of the ω-plane to the interior of the unit circle in the Z-plane. Likewise, the stable region for poles is the lower half of the ω-plane, which is the exterior of the unit circle. Figure 3.12 shows the transformation. Some engineering books choose a different sign convention ($Z = e^{-i\omega}$), but I selected the sign convention of physics.

3.5.5. The meaning of divergence

To prove that one equals zero, take an infinite series such as $1, -1, +1, -1, +1, \ldots$, group the terms in two different ways, and add them as follows:

$$\begin{array}{ccccccccccccc}
(1-1) & + & (1-1) & + & (1-1) & + \cdots & = & 1 & + & (-1+1) & + & (-1+1) & + \cdots \\
0 & + & 0 & + & 0 & + \cdots & = & 1 & + & 0 & + & 0 & + \cdots
\end{array}$$

Figure 3.12: Left is the complex ω-plane with axes $(x, y) = (\Re\omega_0, \Im\omega_0)$. Right is the Z-plane with axes $(x, y) = (\Re Z_0, \Im Z_0)$. The words "Convergent" and "Divergent" are transformed by $Z = e^{i\omega}$. zp-Z [ER]

$$0 \quad = \quad 1$$

Of course this does not prove that one equals zero: it proves that care must be taken with infinite series. Next, take another infinite series in which the terms may be regrouped into any order without fear of paradoxical results. For example, let a pie be divided into halves. Let one of the halves be divided in two, giving two quarters. Then let one of the two quarters be divided into two eighths. Continue likewise. The infinite series is 1/2, 1/4, 1/8, 1/16, ... No matter how the pieces are rearranged, they should all fit back into the pie plate and exactly fill it.

The danger of infinite series is not that they have an infinite number of terms but that they may sum to infinity. Safety is assured if the sum of the absolute values of the terms is finite. Such a series is called “absolutely convergent."

3.5.6. Boundedness

Given different numerical values for Z, we can ask whether $X(Z)$ is finite or infinite. Numerical values of Z of particular interest are $Z = +1$, $Z = -1$, and all those complex values of Z which are unit magnitude, say $|Z| = 1$ or $Z = e^{i\omega}$, where ω is the real Fourier transform variable. When ω is the variable, the Z-transform is a

Fourier sum.

We can restrict our attention to those signals u_t that have a finite amount of energy by demanding that $U(Z)$ be finite for all values of Z on the unit circle $|Z| = 1$. Filter functions always have finite energy.

3.5.7. Causality and the unit circle

The most straightforward way to say that a filter is **causal** is to say that its time-domain coefficients vanish before zero lag, that is, $u_t = 0$ for $t < 0$. Another way to say this is $U(Z)$ is finite for $Z = 0$. At $Z = 0$, the Z-transform would be infinite if the coefficients u_{-1}, u_{-2}, etc., were not zero.

For a causal function, each term in $U(Z)$ will be smaller if Z is taken to be inside the circle $|Z| < 1$ rather than on the rim $|Z| = 1$. Thus, convergence at $Z = 0$ and on the circle $|Z| = 1$ implies convergence everywhere inside the unit circle. So boundedness combined with causality means convergence in the unit circle.

Convergence at $Z = 0$ but not on the circle $|Z| = 1$ would refer to a causal function with infinite energy, a case of no practical interest. What function converges on the circle, at $Z = \infty$, but not at $Z = 0$? What function converges at all

three places, $Z = 0$, $Z = \infty$, and $|Z| = 1$?

3.6. MINIMUM-PHASE FILTERS

Let b_t denote a filter. Then a_t is its inverse filter if the convolution of a_t with b_t is an impulse function. In terms of Z-transforms, an inverse is simply defined by $A(Z) = 1/B(Z)$. Whether the filter $A(Z)$ is causal depends on whether it is finite everywhere inside the unit circle, or really on whether $B(Z)$ vanishes *anywhere* inside the circle. For example, $B(Z) = 1 - 2Z$ vanishes at $Z = 1/2$. There $A(Z) = 1/B(Z)$ must be infinite, that is to say, the series $A(Z)$ must be nonconvergent at $Z = 1/2$. Thus, as we have just seen, a_t is noncausal. A most interesting case, called "**minimum phase**," occurs when both a filter $B(Z)$ and its inverse are causal. In summary,

causal:	$\|B(Z)\| < \infty$	for $\|Z\| \leq 1$
causal inverse:	$\|1/B(Z)\| < \infty$	for $\|Z\| \leq 1$
minimum phase:	both above conditions	

The reason the interesting words "minimum phase" are used is given in chapter 10.

3.6.1. Mechanical interpretation

Because of the stringent conditions on minimum-phase wavelets, you might wonder whether they can exist in nature. A simple mechanical example should convince you that minimum-phase wavelets are plentiful: denote the stress (pressure) in a material by x_t, and denote the strain (volume change) by y_t. Physically, we can specify either the stress or the strain, and nature gives us the other. So obviously the stress in a material may be expressed as a linear combination of present and past strains. Likewise, the strain may be deduced from present and past stresses. Mathematically, this means that the filter that relates stress to strain and vice versa has all poles and zeros outside the unit circle. Of the minimum-phase filters that model the physical world, many conserve energy too. Such filters are called "**impedance**s" and are described further in FGDP and IEI, especially IEI.

3.6.2. Laurent expansion

Given an unknown filter $B(Z)$, to understand its inverse, we need to factor $B(Z)$ into two parts: $B(Z) = B_{out}(Z)B_{in}(Z)$, where B_{out} contains all the roots outside the unit circle and B_{in} contains all the roots inside. Then the inverse of B_{out} is

expressed as a Taylor series about the origin, and the inverse of B_{in} is expressed as a Taylor series about infinity. The final expression for $1/B(Z)$ is called a "**Laurent expansion**" for $1/B(Z)$, and it converges on a ring including the unit circle. Cases with zeros exactly on the unit circle present special problems. For example, the differentiation filter $(1-Z)$ is the inverse of integration, but the converse is not true, because of the additive constant of integration.

EXERCISES:

1 Find the filter that is inverse to $(2-5Z+2Z^2)$. You may just drop higher-order powers of Z, but an exact expression for the coefficients of any power of Z is preferable. (Partial fractions is a useful, though not a necessary, technique.) Sketch the impulse response.

2 Describe a general method for determining $A(Z)$ and $B(Z)$ from a Taylor series of $B(Z)/A(Z) = C_0 + C_1 Z + C_2 Z^2 + \cdots + C_\infty Z^\infty$, where $B(Z)$ and $A(Z)$ are polynomials of unknown degree n and m, respectively. Work out the case $C(Z) = \frac{1}{2} - \frac{3}{4}Z - \frac{3}{8}Z^2 - \frac{3}{16}Z^3 - \frac{3}{32}Z^4 - \cdots$. Do not try this problem unless you are familiar with determinants. (HINT: identify coefficients of $B(Z) = A(Z)C(Z)$.)

3.7. INTRODUCTION TO ALL-PASS FILTERS

An "**all-pass filter**" is a filter whose spectral magnitude is unity. Given an input $X(Z)$ and an output $Y(Z)$, we know that the spectra of the two are the same, i.e., $X(1/Z)X(Z) = Y(1/Z)Y(Z)$. The existence of an infinitude of all-pass filters tells us that an infinitude of wavelets can have the same spectrum. Wave propagation without absorption is modeled by all-pass filters. All-pass filters yield a waveform distortion that can be corrected by methods discussed in chapter 10.

The simplest example of an all-pass filter is the delay operator $Z = e^{i\omega}$ itself. Its phase as a function of ω is simply ω.

A less trivial example of phase distortion can be constructed from a single root Z_r, where Z_r is an arbitrary complex number. The ratio of any complex number to its complex conjugate, say $(x+iy)/(x-iy)$, is of unit magnitude, because, taking $x+iy = \rho e^{i\phi}$ and $x-iy = \rho e^{-i\phi}$, the ratio is $|e^{i2\phi}|$. Thus, given a minimum-phase filter $B(\omega)$, we can take its conjugate and make an all-pass filter $P(Z)$ from the ratio $P(Z) = \overline{B(\omega)}/B(\omega)$. A simple case is

$$B(\omega) \quad = \quad 1 - \frac{Z}{Z_r} \tag{3.45}$$

$$\overline{B(\omega)} \quad = \quad 1 - \frac{1}{Z\overline{Z_r}} \tag{3.46}$$

The all-pass filter $\overline{B}/B$ is not causal because of the presence of $1/Z$ in $\overline{B}$. We can repair that by multiplying by another all-pass operator, namely, Z. The resulting causal all-pass filter is

$$P(Z) \quad = \quad \frac{Z\overline{B}(1/Z)}{B(Z)} \quad = \quad \frac{Z - \frac{1}{\overline{Z}_r}}{1 - \frac{Z}{Z_r}} \tag{3.47}$$

Equation (3.47) can be raised to higher powers to achieve a stronger frequency-dispersion effect. Examples of time-domain responses of various all-pass filters are shown in Figure 3.13.

The denominator of equation (3.47) tells us that we have a pole at Z_r. Let this location be $Z_r = e^{i\omega_0}/\rho$. The numerator vanishes at

$$Z \quad = \quad Z_0 \quad = \quad \frac{1}{\overline{Z}_r} \quad = \quad \rho\, e^{i\omega_0} \tag{3.48}$$

In conclusion, the pole is outside the unit circle, and the zero is inside. They face one another across the circle at the phase angle ω_0.

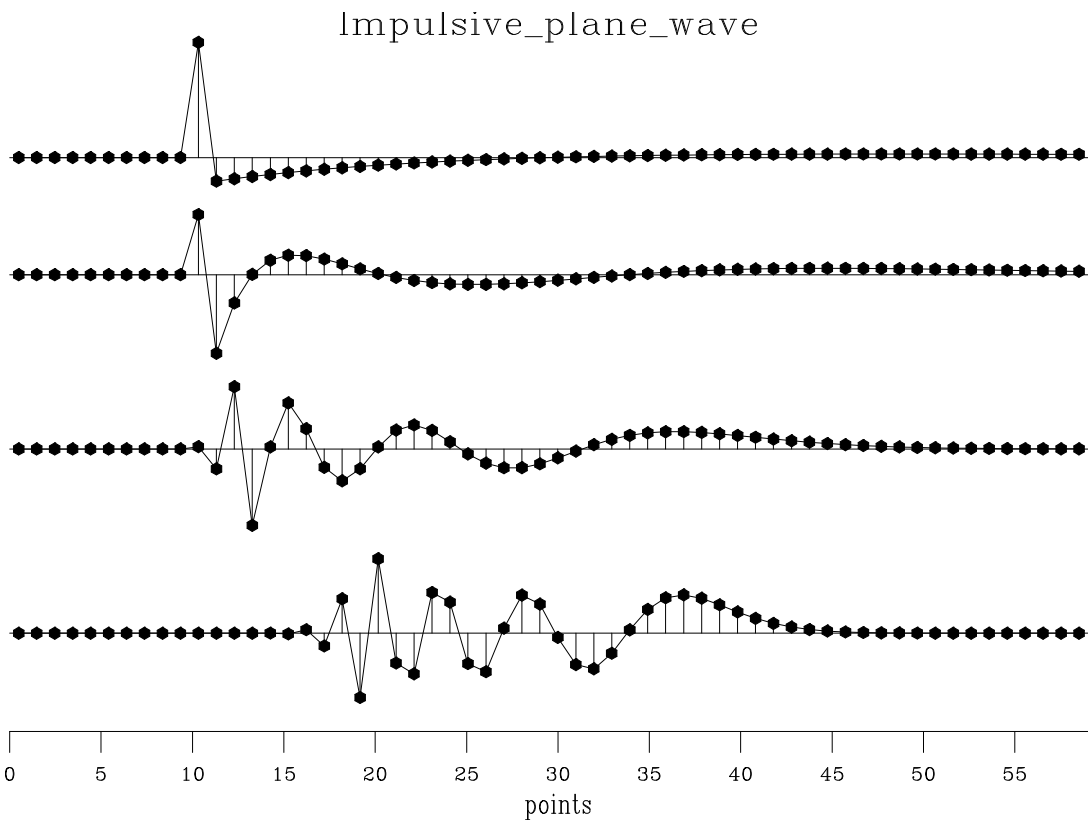


Figure 3.13: Examples of causal all-pass filters with real poles and zeros. These have high frequencies at the beginning and low frequencies at the end. zp-disper [ER]

The all-pass filter (3.47) outputs a **complex-valued signal**, however. To see real outputs, we must handle the negative frequencies in the same way as the positive ones. The filter (3.47) should be multiplied by another like itself but with ω_0 replaced by $-\omega_0$; i.e., with Z_r replaced by $\overline{Z_r}$. The result of this procedure is shown in Figure 3.14.

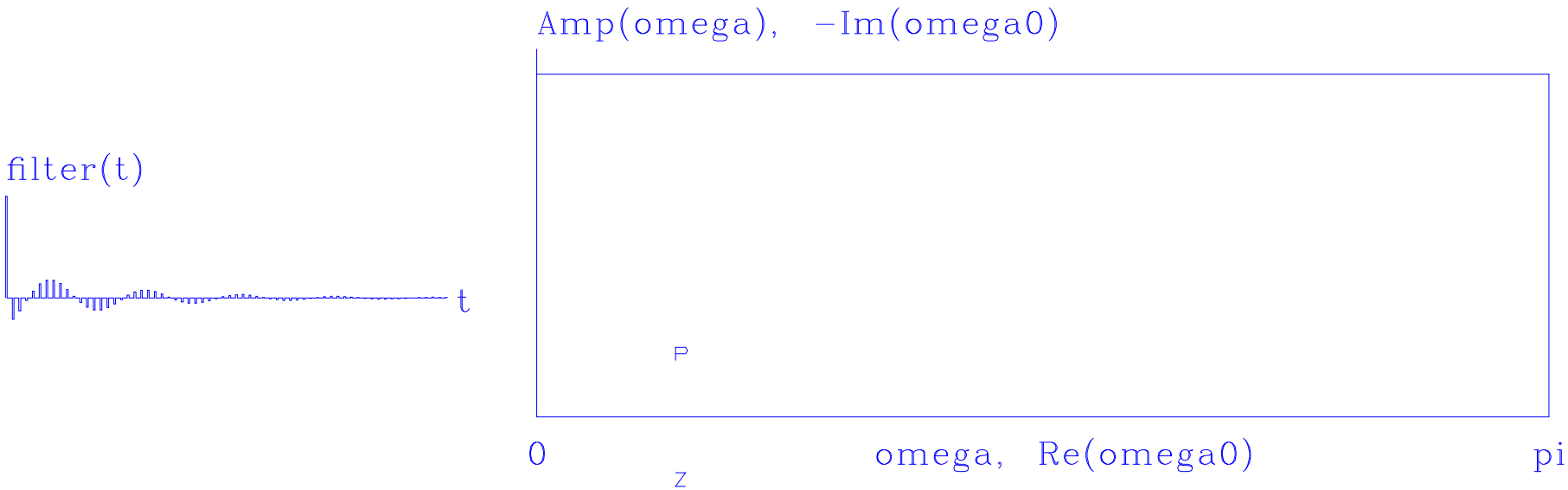


Figure 3.14: All-pass filter with a complex pole-zero pair. The pole and zero are at equal logarithmic distances from the unit circle. zp-allpass [NR]

A general form for an all-pass filter is $P(Z) = Z^N \overline{A}(1/Z)/A(Z)$, where $A(Z)$ is an arbitrary minimum-phase filter. That this form is valid can be verified by checking that $\overline{P}(1/Z)P(Z) = 1$.

EXERCISES:

1 Verify that $\overline{P}(1/Z)P(Z) = 1$ for the general form of an all-pass filter $P(Z) = Z^N \overline{A}(1/Z)/A(Z)$.

2 Given an all-pass filter

$$P(Z) \quad = \quad \frac{d + eZ + fZ^2}{1 + bZ + cZ^2}$$

with poles at $Z_p = 2$ and $Z_p = 3$, what are b, c, d, e, and f?

3.7.1. Notch filter

A "**notch filter**" rejects a narrow frequency band and leaves the rest of the spectrum little changed. The most common example is 60-Hz noise from power lines. Another is low-frequency ground roll. Such filters can easily be made using a slight

variation on the all-pass filter. In the all-pass filter, the pole and zero have equal (logarithmic) relative distances from the unit circle. All we need to do is put the zero closer to the circle. Indeed, there is no reason why we should not put the zero right on the circle: then the frequency at which the zero is located is exactly canceled from the spectrum of input data. Narrow-band filters and sharp cutoff filters should be used with caution. An ever-present penalty for using such filters is that they do not decay rapidly in time. Although this may not present problems in some applications, it will certainly do so in others. Obviously, if the data-collection duration is shorter than or comparable to the impulse response of the narrow-band filter, then the transient effects of starting up the experiment will not have time to die out. Likewise, the notch should not be too narrow in a 60-Hz rejection filter. Even a bandpass filter (an example of which, a Butterworth filter, is implemented in chapter 10) has a certain decay rate in the time domain which may be too slow for some experiments. In radar and in reflection seismology, the importance of a signal is not related to its strength. Late arriving echoes may be very weak, but they contain information not found in earlier echoes. If too sharp a frequency characteristic is used, then filter resonance from early strong arrivals may not have decayed enough by the time the weak late echoes arrive.

A curious thing about narrow-band reject filters is that when we look at their impulse responses, we always see the frequency being rejected! For example, look at Figure 3.15. The filter consists of a large spike (which contains all frequencies) and then a sinusoidal tail of polarity opposite to that of the frequency being rejected.

The vertical axis in the complex frequency plane in Figure 3.15 is not exactly $\Im\omega_0$. Instead it is something like the logarithm of $\Im\omega_0$. The logarithm is not precisely appropriate either because zeros may be exactly on the unit circle. I could not devise an ideal theory for scaling $\Im\omega_0$, so after some experimentation, I chose $\Im\omega_0 = -(1+y^2)/(1-y^2)$, where y is the vertical position in a window of vertical range $0 < y < 1$. Because of the minus sign, the outside of the unit circle is above the $\Re\omega_0$ axis, and the inside of the unit circle is below it.

EXERCISES:

1 Find a three-term real feedback filter to reject 59-61 Hz on data that is sampled at 500 points/s. (Try for about 50% rejection at 59 and 61.) Where are the poles? What is the decay time of the filter?

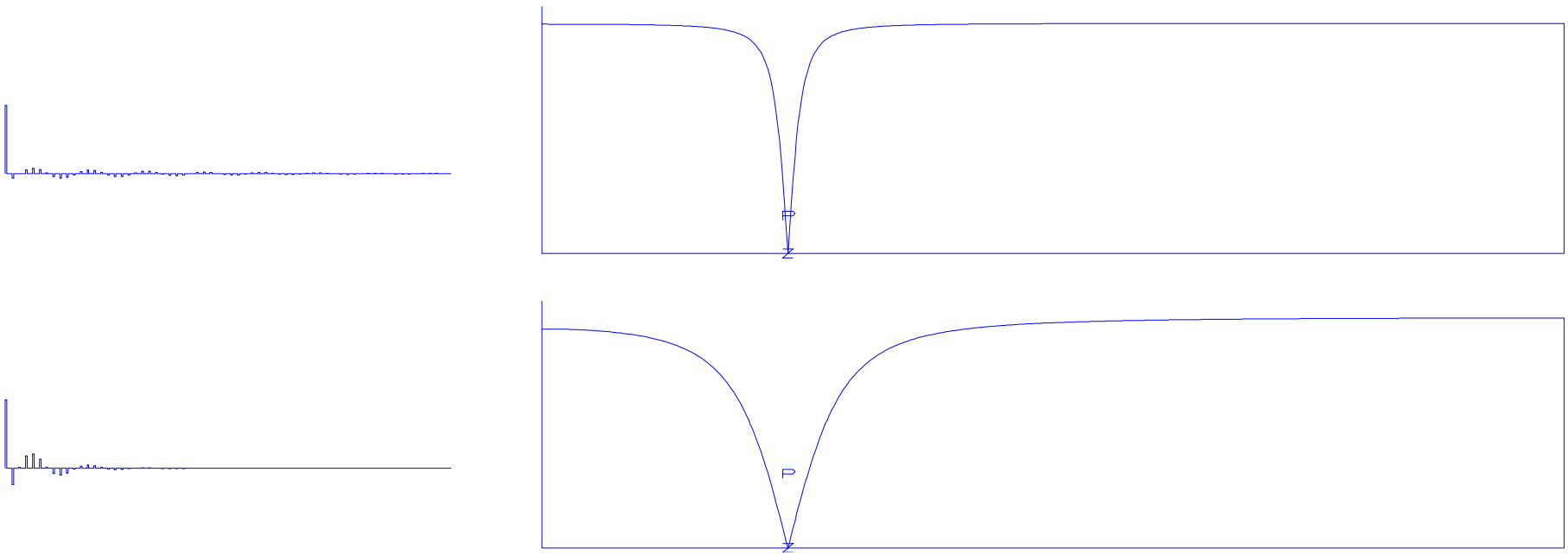


Figure 3.15: Top: a zero on the real frequency axis and a pole just above it give a notch filter; i.e., the zeroed frequency is rejected while other frequencies are little changed. Bottom: the notch has been broadened by moving the pole further away from the zero. (This notch is at 60 Hz, assuming $\Delta t = .002$ s.) zp-notch2 [NR]

3.8. PRECISION EXHAUSTION

As we reach the end of this chapter on poles and feedback filtering, we might be inclined to conclude that all is well if poles are outside the unit circle and that they may even come close to the circle. Further, if we accept anticausal filtering, poles can be inside the unit circle as well.

Reality is more difficult. Big trouble can arise from just a modest clustering of poles at a moderate distance from the unit circle. This is shown in Figure 3.16, where the result is completely wrong. The spectrum should look like the spectrum in Figure 3.8 multiplied by itself about six or seven times, once for each pole. The effect of such repetitive multiplication is to make the small spectral values become very small. When I added the last pole to Figure 3.16, however, the spectrum suddenly became rough. The time response now looks almost divergent. Moving poles slightly creates very different plots. I once had a computer that crashed whenever I included one too many poles.

To understand this, notice that the peak spectral values in Figure 3.16 come from the *minimum* values of the denominator. The denominator will not go to a properly small value if the **precision** of its terms is not adequate to allow them to extinguish one another. Repetitive multiplication has caused the dynamic range

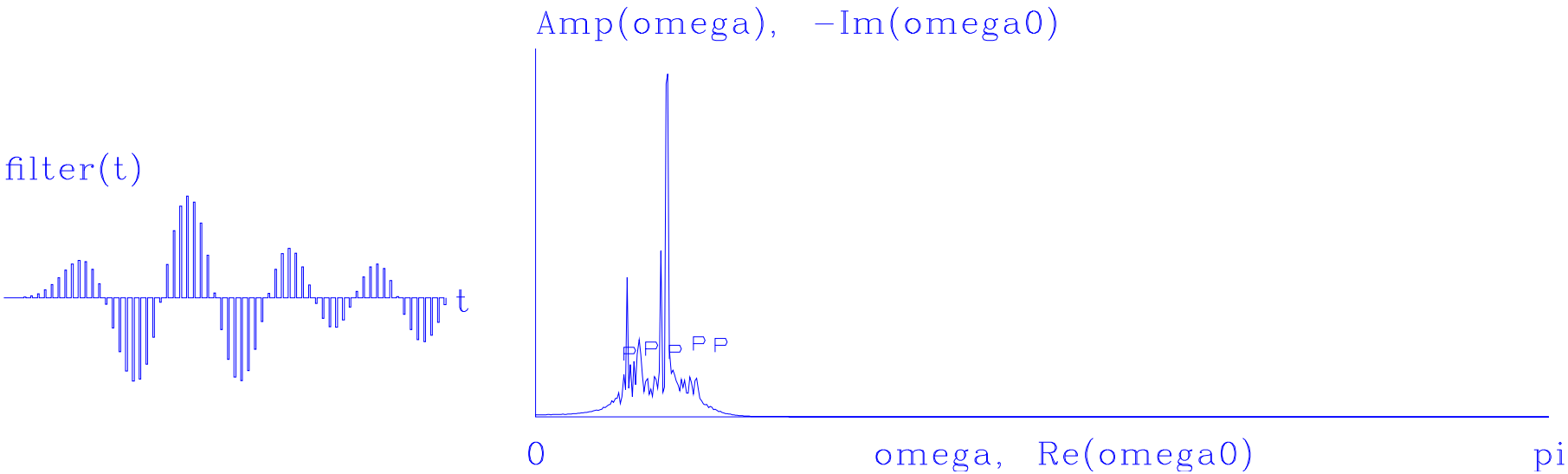


Figure 3.16: A pathological failure when poles cluster too much. This situation requires more than single-word precision. zp-path [NR]

(the range between the largest and smallest amplitudes as a function of frequency) of single-precision arithmetic, about 10^6.

When single-word precision becomes a noticeable problem, the obvious path is to choose double precision. But considering that most geophysical data has a precision of less than one part in a hundred, and only rarely do we see precision of one part in a thousand, we can conclude that the failure of single-word precision arithmetic, about one part in 10^{-6}, is more a sign of conceptual failure than of numerical precision inadequacy.

If an application arises for which you really need an operator that raises a polynomial to a high degree, you may be able to accomplish your goal by applying the operator in stages. Say, for example, you need the all-pass filter $(.2 - Z)^{100}/(1 - .2Z)^{100}$. You should be able to apply this filter in a hundred stages of $(.2 - Z)/(1 - .2Z)$, or maybe in ten stages of $(.2 - Z)^{10}/(1 - .2Z)^{10}$.

Other ways around this precision problem are suggested by reflection-coefficient modeling in a layered earth, described in FGDP.

3.9. MY FAVORITE WAVELET

I will describe my favorite wavelet for seismic modeling, shown in Figure 3.17. Of course the ideal wavelet is an impulse, but the wavelet I describe is intended to mimic real life. I use some zeros at high frequency to force continuity in the time domain and a zero at the origin to suppress zero frequency. I like to simulate the suppression of low-frequency ground roll, so I put another zero not at the origin, but at a low frequency. Theory demands a conjugate pair for this zero; effectively, then, there are three roots that suppress low frequencies. I use some poles to skew the passband toward low frequencies. These poles also remove some of the oscillation caused by the three zeros. (Each zero is like a derivative and causes another lobe in the wavelet.) There is a trade-off between having a long low-frequency tail and having a rapid spectral rise just above the ground roll. The trade-off is adjustable by repositioning the lower pole. The time-domain wavelet shows its high frequencies first and its low frequencies only later. I like this wavelet better than the Ricker wavelet (second derivative of a Gaussian). My wavelet does not introduce as much signal delay. It looks like an impulse response from the physical world.

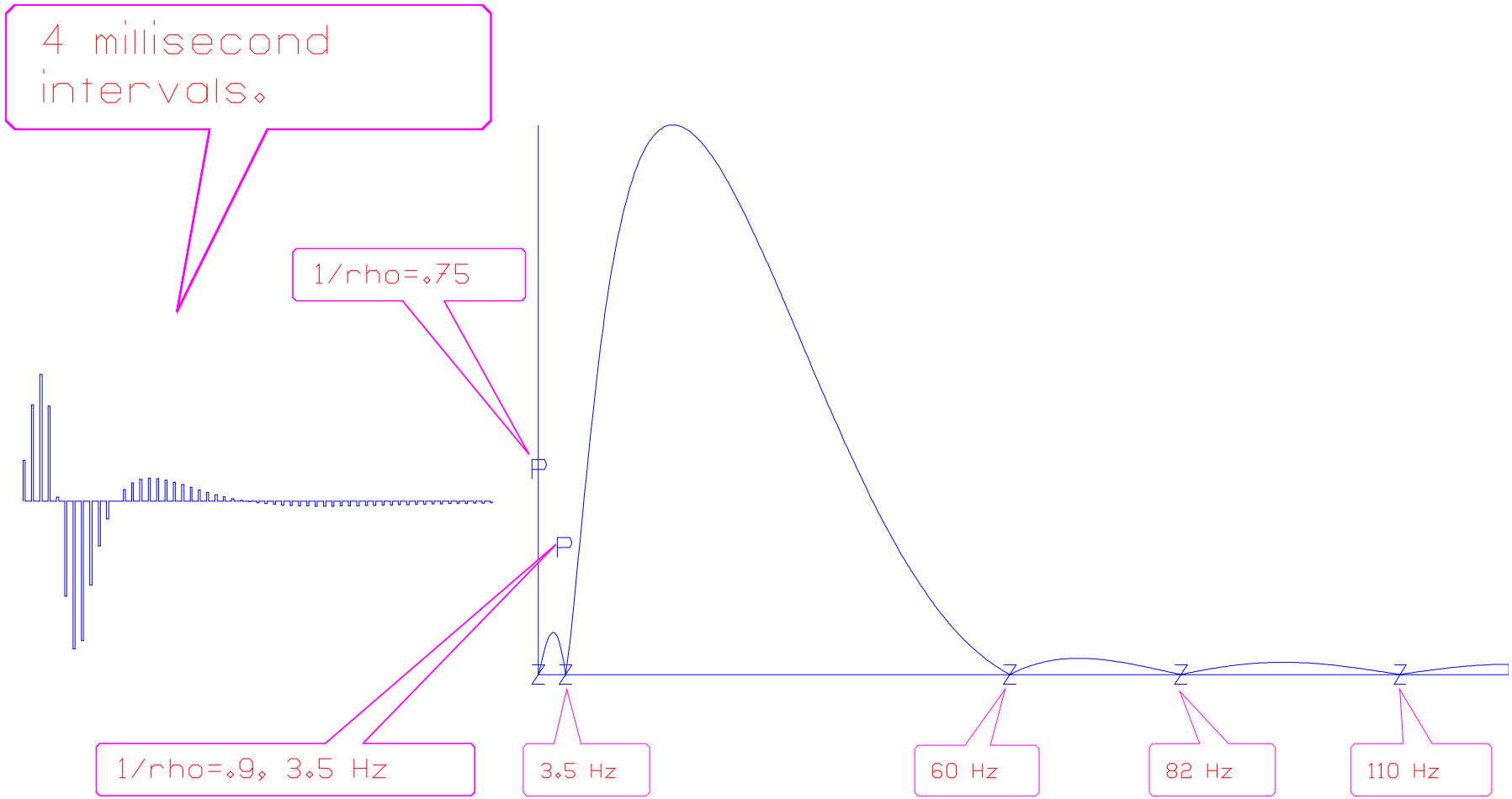


Figure 3.17: My favorite wavelet for seismic modeling. zp-favorite [NR]

3.10. IMPEDANCE FILTERS

Impedance filters are a special class of minimum-phase filters that model energy-conserving devices and media. The real part of the Fourier transform of an impedance filter is positive. Impedances play a basic role in mathematical physics. There are simple ways of making complicated mechanical systems from simple ones, and corresponding mathematical rules allow construction of complicated impedances from simple ones. Also, impedances can be helpful in stabilizing numerical calculations. Logically, a chapter on impedance filters belongs here, but I have little to add to what is already found in FGDP and IEI. FGDP describes the impedance concept in sampled time and its relation to special matrices called "**Toeplitz**" matrices. IEI describes impedances in general as well as their role in physical modeling and imaging with the wave equation.

Chapter 4

Univariate problems

This chapter looks at problems in which there is just one unknown. These "**univariate**" problems illustrate some of the pitfalls, alternatives, and opportunities in data analysis. Following our study of univariate problems we move on, in the next five chap-

ters, to problems with *multiple* unknowns (which obscure the pitfalls, alternatives, and opportunities).

4.1. INSIDE AN ABSTRACT VECTOR

In engineering, a vector has three scalar components which correspond to the three dimensions of the space in which we live. In least-squares data analysis, a vector is a one-dimensional array that can contain many different things. Such an array is an "**abstract vector**." For example, in earthquake studies, the vector might contain the time an earthquake began as well as its latitude, longitude, and depth. Alternately, the abstract vector might contain as many components as there are seismometers, and each component might be the onset time of an earthquake. In signal analysis, the vector might contain the values of a signal at successive instants in time or, alternately, a collection of signals. These signals might be "**multiplex**ed" (interlaced) or "demultiplexed" (all of each signal preceding the next). In image analysis, the one-dimensional array might contain an image, which could itself be thought of as an array of signals. Vectors, including abstract vectors, are usually denoted by boldface letters such as **p** and **s**. Like physical vectors, abstract vectors are **orthogonal**

when their dot product vanishes: $\mathbf{p} \cdot \mathbf{s} = 0$. Orthogonal vectors are well known in physical space; we will also encounter them in abstract vector space.

4.2. SEGREGATING P AND S CROSSTALK

Signals can be contaminated by other signals, and images can be contaminated by other images. This contamination is called "**crosstalk**." An everyday example in seismology is the mixing of **pressure wave**s and **shear wave**s. When waves come straight up, vertical detectors record their pressure-wave component, and horizontal detectors record their shear-wave component. Often, however, waves do not come exactly straight up. In these cases, the simple idealization is contaminated and there is crosstalk. Here we study a simplified form of this signal-corruption problem, as given by the equations

$$\mathbf{v} = \mathbf{p} + \alpha \mathbf{s} + \mathbf{n} \tag{4.1}$$

$$\mathbf{h} = \mathbf{s} + \alpha' \mathbf{p} + \mathbf{n}' \tag{4.2}$$

where $\mathbf{v}$ and $\mathbf{h}$ represent vertical and horizontal observations of earth motion, $\mathbf{p}$ and $\mathbf{s}$ represent theoretical pressure and shear waves, $\mathbf{n}$ and $\mathbf{n}'$ represent noises, and α

and α' are the cross-coupling parameters. You can think of **v**, **h**, **p**, **s**, **n** and **n**$'$ as collections of numbers that can be arranged into a signal or into an image. Mathematically, they are **abstract vector**s. In our notation, boldface **v** represents the vector as a whole, and italic v represents any single component in it. (Traditionally, a component is denoted by v_i.)

- **Two univariate problems**

Communication channels tend to mix information in the way equations (4.1) and (4.2) do. This is "**crosstalk**." Everything on the right sides of equations (4.1) and (4.2) is unknown. This problem can be formulated in an elaborate way with estimation theory. Here we will postpone the general theory and leap to guess that the pressure-wave field **p** will be some linear combination of **v** and **h**, and the shear-wave component **s** will be something similar:

$$\mathbf{p} = \mathbf{v} - \alpha\mathbf{h} \tag{4.3}$$

$$\mathbf{s} = \mathbf{h} - \alpha'\mathbf{v} \tag{4.4}$$

We will understand the crosstalk question to ask us to find the constant value of α and of α'. Although I will describe only the mathematics of finding α, each figure

will show you the results of both estimations, by including one part for α and one part for α'. The results for α and α' differ, as you will see, because of differences in **p** and **s**.

- **The physics of crosstalk**

Physically, the value of α depends on the angle of incidence, which in turn depends critically on the soil layer. The soil layer is generally ill defined, which is why it is natural to take α as an unknown. In real life α should be time-dependent, but we will ignore this complication.

4.2.1. Failure of straightforward methods

The conventional answer to the crosstalk question is to choose α so that $\mathbf{p} = \mathbf{v} - \alpha\mathbf{h}$ has minimum power. The idea is that since adding one signal **p** to an independent signal **s** is likely to increase the power of **p**, removing as much power as possible may be a way to separate the independent components. The theory proceeds as follows. Minimize the dot product

$$\text{Energy} \quad = \quad \mathbf{p}\cdot\mathbf{p} \quad = \quad (\mathbf{v}-\alpha\mathbf{h})\cdot(\mathbf{v}-\alpha\mathbf{h}) \tag{4.5}$$

by differentiating the energy with respect to α, and set the derivative to zero. This gives

$$\alpha \quad = \quad \frac{\mathbf{v}\cdot\mathbf{h}}{\mathbf{h}\cdot\mathbf{h}} \tag{4.6}$$

Likewise, minimizing $(\mathbf{s}\cdot\mathbf{s})$ yields $\alpha' = (\mathbf{h}\cdot\mathbf{v})/(\mathbf{v}\cdot\mathbf{v})$.

In equation (4.5) the "**fitting function**" is **h**, because various amounts of **h** can be subtracted to minimize the power in the residual $(\mathbf{v}-\alpha\mathbf{h})$. Let us verify the well-known fact that after the energy is minimized, the **residual** is **orthogonal** to the **fitting function**. Take the dot product of the fitting function **h** and the residual $(\mathbf{v}-\alpha\mathbf{h})$, and insert the optimum value of α from equation (4.6):

$$\begin{aligned} \mathbf{h}\cdot(\mathbf{v}-\alpha\mathbf{h}) &= \mathbf{h}\cdot\mathbf{v}-\alpha\mathbf{h}\cdot\mathbf{h} \\ &= 0 \end{aligned}$$

Results for both **p** and **s** are shown in Figure 4.1. At first it is hard to believe the result: the crosstalk is *worse* on the output than on the input. Our eyes are drawn to the weak signals in the open spaces, which are obviously unwanted new crosstalk. We do not immediately notice that the new crosstalk has a negative polarity. Negative polarity results when we try to extinguish the strong positive polarity of the

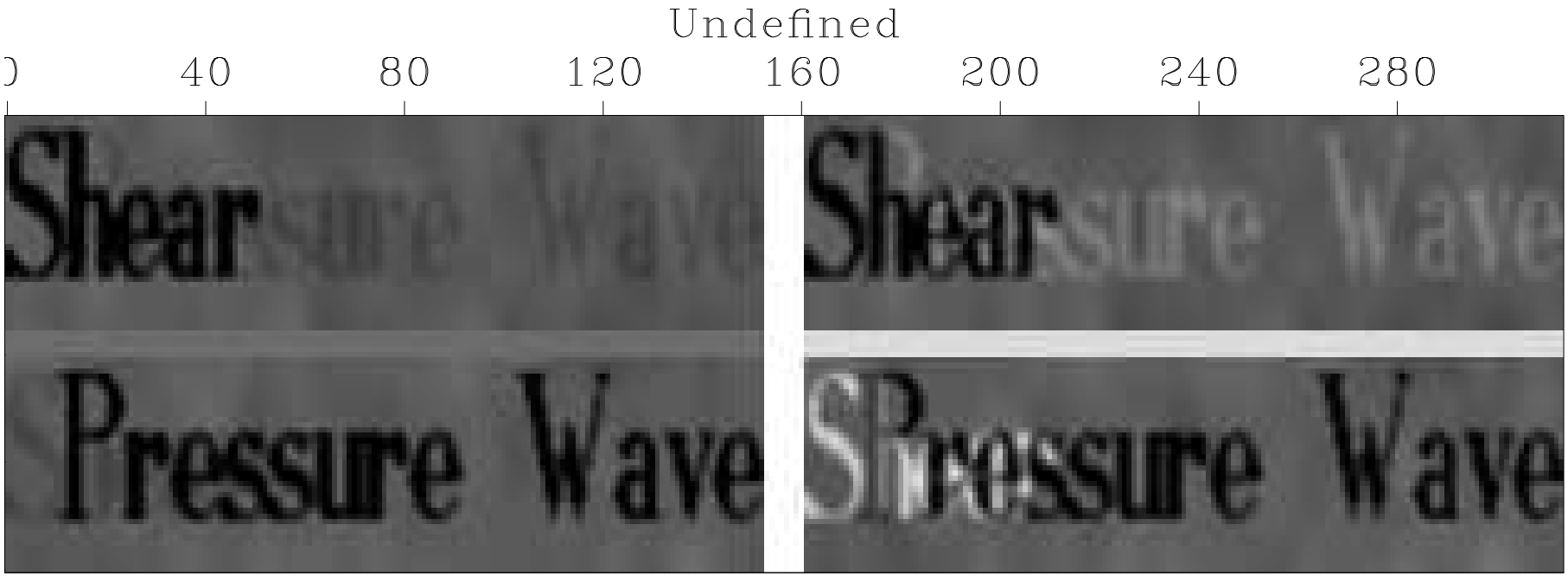


Figure 4.1: Left shows two panels, a “Pressure Wave” contaminated by crosstalk from “Shear” and vice versa. Right shows a least-squares attempt to remove the crosstalk. It is disappointing to see that the crosstalk has become worse. uni-uniform [ER]

main signal. Since the residual misfit is *squared*, our method tends to ignore small residuals and focus attention on big ones: hence the wide-scale growth of small residuals.

The least-squares method is easy to oversimplify, and it is not unusual to see it give disappointing results. Real-life data are generally more complicated than artificial data like the data used in these examples. It is always a good idea to test programs on such **synthetic data** since the success or failure of a least-squares method may not be apparent if the method is applied to real data without prior testing.

- **Failure of independence assumption**

The example in Figure 4.1 illustrates a **pitfall** of classical inversion theory. Had **p** not overlapped **s**, the crosstalk would have been removed perfectly. We were not interested in destroying **p** with **s**, and vice versa. This result was just an accidental consequence of their overlap, which came to dominate the analysis because of the squaring in least squares. Our failure could be attributed to a tacit assumption that since **p** and **s** are somehow "independent," they can be regarded as *orthogonal*, i.e., that $\mathbf{p} \cdot \mathbf{s} = 0$. But the (potential) physical independence of **p** and **s** does nothing

to make a short sample of **p** and **s** orthogonal. Even vectors containing random numbers are unlikely to be orthogonal unless the vectors have an infinite number of components. Perhaps if the text were as long as the works of Shakespeare . . . .

4.2.2. Solution by weighting functions

Examining Figure 4.1, we realize that our goals were really centered in the quiet regions. We need to boost the importance of those quiet regions in the analysis. What we need is a **weighting function**. Denote the i-th component of a vector with the subscript i, say v_i. When we minimize the sums of squares of $v_i - \alpha h_i$, the weighting function for the i-th component is

$$w_i = \frac{1}{v_i^2 + \sigma^2} \tag{4.7}$$

and the minimization itself is

$$\min_{\alpha} \left[\sum_i w_i (v_i - \alpha h_i)^2 \right] \tag{4.8}$$

To find α', the weighting function would be $w = 1/(h^2 + \sigma^2)$.

The detailed form of these weighting functions is not important here. The form I chose is somewhat arbitrary and may be far from optimal. The choice of the constant σ is discussed on page 200. What is more important is the idea that instead of minimizing the sum of *errors themselves*, we are minimizing something like the sum of **relative error**s. Weighting makes any region of the data plane as important as any other region, regardless of whether a letter (big signal) is present or absent (small signal). It is like saying a zero-valued signal is just as important as a signal with any other value. A zero-valued signal carries information.

When signal strength varies over a large range, a nonuniform weighting function should give better regressions. The task of weighting-function design may require some experimentation and judgment.

- **A nonlinear-estimation method**

What I have described above represents my first iteration. It can be called a "**linear-estimation** method." Next we will try a "**nonlinear-estimation** method" and see

that it works better. If we think of minimizing the *relative* error in the residual, then in linear estimation we used the wrong divisor—that is, we used the squared data v^2 where we should have used the squared residual $(v-\alpha h)^2$. Using the wrong divisor is roughly justified when the crosstalk α is small because then v^2 and $(v-\alpha h)^2$ are about the same. Also, at the outset the residual was unknown, so we had no apparent alternative to v^2, at least until we found α. Having found the residual, we can now use it in a second iteration. A second iteration causes α to change a bit, so we can try again. I found that, using the same data as in Figure 4.1, the sequence of iterations converged in about two iterations. Figure 4.2 shows the results of the various weighting methods. Mathematical equations summarizing the bottom row of this figure are:

$$\text{left}: \qquad \min_{\alpha} \sum_i (v_i - \alpha h_i)^2 \tag{4.9}$$

$$\text{middle}: \qquad \min_{\alpha_0} \sum_i \frac{1}{v_i^2 + \sigma^2} (v_i - \alpha_0 h_i)^2 \tag{4.10}$$

$$\text{right}: \qquad \lim_{n\to\infty} \min_{\alpha_n} \sum_i \frac{1}{(v_i - \alpha_{n-1} h_i)^2 + \sigma^2} (v_i - \alpha_n h_i)^2 \tag{4.11}$$

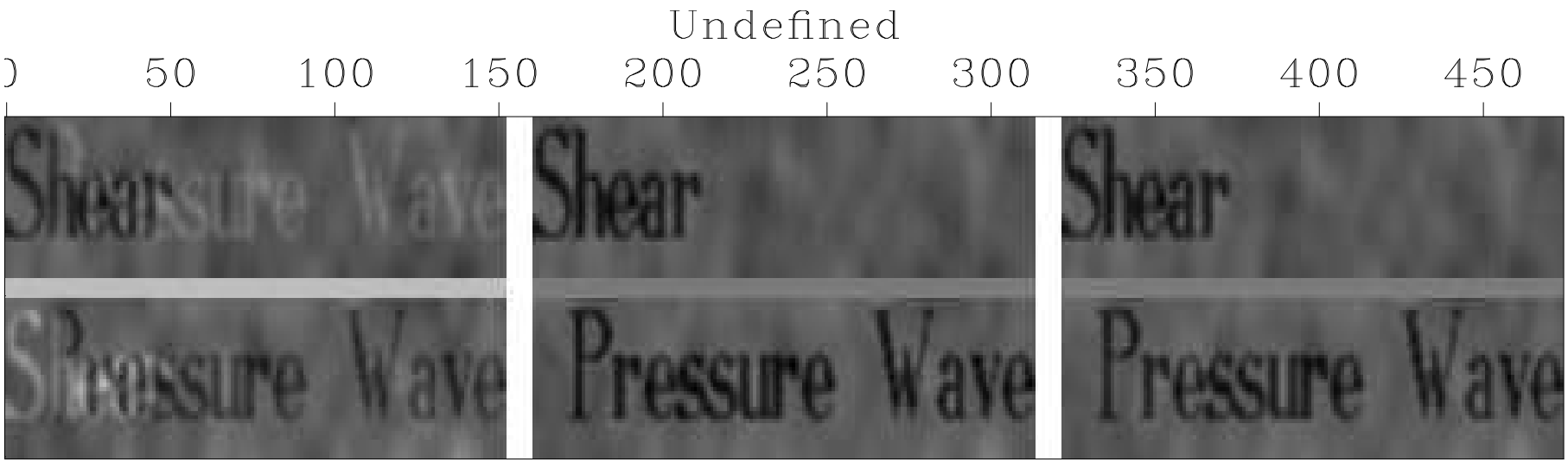


Figure 4.2: Comparison of weighting methods. Left shows crosstalk as badly removed by uniformly weighted least squares. Middle shows crosstalk removed by deriving a weighting function from the input *data*. Right shows crosstalk removed by deriving a weighting function from the fitting *residual*. Press button for movie over iterations. uni-reswait [ER,M]

For the top row of the figure, these equations also apply, but **v** and **h** should be swapped.

- **Clarity of nonlinear picture**

You should not have any difficulty seeing on the figure that the uniform weight leaves the most crosstalk, the nonuniform weights of the linear-estimation method leave less crosstalk, and the nonlinear-estimation method leaves no visible crosstalk. If you cannot see this, then I must blame the method of reproduction of the figures, because the result is clear on the originals, and even clearer on the video screen from which the figure is derived. On the video screen the first iteration is clearly inferior to the result of a few more iterations, but on the printed page these different results are not so easy to distinguish.

- **Nonuniqueness and instability**

We cannot avoid defining σ^2, because without it, any region of zero signal would get an infinite weight. This is likely to lead to undesirable performance: in other words, although with the data of Figure 4.2 I found rapid convergence to a satisfactory answer, there is no reason that this had to happen. The result could also have failed

to converge, or it could have converged to a nonunique answer. This unreliable performance is why academic expositions rarely mention estimating weights from the data, and certainly do not promote the nonlinear-estimation procedure. We have seen here how important these are, however.

I do not want to leave you with the misleading impression that convergence in a simple problem always goes to the desired answer. With the program that made these figures, I could easily have converged to the wrong answer merely by choosing data that contained too much crosstalk. In that case both images would have converged to **s**. Such instability is not surprising, because when α exceeds unity, the meanings of **v** and **h** are reversed.

- **Estimating the noise variance**

Choosing σ^2 is a subjective matter; or at least how we choose σ^2 could be the subject of a lengthy philosophical analysis. Perhaps that is why so much of the literature ignores this question. Without any firm theoretical basis, I chose $|\sigma|$ to be approximately the noise level. I estimated this as follows.

The simplest method of choosing σ^2 is to find the average v^2 in the plane and then choose some arbitrary fraction of it, say 10% of the average. Although this

method worked in Figure 4.2, I prefer another. I chose σ^2 to be the **median** value of v^2. (In other words, we conceptually prepare a list of the numbers v^2; then we sort the list from smallest to largest; and finally we choose the value in the middle. In reality, median calculation is quicker than sorting.)

Notice that Figure 4.2 uses more initial crosstalk than Figure 4.1. Without the extra crosstalk I found that the first iteration worked so well, the second one was not needed. Thus I could not illustrate the utility of nonlinear estimation without more crosstalk.

- **Colored noise**

I made the noise in Figure 4.2 and 4.3 from random numbers that I filtered spatially to give a lateral coherence on a scale something like the size of a letter—which is somewhat larger than a line (which makes up the letter) width. The noise looks like paper mottling. The spectral **color** (spatial coherence) of the noise does not affect the results much, if at all. In other words, independent random numbers of the same amplitude yield results that are about the same. I chose this particular noise color to maximize the chance that noise can be recognized on a poor reproduction. We can see on Figure 4.2 that the noise amplitude is roughly one-third of the signal

amplitude. This data thus has a significant amount of noise, but since the signal is bigger than the noise, we should really call this "good" data.

Next we will make the noise bigger than the signal and see that we can still solve the problem. We will need more powerful techniques, however.

4.2.3. Noise as strong as signal

First we will make the problem tougher by boosting the noise level to the point where it is comparable to the signal. This is shown in Figure 4.3. Notice that the attempt to remove crosstalk is only partly successful. Interestingly, unlike in Figure 4.1, the crosstalk retains its original polarity, because of the strong noise. Imagine that the noise $\mathbf{n}$ dominated everything: then we would be minimizing something like $(\mathbf{n}_v - \alpha\mathbf{n}_h) \cdot (\mathbf{n}_v - \alpha\mathbf{n}_h)$. Assuming the noises were uncorrelated and sample sizes were infinite, then $\mathbf{n}_v \cdot \mathbf{n}_h = 0$, and the best α would be zero. In real life, samples have finite size, so noises are unlikely to be more than roughly orthogonal, and the predicted α in the presence of strong noise is a small number of random polarity. Rerunning the program that produced Figure 4.3 with different random noise seeds produced results with significantly more and significantly less estimated crosstalk.

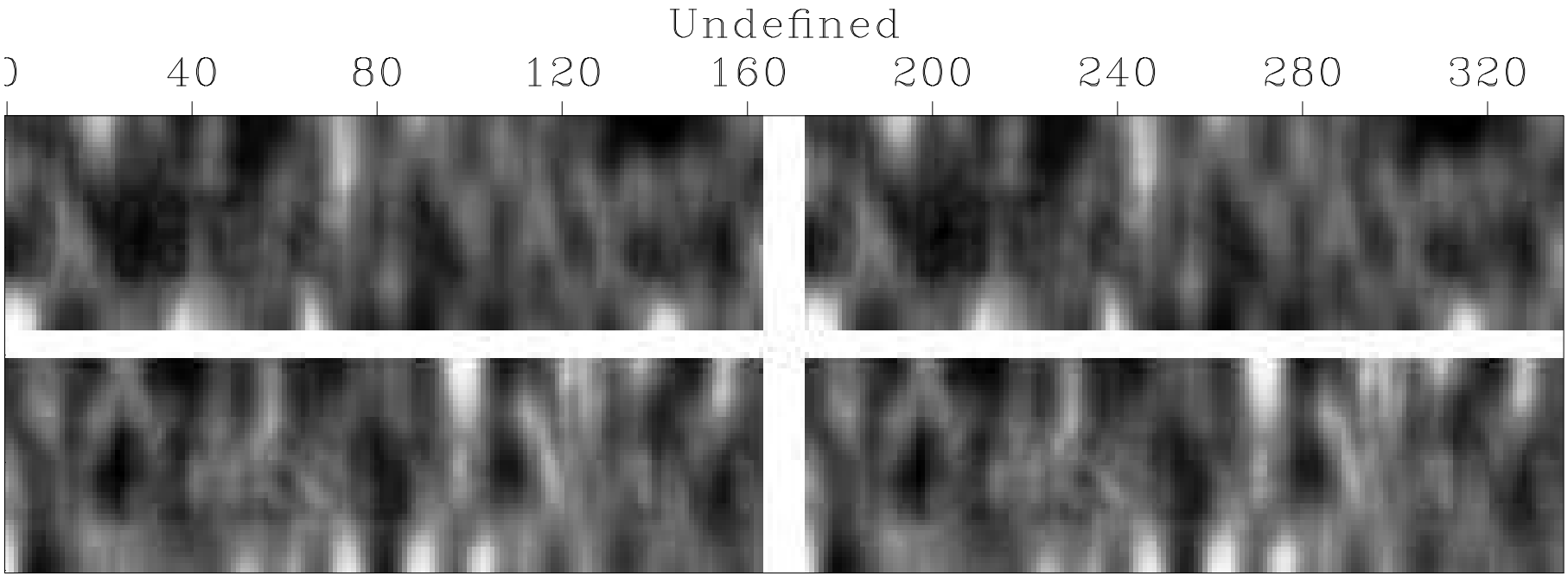


Figure 4.3: Left: data with crosstalk. Right: residuals after attempted crosstalk removal using uniform weights. uni-neqs [ER]

The results are dominated more by the noise than the difference between **p** and **s**. More about random fluctuations with finite sample sizes will follow in chapter 11.

4.2.4. Spectral weighting function

Since we humans can do a better job than the mathematical formulation leading up to Figure 4.3, we naturally want to consider how to reformulate our mathematics to make it work better. Apparently, our eyes sense the difference between the **spatial spectra** of the signals and the noise. Visually, we can suppress the noise because of its noticeably lower frequency. This suggests filtering the data to suppress the noise.

On the filtered data with the noise suppressed, we can estimate the crosstalk parameter α. Of course, filtering the noise will filter the signal too, but we need not display the filtered data, only use it to estimate α. That estimated α is applied to the raw (unfiltered) data and presented as "the answer."

Of course, we may as well display both filtered and unfiltered data and label them accordingly. We might prefer unfiltered noisy images or we might prefer filtered images with less noise. Seismograms present a similar problem. Some people think they prefer to look at a best image of the earth's true velocity, impedance, or

whatever, while others prefer to look at a filtered version of the same, especially if the filter is known and the image is clearer.

Here I chose a simple filter to suppress the low-frequency noise. It may be far from optimal. (What actually is optimal is a question addressed in chapters 7 and 8.) For simplicity, I chose to apply the **Laplacian** operator $\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$ to the images to roughen them, i.e., to make them less predictable. The result is shown in Figure 4.4. The bottom rows are the roughened images. On the left is the input data. Although the crosstalk is visible on both the raw images and the filtered images, it seems more clearly visible on the filtered images. "Visibility" is not the sole criterion here because the human eye can be an effective filter device too. There can be no doubt that the crosstalk has larger amplitude (above the background noise) on the filtered images. This larger amplitude is what is important in the dot-product definition of α. So the bottom panels of filtered data are used to compute α, and the top panels are computed from that α. Finally, notice that the unfiltered data looks somewhat *worse* after crosstalk removal. This is because the combination of **v** and **h** contains noise from each.

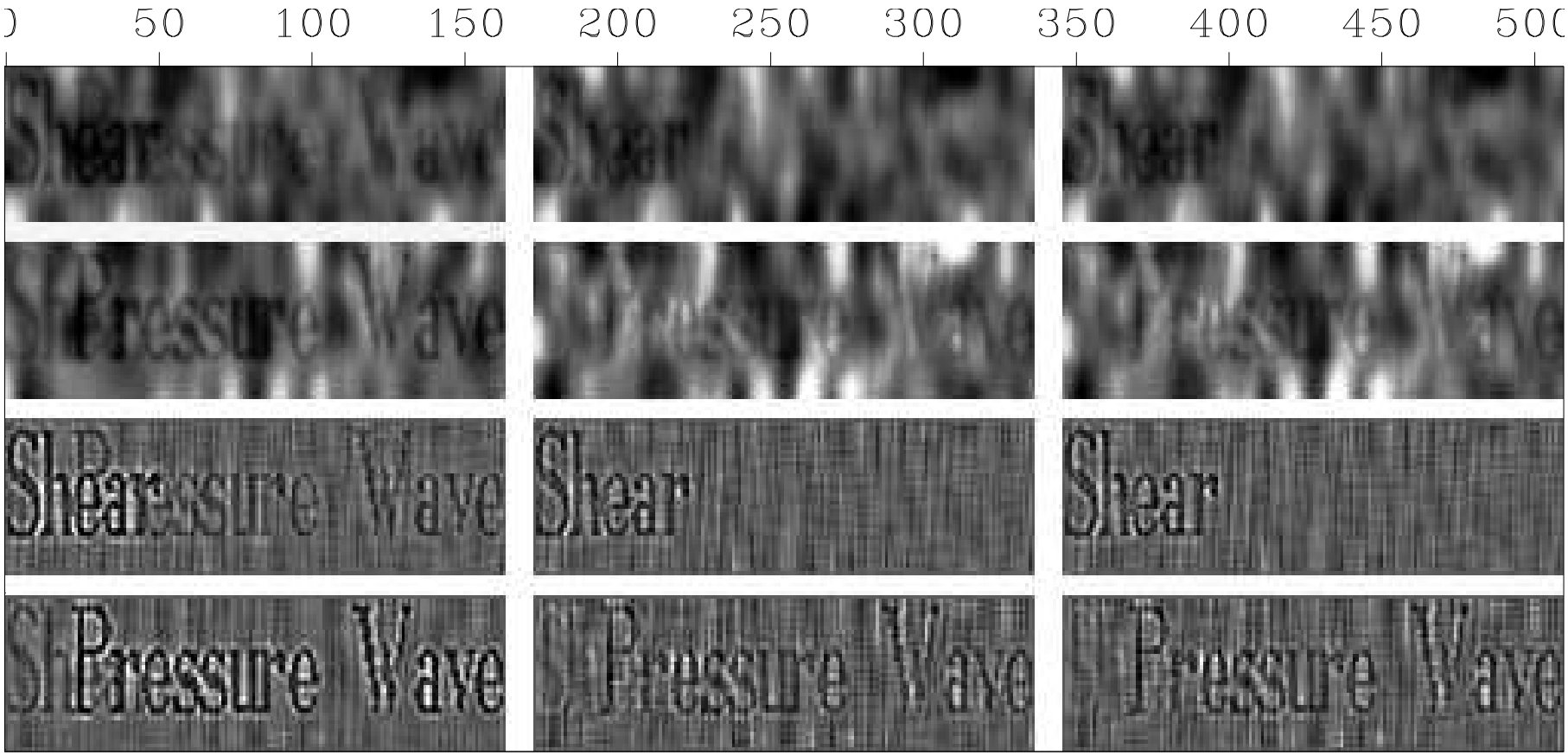


Figure 4.4: Estimation on spatially filtered signals. Top: unfiltered signal with crosstalk. Bottom: filtered signal with crosstalk. Left: input data. Center: residual using uniform weights. Right: residual using inverse-signal weights. uni-rufn [ER]

4.2.5. Flame out

The simple crosstalk problem illustrates many of the features of general modeling and inversion (finding models that fit data). We have learned the importance of weighting functions—not just their amplitudes, but also their spectral amplitudes. Certainly we have known for centuries, from the time of **Gauss** (see Strang, 1986), that the "proper" weighting function is the "inverse **covariance matrix**" of the noise (a generalized relative error, that is, involving the relative amplitudes and relative spectra), formally defined in chapter 11. I do not know that anyone disagrees with Gauss's conclusion, but in real life, it is often ignored. It is hard to find the covariance matrix: we set out to measure a mere *scalar* (α), and Gauss tells us we need to figure out a *matrix* first! It is not surprising that our illustrious statisticians and geophysical theoreticians often leave this stone unturned. As we have seen, different weighting functions can yield widely different answers. Any inverse theory that does not tell us how to choose weighting functions is incomplete.

4.3. References

Aki, K., and Richards, P.G., 1980, Quantitative seismology: theory and methods, vol. 2: W. H. Freeman.

Backus, G.E., and Gilbert, J.F., 1967, Numerical applications of a formalism for geophysical inverse problems: Geophys. J. R. astr. Soc., **13**, 247-276.

Gauss, K.F.: see Strang, 1986.

Menke, W., 1989, Geophysical data analysis: discrete inverse theory, rev. ed.: Academic Press, Inc.

Strang, G., 1986, Introduction to applied mathematics, p. 144: Wellesley-Cambridge Press.

Tarantola, A., 1987, Inverse problem theory: methods for data fitting and model parameter estimation: Elsevier.

4.4. HOW TO DIVIDE NOISY SIGNALS

Another univariate statistical problem arises in Fourier analysis, where we seek a "best answer" at each frequency, then combine all the frequencies to get a best signal. Thus emerges a wide family of interesting and useful applications. However, Fourier analysis first requires us to introduce complex numbers into statistical estimation.

Multiplication in the Fourier domain is convolution in the time domain. Fourier-domain division is time-domain deconvolution. In chapter 3 we encountered the polynomial-division feedback operation $X(Z) = Y(Z)/F(Z)$. This division is challenging when F has observational error. By switching from the Z-domain to the ω-domain we avoid needing to know if F is minimum phase. The ω-domain has pitfalls too, however. We may find for some real ω that $F(Z(\omega))$ vanishes, so we cannot divide by that F. Failure erupts if zero division occurs. More insidious are the poor results we obtain when zero division is avoided by a near miss.

4.4.1. Dividing by zero smoothly

Think of any real numbers x, y, and f and any program containing $x = y/f$. How can we change the program so that it never divides by zero? A popular answer is to change $x = y/f$ to $x = yf/(f^2+\epsilon^2)$, where ϵ is any tiny value. When $|f| >> |\epsilon|$, then x is approximately y/f as expected. But when the divisor f vanishes, the result is safely zero instead of infinity. The transition is smooth, but some criterion is needed to choose the value of ϵ. This method may not be the only way or the best way to cope with **zero division**, but it is a good way, and it permeates the subject of signal analysis.

To apply this method in the Fourier domain, suppose X, Y, and F are complex numbers. What do we do then with $X = Y/F$? We multiply the top and bottom by the complex conjugate $\overline{F}$, and again add ϵ^2 to the denominator. Thus,

$$X(\omega) \quad = \quad \frac{\overline{F(\omega)}\, Y(\omega)}{\overline{F(\omega)}F(\omega) \,+\, \epsilon^2} \tag{4.12}$$

Now the denominator must always be a positive number greater than zero, so division is always safe.

In preparing figures with equation (4.12), I learned that it is helpful to recast

the equation in a scaled form. First replace ϵ^2, which has physical units of $|F|^2$, by $\epsilon^2 = \lambda\sigma_F^2$, where λ is a dimensionless parameter and σ_F^2 is the average value of $\overline{F}F$. Then I rescaled equation (4.12) to

$$X(\omega) \quad = \quad \frac{\overline{F(\omega)}\, Y(\omega)}{\overline{F(\omega)}F(\omega) \,+\, \lambda\sigma_F^2} \;(2+\lambda/2)\sigma_F \tag{4.13}$$

The result is that the scale of X is independent of the scale of F and the scale of λ. This facilitates plotting X over a range of those parameters. I found the 2s in the expression by experimentation. Of course, if the plotting software you are using adjusts a scale factor to fill a defined area, then the scaling may be unimportant. Equation (4.13) ranges continuously from **inverse filter**ing with $X = Y/F$ to filtering with $X = \overline{F}Y$, which is called "**matched filter**ing." Notice that for any complex number F, the phase of $1/F$ equals the phase of $\overline{F}$, so all these filters have the same phase.

The filter $\overline{F}$ is called the "matched filter." If nature created Y by random bursts of energy into F, then building X from Y and F by choosing $\lambda = \infty$ in equation (4.13) amounts to $X = Y\overline{F}$ which **crosscorrelate**s F with the randomly placed copies of F that are in Y.

4.4.2. Damped solution

Equation (4.12) is the solution to an optimization problem that arises in many applications. Now that we know the solution, let us formally define the problem. First, we will solve a simpler problem with real values: we will choose to minimize the quadratic function of x:

$$Q(x) \quad = \quad (fx-y)^2+\epsilon^2 x^2 \tag{4.14}$$

The second term is called a "**damping** factor" because it prevents x from going to $\pm\infty$ when $f \to 0$. Set $dQ/dx = 0$, which gives

$$0 \quad = \quad f(fx-y)+\epsilon^2 x \tag{4.15}$$

This yields the earlier answer $x = fy/(f^2+\epsilon^2)$.

With Fourier transforms, the signal X is a complex number at each frequency ω. So we generalize equation (4.14) to

$$Q(\bar{X},X) \quad = \quad (\overline{FX-Y})(FX-Y)+\epsilon^2\bar{X}X \quad = \quad (\bar{X}\bar{F}-\bar{Y})(FX-Y)+\epsilon^2\bar{X}X \tag{4.16}$$

To minimize Q we could use a real-values approach, where we express $X = u+iv$ in terms of two real values u and v and then set $\partial Q/\partial u = 0$ and $\partial Q/\partial v = 0$. The

approach we will take, however, is to use complex values, where we set $\partial Q/\partial X = 0$ and $\partial Q/\partial \bar{X} = 0$. Let us examine $\partial Q/\partial \bar{X}$:

$$\frac{\partial Q(\bar{X}, X)}{\partial \bar{X}} \quad = \quad \bar{F}(FX - Y) + \epsilon^2 X \quad = \quad 0 \tag{4.17}$$

The derivative $\partial Q/\partial X$ is the complex conjugate of $\partial Q/\partial \bar{X}$. So if either is zero, the other is too. Thus we do not need to specify both $\partial Q/\partial X = 0$ and $\partial Q/\partial \bar{X} = 0$. I usually set $\partial Q/\partial \bar{X}$ equal to zero. Solving equation (4.17) for X gives equation (4.12).

4.4.3. Example of deconvolution with a known wavelet

The top trace of Figure 4.5 shows a marine reflection seismic trace from northern Scandinavia. Its most pronounced feature is a series of multiple reflections from the ocean bottom seen at .6 second intervals. These reflections share a similar waveshape that alternates in **polarity**. The alternation of polarity (which will be more apparent after deconvolution) results from a negative **reflection coefficient** at the ocean surface (where the acoustic pressure vanishes). The spectrum of the top trace has a **comb** pattern that results from the periodicity of the multiples. In Figure 4.5,

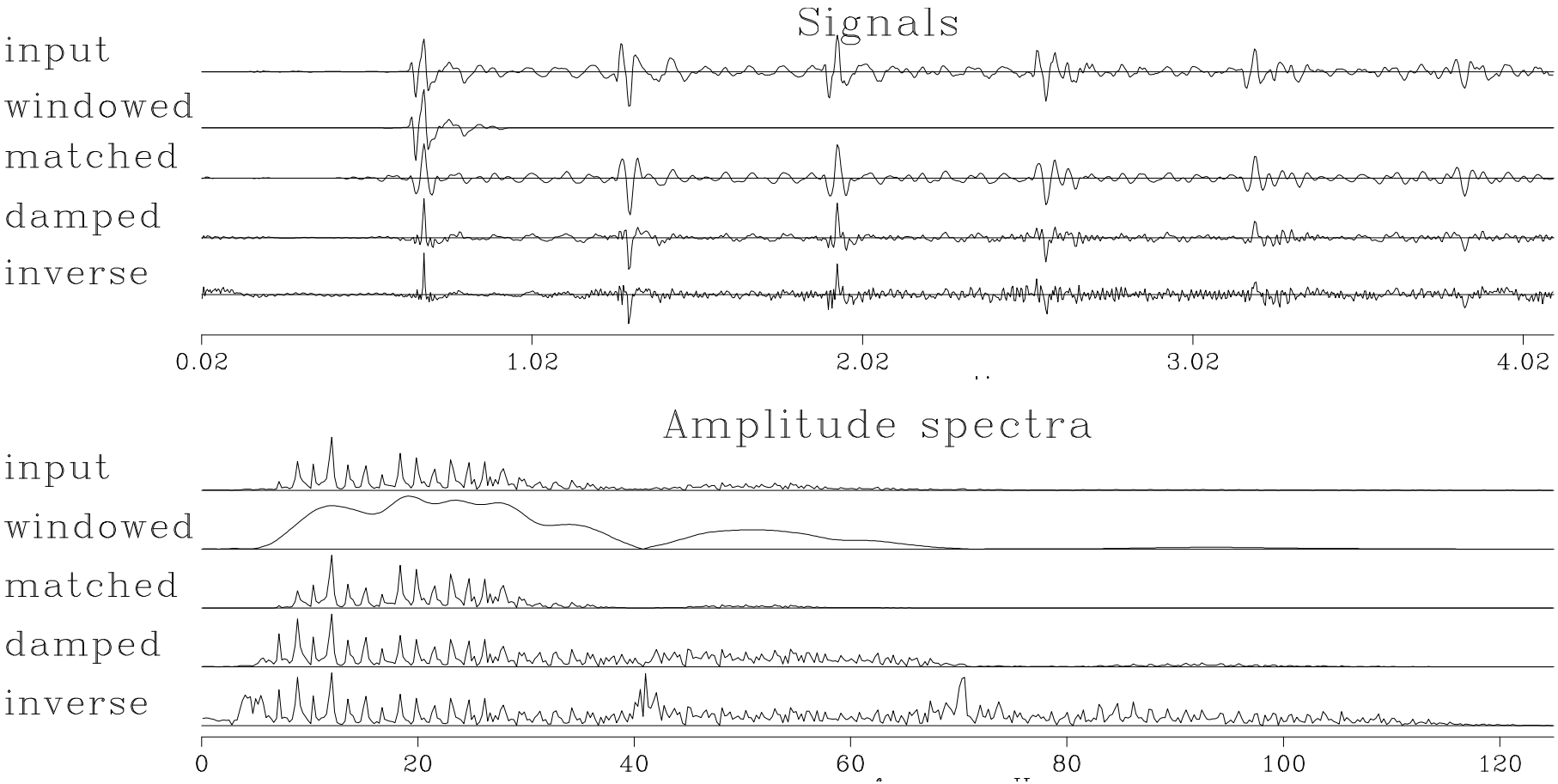


Figure 4.5: The signals on the top correspond to the spectra on the bottom. The top signal is a marine seismogram 4 seconds long. A wavelet windowed between 0.5 s and 1 s was used to deconvolve the signal with various values of λ. (Adapted from Bill **Harlan**, by personal communication.) uni-dekon [ER]

I let the input trace be Y and chose the filter F by extracting (windowing) from Y the water-bottom reflection, as shown in the second trace. The spectrum of the windowed trace is like that of the input trace except that the comb modulation is absent (see chapter 9 for the reason for the appearance of the comb). The trace labeled "matched" in Figure 4.5 is the input after matched filtering, namely $Y\overline{F}$. The trace labeled "damped" shows the result of a value of $\lambda = .03$, my best choice. The wavelets are now single pulses, alternating in polarity. The trace labeled "inverse" is actually not the inverse, but the result of a too small damping factor $\lambda = .001$. The inverse trace is noisy at high frequencies. Notice how the spectral **bandwidth** increases from the matched to the damped to the undamped. Increasing noise (bad) is associated with sharpening of the pulse (good).

Bill **Harlan** and I each experimented with varying λ with frequency but did not obtain results interesting enough to show.

Another example of deconvolution with a known wavelet which is more typical and less successful is shown in Figure 4.6. Here a filter designed in a window on the water-bottom reflection of a single signal fails to succeed in compressing the wavelets of multiple reflections on the same trace. It also fails to compress the water-bottom reflection of a nearby trace. We need more sophisticated methods for

finding the appropriate filter.

4.4.4. Deconvolution with an unknown filter

Equation (4.12) solves $Y = XF$ for X, giving the solution for what is called "the deconvolution problem with a known wavelet F." We can also use $Y = XF$ when the filter F is unknown, but the input X and output Y are given. Here stabilization might not be needed but would be necessary if the input and output did not fill the frequency band. Taking derivatives as above, but with respect to F instead of X, gives again equation (4.12) with X and F interchanged:

$$F(\omega) \quad = \quad \frac{\overline{X(\omega)}\, Y(\omega)}{\overline{X(\omega)} X(\omega) + \epsilon^2} \tag{4.18}$$

4.4.5. Explicit model for noise

In all the signal analysis above there was no explicit model for **noise**, but implicitly the idea of noise was there. Now we will recognize it and solve explicitly for it. This leads to what is called "**linear-estimation** theory." Instead of simply $Y = FX$,

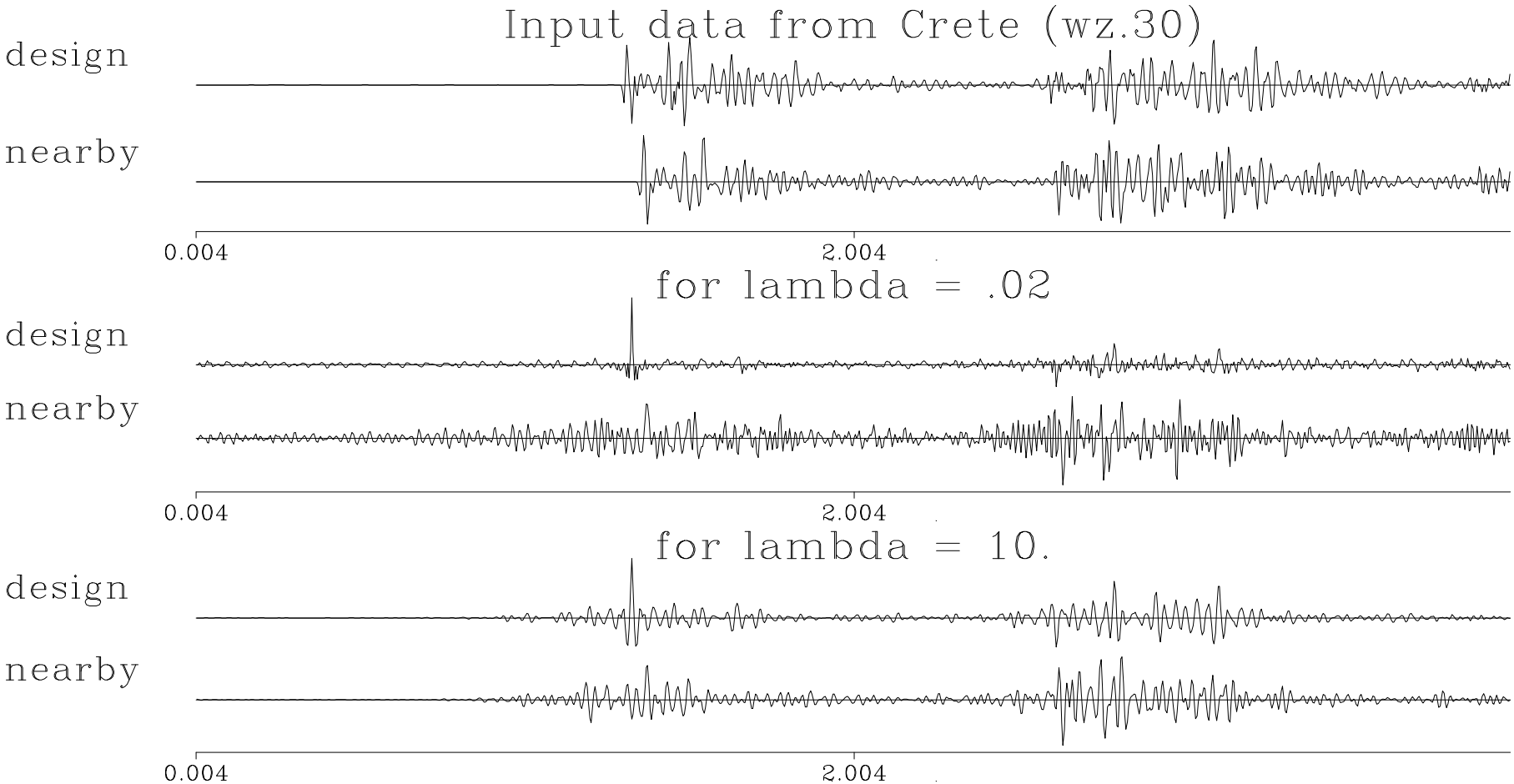


Figure 4.6: Division by water-bottom wavelet. uni-crete [ER]

we add noise $N(\omega)$ into the defining equation:

$$Y(\omega) \quad = \quad F(\omega)X(\omega) + N(\omega) \tag{4.19}$$

To proceed we need to define the "**variance**" (described more fully in chapter 11) as

$$\sigma_X^2 \quad = \quad \frac{1}{n}\sum_{j=1}^{n} \bar{X}(\omega_j)X(\omega_j) \tag{4.20}$$

and likewise the noise variance σ_N^2.

The general linear-estimation method minimizes something that looks like a sum of **relative error**s:

$$Q(X,N) \quad = \quad \frac{\bar{X}\,X}{\sigma_X^2} + \frac{\bar{N}\,N}{\sigma_N^2} \tag{4.21}$$

Notice that the variances put both terms of the sum into the same physical units. I have not derived equation (4.21) but stated it as reasonable: from it we will derive reasonable answers which we have already seen. The rationale for the minimization of (4.21) is that we want the noise to be small, but because we must guard against zero division in $X = Y/F$, we ask for X to be small too. Actually, by introducing

equation (4.19), we have abandoned the model $X = Y/F$ and replaced it with the model $X = (Y - N)/F$. Thus, instead of thinking of falsifying F to avoid dividing by zero in $X = Y/F$, we now think of finding N so the numerator in $(Y - N)/X$ vanishes wherever the denominator does.

By introducing (4.19) into (4.21) we can eliminate either N or X. Eliminating N, we have

$$Q(X) = \frac{\bar{X}\,X}{\sigma_X^2} + \frac{(\overline{FX - Y})(FX - Y)}{\sigma_N^2} \tag{4.22}$$

Minimizing $Q(X)$ by setting its derivative by $\bar{X}$ to zero gives

$$0 = \frac{X}{\sigma_X^2} + \frac{\bar{F}(FX - Y)}{\sigma_N^2} \tag{4.23}$$

$$X = \frac{\bar{F}Y}{\bar{F}F + \frac{\sigma_N^2}{\sigma_X^2}} \tag{4.24}$$

Equation (4.24) is the same as equation (4.12), except that it gives us a numerical interpretation of the value of ϵ in equation (4.12).

We can find an explicit equation for the noise in terms of the data and filter by substituting equation (4.24) into equation (4.19) and solving for N.

4.4.6. A self-fulfilling prophecy?

Equation (4.24) and its surrounding theory are easily misunderstood and misused. I would like to show you a **pitfall**. Equation (4.24) expresses the answer to the deconvolution problem, but does so in terms of the unknowns σ_N^2 and σ_X^2. Given an initial estimate of σ_N^2/σ_X^2, we see that equation (4.24) gives us X and (4.19) gives N, so that we can compute σ_N^2 and σ_X^2. Presumably these computed values are better than our initial guesses. In statistics, the variances in equation (4.24) are called “priors," and it makes sense to check them, and even more sense to correct them. From the corrected values we should be able to iterate, further improving the corrections. Equation (4.24) applies for each of the *many* frequencies, and there is only a *single* unknown, the ratio σ_N^2/σ_X^2. Hence it seems as if we have plenty of information, and the bootstrapping procedure might work. A pessimist might call this bootstrapping a self-fulfilling prophecy, but we will see. What do you think?

Truth is stranger than fiction. I tried bootstrapping the variances. With my first

starting value for the ratio σ_N^2/σ_X^2, iterating led to the ratio being infinite. Another starting value led to the ratio being zero. All starting values led to zero or infinity. Eventually I deduced that there must be a **metastable** starting value. Perhaps the metastable value is the appropriate one, but I lack a rationale to assert it. It seems we cannot bootstrap the variances because the solutions produced do not tend to the correct variance, nor is the variance ratio correct. Philosophically, we can be thankful that these results failed to converge, since this outcome prevents us from placing a false confidence in the bootstrapping procedure.

> The variance of the solution to a least-squares problem is not usable to bootstrap to a better solution.

I conclude that **linear-estimation** theory, while it appears to be a universal guide to practice, is actually incomplete. Its incompleteness grows even more significant in later chapters, when we apply least squares to multivariate problems where the scalar σ_x^2 becomes a matrix. We continue our search for "universal truth" by studying more examples.

EXERCISES:

1 Using the chain rule for differentiation, verify that $\partial Q/\partial u = 0$ and $\partial Q/\partial v = 0$ is equivalent to $\partial Q/\partial \overline{x}$, where $x = u + iv$.

2 Write code to verify the instability in estimating the variance ratio.

4.5. NONSTATIONARITY

Frequencies decrease with time; velocities increase with depth. Reverberation periods change with offset; dips change with location. Still, we often find it convenient to presume that the relevant statistical aspects of data remain constant over a large domain. In mathematical statistics this is called a “stationarity" assumption. To assume stationarity is to enjoy a simplicity in analysis that has limited applicability in the real world. To avoid seduction by the **stationarity** assumption we will solve here a problem in which stationarity is obviously an unacceptable presumption. We will gain skill in and feel comfortable with the computer techniques of estimation in moving windows. The first requirement is to learn reliable ways of limiting the potentially destructive effects of the edges of windows.

The way to cope with spatial (or temporal) variation in unknown parameters is to estimate them in moving windows. Formulating the estimation might require special shrewdness so that window edges do not strongly affect the result.

To illustrate computation technique in a nonstationary environment, I have chosen the problem of **dip** estimation. Before we take up this problem, however, we will examine a generic program for moving a window around on a wall of data. The window-moving operation is so cluttered that the first example of it simply counts the number of windows that hit each point of the wall. Inspecting subroutine `nonstat()` /prog:nonstat we first notice that the 1-axis is handled identically with the 2-axis. (`Ratfor` makes this more obvious than `Fortran` could.) Notice the bottom of the loops where variables `(e1,e2)` which will be the ends of the windows are jumped along in steps of `(j1,j2)`. Then notice the tops of the loops where processing terminates when the ends of the windows pass the ends of the wall. Also at the tops of the loops, the window count `(k1,k2)` is incremented, and the starting points of each window are defined as the window ends `(e1,e2)` minus their widths `(w1,w2)`. nonstat

```
# slide a window around on a wall of data.  Count times each data point used.
#
subroutine nonstat( n1,n2, w1,w2, j1,j2, count)
integer n1,n2           # size of data wall
integer w1,w2           # size of window
integer j1,j2           # increments for jumping along the wall
integer s1,s2, e1,e2    # starting and ending points of window on wall
integer k1,k2           # output math size of array of windows
integer i1,i2
real       count( n1,n2)
call null( count, n1*n2)
k2=0; e2=w2; while( e2<=n2) {  k2=k2+1; s2=e2-w2+1
k1=0; e1=w1; while( e1<=n1) {  k1=k1+1; s1=e1-w1+1
        do i1= s1, e1 {
        do i2= s2, e2 {
                count(i1,i2) =  count(i1,i2) + 1.
                }}
        e1=e1+j1            }
        e2=e2+j2            }
return; end
```

Back

A sample result is shown in Figure 4.7. Since window widths do not match window jumps, the count is not a constant function of space. We see ridges where the rectangles overlapped a little. Likewise, since the windows were not fitted to the wall, some data values near the end of each axis failed to be used in any window. Next we address the problem of splicing together data processing outputs

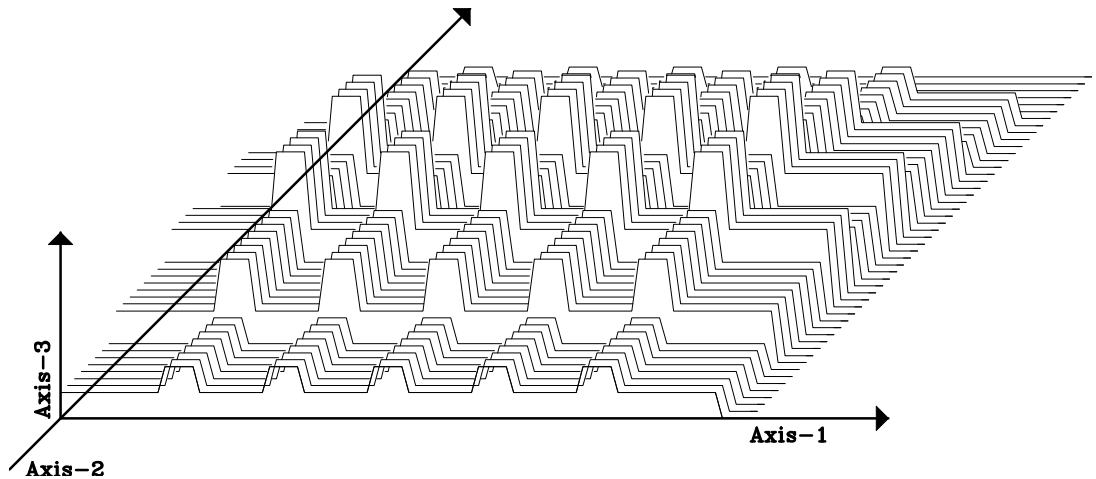


Figure 4.7: Sample output of `nonstat()` with n1=100, w1=20, j1=15, n2=50, w2=20, j2=8. uni-nonstat [ER]

derived in each window. This could be done with rectangle weights derived from `count` in subroutine `nonstat()` but it is not much more difficult to patch together triangle weighting functions as shown in subroutine `nonstat2()` /prog:nonstat2.

nonstat2 Triangles allow for a more gradual transition from one window to an-

```
# slide a window around on a wall of data.  Triangle weighting.
#
subroutine nonstat2( n1,n2, w1,w2, j1,j2, data, output, weight)
integer              n1,n2, w1,w2, j1,j2,          s1,s2, e1,e2, k1,k2, i1,i2
real    data(n1,n2), output(n1,n2), weight(n1,n2),  triangle1, triangle2, shape
temporary real window(w1,w2), winout(w1,w2)
call null( weight, n1*n2)
call null( output, n1*n2)
k2=0; e2=w2; while( e2<=n2) {  k2=k2+1;  s2=e2-w2+1
k1=0; e1=w1; while( e1<=n1) {  k1=k1+1;  s1=e1-w1+1
        do i1= 1, w1 {
        do i2= 1, w2 {  window(i1,i2) = data(s1+i1-1,s2+i2-1)
                        }}
        do i1= 1, w1 {                          # Trivial data processing
        do i2= 1, w2 {  winout(i1,i2) = window(i1,i2)
                        }}
        do i1= s1, e1 { triangle1= amax1(0., 1. - abs(i1-.5*(e1+s1)) / (.5*w1))
        do i2= s2, e2 { triangle2= amax1(0., 1. - abs(i2-.5*(e2+s2)) / (.5*w2))
                 shape = triangle1 * triangle2
                 output(i1,i2) = output(i1,i2) + shape * winout(i1-s1+1,i2-s2+1)
                 weight(i1,i2) = weight(i1,i2) + shape
                 }}
        e1=e1+j1            }
        e2=e2+j2            }
do i1= 1, n1 {
do i2= 1, n2 {  if( weight(i1,i2) > 0. )
                        output(i1,i2) = output(i1,i2) / weight(i1,i2)
        }}
return; end
```

Back

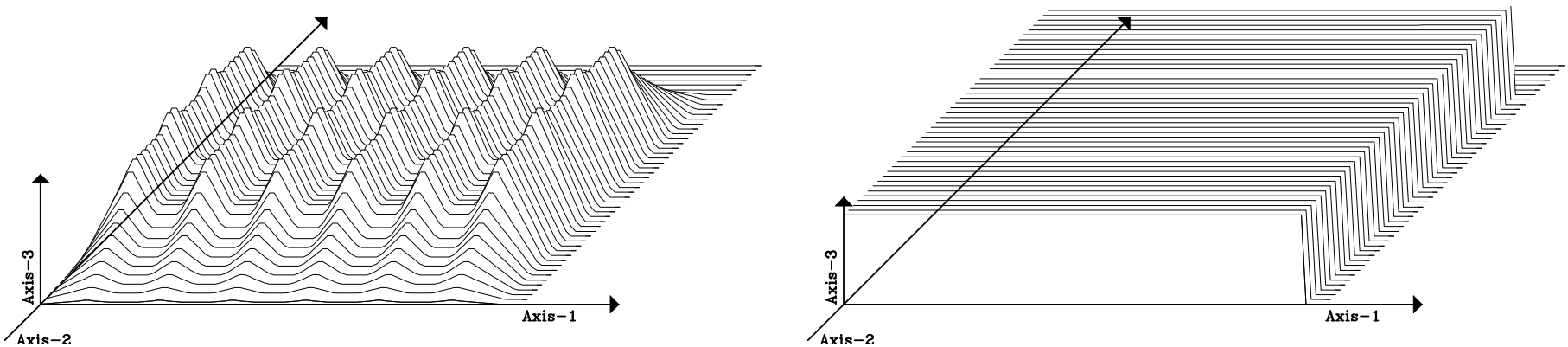


Figure 4.8: Sample output of `nonstat2()` with same parameters as Figure 4.7. Left is `weight(n1,n2)` and right is `output(n1,n2)` for constant data. The flattness of the output means that in practice we may allow window overlap greater or less than the triangle half width. uni-nstri [ER]

other. In `nonstat2()`, data is first pulled from the wall to the window. Next should be the application-specific operation on the data that processes the data window into an output window. (This output is often a residual image of a least squares procedure). To avoid getting into many application-specific details, here we simply copy the input data window to the output window. Next we devise some triangular weighting functions. These are used to weight the output window as it is copied onto the wall of accumulating weighted outputs. Simultaneously, at each point on the wall, the sum of all applied weights is accumulated. Finally, the effect of weight shape and window overlap is compensated for by dividing the value at each point on the wall of outputs by the sum of weights at that point. Figure 4.7 applies `nonstat2()` to constant data. As expected, the output is also constant, except at edges where it is zero because no windows overlap the input data. The flattness of the output means that in practice we may allow window overlap greater or less than the triangle half width. Notice that five ridges in Figure 4.7 correspond to five valleys in Figure 4.8.

In a typical application, there is one more complication. The filter outputs in each window are shorter than the input data because the filters themselves may not run over the edges else there would be truncation transients. Thus some of the

values of the output in each window are undefined. The application-specific filter program may leave these values undefined or it may set them to zero. If they come out zeros, it is safe to add them in to the wall of outputs, but care must be taken that the window weight that is normally accumulated on the wall of weights is omitted. There is one final complication for those of you who plan to be really meticulous. The triangles designed in `nonstat2()` /prog:nonstat2 taper to zero just beyond the ends of the window of *data*. They should taper to zero just beyond the ends of the window of *outputs*.

4.6. DIP PICKING WITHOUT DIP SCANNING

"**Picking**" is the process of identifying dipping seismic events. Here we will do something like picking, but in a continuum; i.e., dips will be picked continuously and set on a uniform mesh. Customarily, dip picking is done by scanning two-dimensional data along various dips. We will see that our method, based on the "plane-wave destructor operator," does not have its resolution limited by the spatial extent of a dip scan.

4.6.1. The plane-wave destructor

A plane wave in a wave field $u(t,x)=u(t-px)$ with stepout p can be extinguished with a partial differential operator, which we write as a matrix $\mathbf{A}$, where

$$0 \quad \approx \quad v(t,x) = \quad \left(\frac{\partial}{\partial x} + p_i \frac{\partial}{\partial t}\right) u(t,x) \tag{4.25}$$

$$\mathbf{0} \quad \approx \quad \mathbf{v} = \quad \mathbf{A} \quad \mathbf{u} \tag{4.26}$$

The parameter p is called the "wavenumber" or "**Snell parameter**," and $|p|$ can take on any value less than $1/v$, where v is the medium velocity. The angle of propagation of the wave is $\sin\theta = pv$.

We need a method of discretization that allows the mesh for du/dt to overlay exactly du/dx. To this end I chose to represent the t-derivative by

$$\frac{du}{dt} \quad \approx \quad \frac{1}{2}\left(\frac{u(t+\Delta t,x)-u(t,x)}{\Delta t}\right) + \frac{1}{2}\left(\frac{u(t+\Delta t,x+\Delta x)-u(t,x+\Delta x)}{\Delta t}\right) \tag{4.27}$$

and the x-derivative by an analogous expression with t and x interchanged. Now the difference operator $\delta_x + p_i\,\delta_t$ is a two-dimensional filter that fits on a 2×2 dif-

ferencing star. As a matrix operation, this two-dimensional convolution is denoted **A**. (More details about finite differencing can be found in IEI.)

The program `wavekill1()` applies the operator $a\delta_x + p\delta_t$, which can be specialized to the operators δ_x, δ_t, $\delta_x + p_i\delta_t$. wavekill1 I carefully arranged the side boundaries so that the filter never runs off the sides of the data. Thus the output is shorter than the input by one point on both the t-axis and the x-axis. The reason for using these side boundaries is that large datasets can be chopped into independent sections without the boundaries themselves affecting the result. By chopping a large dataset into sections, we can handle curved events as **piecewise linear**.

When only one wave is present and the data is adequately sampled, then finding the best value of p is a single-parameter, linear least-squares problem. Let $\mathbf{x}$ be an abstract vector whose components are values of $\partial u/\partial x$ taken on a mesh in (t,x). Likewise, let $\mathbf{t}$ contain $\partial u/\partial t$. Since we want $\mathbf{x}+p\,\mathbf{t}\approx\mathbf{0}$, we minimize the quadratic function of p,

$$Q(p) \quad = \quad (\mathbf{x}+p\,\mathbf{t})\cdot(\mathbf{x}+p\,\mathbf{t}) \tag{4.28}$$

by setting to zero the derivative. We get

$$p \quad = \quad -\frac{\mathbf{x}\cdot\mathbf{t}}{\mathbf{t}\cdot\mathbf{t}} \tag{4.29}$$

```
#   vv = (aa Dx + pp Dt) uu
#
subroutine wavekill1( aa, pp,    n1,n2,uu,  vv )
integer i1,i2,                      n1,n2
real aa, pp, s11, s12, s21, s22, uu(n1,n2), vv( n1-1,  n2-1)
s11 = -aa-pp;   s12 = aa-pp
s21 = -aa+pp;   s22 = aa+pp
call null(                                  vv,(n1-1)*(n2-1))
do i1= 1, n1-1 {                # vv is one point shorter than uu on both axes.
do i2= 1, n2-1 {
        vv(i1,i2) = vv(i1,i2) +
                uu(i1  ,i2) * s11 + uu(i1  ,i2+1) * s12 +
                uu(i1+1,i2) * s21 + uu(i1+1,i2+1) * s22
        }}
return; end
```

Back

Since data will not always fit the model very well, it may be helpful to have some way to measure how good the fit is. I suggest

$$C^2 \quad = \quad 1 - \frac{(\mathbf{x}+p\,\mathbf{t})\cdot(\mathbf{x}+p\,\mathbf{t})}{\mathbf{x}\cdot\mathbf{x}} \tag{4.30}$$

which, on inserting $p=-(\mathbf{x}\cdot\mathbf{t})/(\mathbf{t}\cdot\mathbf{t})$, leads to C, where

$$C \quad = \quad \frac{\mathbf{x}\cdot\mathbf{t}}{\sqrt{(\mathbf{x}\cdot\mathbf{x})(\mathbf{t}\cdot\mathbf{t})}} \tag{4.31}$$

is known as the "**normalized correlation**." The program for this calculation is straightforward. I named the program `puck()` to denote *picking* on a contin*uu*m.

puck

Finally and parenthetically, an undesirable feature of the plane-wave destructor method is that the residual **v** has no particular relation to the data **u**, unlike in time-series analysis—see chapter 7. Another disadvantage, well known to people who routinely work with finite-difference solutions to partial differential equations, is that for short wavelengths a difference operator is not the same as a differential operator; thereby the numerical value of p is biased.

```
# measure coherency and dip, and compute residual  res = (Dx + p Dt) uu
#
subroutine puck ( n1, n2, uu, coh, pp, res )
integer i1, i2,   n1, n2
real    uu(n1,n2), res(n1,n2), xx, xt, tt, coh, pp
temporary real dx(n1,n2-1), dt(n1-1,n2-1)
call wavekill1( 1., 0.,  n1,n2 , uu, dx)
call wavekill1( 0., 1.,  n1,n2 , uu, dt)
xx = 1.e-30; tt = 1.e-30; xt = 0.
do i1= 1, n1-1 {
do i2= 1, n2-1 {
        xt = xt + dt(i1,i2) * dx(i1,i2)
        tt = tt + dt(i1,i2) * dt(i1,i2)
        xx = xx + dx(i1,i2) * dx(i1,i2)
        }}
coh = sqrt((xt/tt)*(xt/xx))
pp  =  - xt/tt
call wavekill1( 1., pp,  n1,n2 , uu, res)
return; end
```

Back

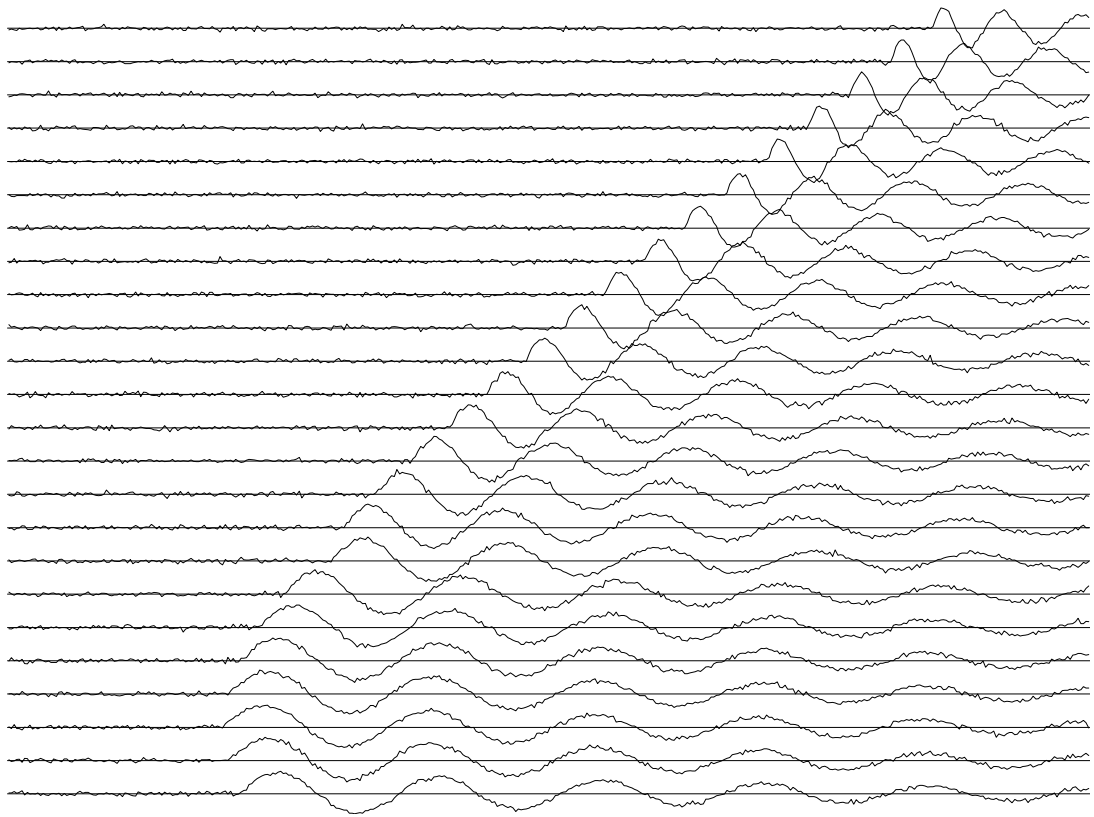
4.6.2. Moving windows for nonstationarity

Wavefronts generally curve. But a curved line viewed only over a small range is barely distinguishable from a straight line. A straight-line wavefront is much easier to manage than a curved one. If we think of the slope of the line as a parameter estimated statistically, then it is a *nonstationary* variable—it varies from place to place. So we can work with curved wavefronts by working in a small window that is moved around. The main thing to beware of about small windows is that unless we are very careful, their side boundaries may bias the result.

The `puck()` method was designed to be ignorant of side boundaries: it can be applied in a small window and the window moved freely around the data. A strength of the `puck()` method is that the window can be smaller than a wavelength—it can be merely two traces wide. A sample based on synthetic data is shown in Figures 4.9 through 4.11. The synthetic data in 4.9 mimics a reflection seismic field profile, including one trace that is slightly delayed as if recorded on a patch of unconsolidated **soil**. Notice a low level of noise in the synthetic data.

Figure 4.10 shows the **residual**. The residual is small in the central region of the data; it is large where there is **spatial alias**ing; and it is large at the transient onset of the signal. The residual is rough because of the noise in the signal,

Figure 4.9: Input synthetic data. uni-puckin [ER]



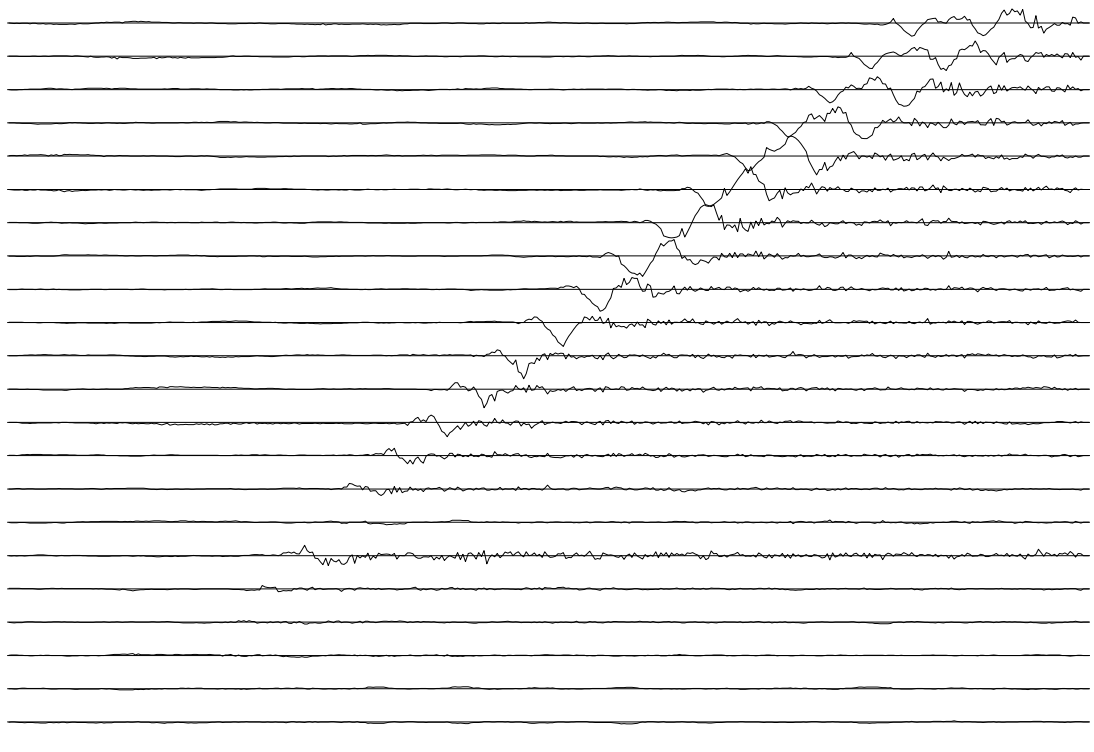


Figure 4.10: Residuals, i.e., an evaluation of $U_x + pU_t$. uni-residual [ER]

because it is made from derivatives, and because the synthetic data was made by nearest-neighbor interpolation. Notice that the residual is not particularly large for the delayed trace.

Figure 4.11 shows the dips. The most significant feature of this figure is the

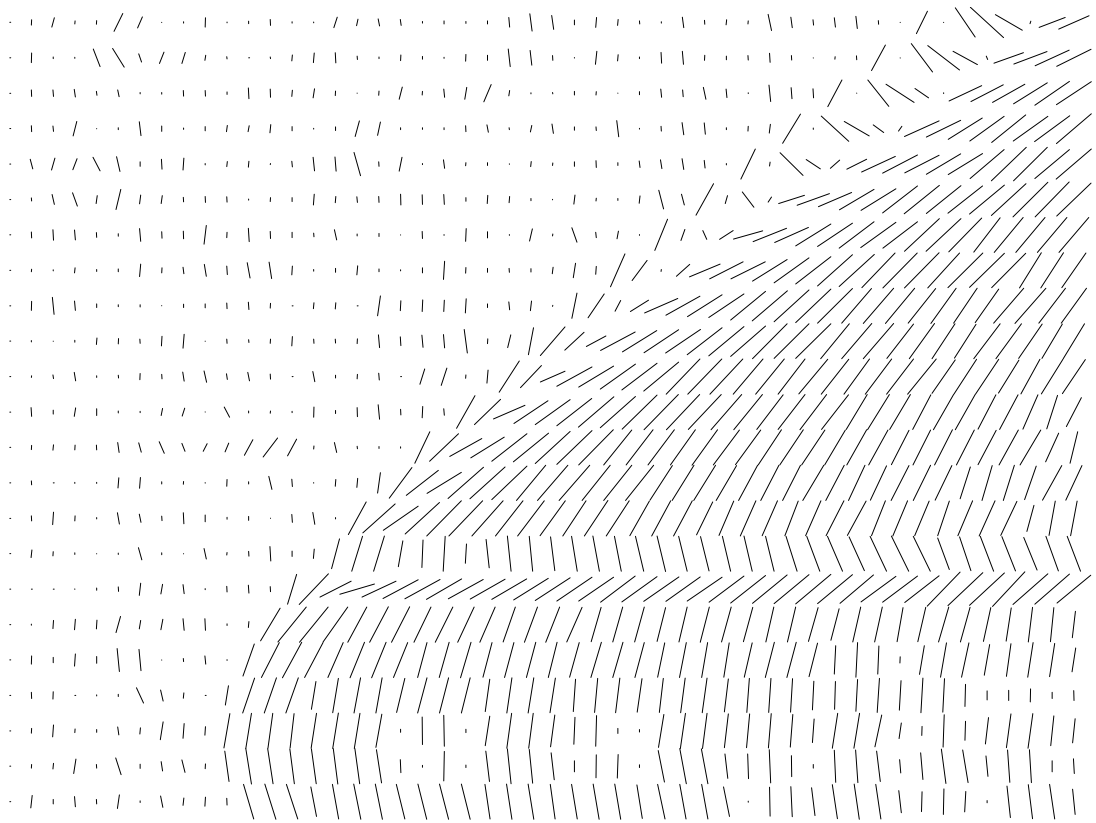


Figure 4.11: Output values of p are shown by the slope of short line segments. uni-puckout [ER]

sharp localization of the dips surrounding the delayed trace. Other methods based on wave or Fourier concepts might lead us to conclude that the aperture must be large to resolve a wide range of angles. Here we have a narrow aperture (two traces), but the dip can change rapidly and widely.

Subroutine `slider()` /prog:slider below shows the code that generated Figure 4.9 through 4.11. slider

A disadvantage of the `puck()` method is that the finite-difference operator is susceptible to spatial aliasing as well as to distortions at spatial frequencies that are high but not yet aliased. This suggests a logical step—estimating missing interlaced traces—which we take up in chapter 8.

EXERCISES:

1 It is possible to reject two dips with the operator

$$(\partial_x + p_1 \partial_t)(\partial_x + p_2 \partial_t) \tag{4.32}$$

This is equivalent to

$$\left(\frac{\partial^2}{\partial x^2} + a \frac{\partial^2}{\partial x \partial t} + b \frac{\partial^2}{\partial t^2} \right) u(t,x) \quad = \quad v(t,x) \quad \approx \quad 0 \tag{4.33}$$

```
# slide a window around on a wall of data measuring coherency, dip, residual
#
subroutine slider( n1,n2, w1,w2, k1,k2, data, coh, pp, res)
integer i1,i2,     n1,n2, w1,w2, k1,k2,                      s1,s2, e1,e2
integer p1,p2                              # number of subwindows is p1*p2
real    data(n1,n2)                        # input
real    res(n1,n2)                         # outputs. math size (n1-1,n2-1)
real    pp(n1,n2), coh(n1,n2)              # outputs defined at pp( 1..p1, 1..p2)
temporary real  count( n1,n2)
temporary real  window(w1,w2), tres(w1-1,w2-1)
call null(      count, n1*n2)
call null(        res, n1*n2)
p2=0; e2=w2; while( e2<=n2) {  p2=p2+1; s2=e2-w2+1
p1=1; e1=w1; while( e1<=n1) {  p1=p1+1; s1=e1-w1+1
        do i1 = 1, w1 {
        do i2 = 1, w2 { window(i1,i2) = data(i1+s1-1,i2+s2-1)
                        }}
        call null(                                          tres, (w1-1)*(w2-1))
        call puck ( w1, w2, window, coh(p1,p2), pp(p1,p2), tres)
        do i1= s1, e1-1 {
        do i2= s2, e2-1 {
                res(  i1,i2) =    res(i1,i2) + tres( i1-s1+1, i2-s2+1)
                count(i1,i2) =  count(i1,i2) + 1.
                }}
        e1=e1+k1              }
        e2=e2+k2              }
do i2= 1, n2-1 {
do i1= 1, n1-1 {        if( count(i1,i2) > 0. )
                                res(i1,i2) = res(i1,i2) / count(i1,i2)
                }}
return; end
```

Back

where u is the input signal and v is the output signal. Show how to solve for a and b by minimizing the energy in v.

2 Given a and b from the previous exercise, what are p_1 and p_2?

Chapter 5

Adjoint operators

A great many of the calculations we do in science and engineering are really matrix multiplication in disguise. The first goal of this chapter is to unmask the disguise by showing many examples. Second, we will illuminate the meaning of the **adjoint**

operator (matrix transpose) in these many examples.

Geophysical modeling calculations generally use linear operators that predict data from models. Our usual task is to find the inverse of these calculations, i.e., to find models (or make maps) from the data. Logically, the adjoint is the first step and a part of all subsequent steps in this **inversion** process. Surprisingly, in practice the adjoint sometimes does a better job than the inverse! This is because the adjoint operator tolerates imperfections in the data and does not demand that the data provide full information.

Using the methods of this chapter, you will find that once you grasp the relationship between operators in general and their adjoints, you can have the adjoint just as soon as you have learned how to code the modeling operator.

If you will permit me a poet's license with words, I will offer you the following table of **operator**s and their **adjoint**s:

matrix multiply	conjugate-transpose matrix multiply
convolution	crosscorrelation
stretching	squeezing
zero padding	truncation
causal integration	anticausal integration

add functions	do integrals
plane-wave superposition	slant stack / beam forming
superposing on a curve	summing along a curve
upward continuation	downward continuation
diffraction modeling	imaging by migration
hyperbola modeling	CDP stacking
ray tracing	**tomography**

The left column above is often called “**modeling**," and the adjoint operators on the right are often used in “data **processing**."

When the adjoint operator is *not* an adequate approximation to the inverse, then you apply the techniques of fitting and optimization which require iterative use of the modeling operator and its adjoint.

The adjoint operator is sometimes called the “**back projection**" operator because information propagated in one direction (earth to data) is projected backward (data to earth model). With complex-valued operators the transpose and complex conjugate go together and in Fourier analysis, taking the complex conjugate of $\exp(i\omega t)$ reverses the sense of time. Still assuming poetic license, I will say that

adjoint operators *undo* the time and phase shifts of modeling operators. The inverse operator does this too, but it also divides out the color. For example, with linear interpolation high frequencies are smoothed out, so inverse interpolation must restore them. You can imagine the possibilities for noise amplification. That is why adjoints are safer than inverses. But nature determines in each application what is the best operator to use, whether to stop after the adjoint, to go the whole way to the inverse, or to stop part-way.

We will see that computation of the **adjoint** is a straightforward adjunct to the computation itself, and the computed adjoint should be, and generally can be, exact (within machine precision). If the application's operator is computed in an approximate way, we will see that it is natural and best to compute the adjoint with adjoint approximations. Much later in this chapter is a formal definition of adjoint operator. Throughout the chapter we handle an adjoint operator as a matrix transpose, but we hardly ever write down any matrices or their transposes. Instead, we always prepare two subroutines, one that performs $\mathbf{y} = \mathbf{A}\mathbf{x}$ and another that performs $\tilde{\mathbf{x}} = \mathbf{A}'\mathbf{y}$, so we need a test that the two subroutines really embody the essential aspects of matrix transposition. Although the test is an elegant and useful test and is itself a fundamental definition, curiously, that definition helps us not one

bit in constructing adjoint operators, so I postpone the formal definition of adjoint until after we have seen many examples.

5.1. FAMILIAR OPERATORS

The operation $y_i = \sum_j b_{ij} x_j$ is multiplication of a matrix **B** times a vector **x**. The adjoint operation is $\tilde{x}_j = \sum_i b_{ij} y_i$. The operation adjoint to multiplying by a matrix is multiplying by the transposed matrix (unless the matrix has complex elements, in which case we need the complex-conjugated transpose). The following **pseudocode** does matrix multiplication $\mathbf{y} = \mathbf{B}\mathbf{x}$ and multiplication by the transpose matrix $\tilde{\mathbf{x}} = \mathbf{B}'\mathbf{y}$:

```
if operator itself
        then erase y
if adjoint
        then erase x
do iy = 1, ny {
do ix = 1, nx {
        if operator itself
                y(iy) = y(iy) + b(iy,ix) × x(ix)
        if adjoint
                x(ix) = x(ix) + b(iy,ix) × y(iy)
}}
```

Notice that the "bottom line" in the program is that *x* and *y* are simply interchanged. The above example is a prototype of many to follow, so observe carefully the similarities and differences between the operation and its adjoint.

A formal program for matrix multiply and its adjoint is found below. The first step is erasing the output. That may seem like too trivial a function to put in a separate library routine, but, at last count, 15 other routines in this book use the

```
subroutine adjnull( adj, add, x, nx,  y, ny )
integer ix, iy,     adj, add,    nx,     ny
real                             x( nx), y( ny )
if( add == 0 )
        if( adj == 0 )
                do iy= 1, ny
                        y(iy) = 0.
        else
                do ix= 1, nx
                        x(ix) = 0.
return; end
```

Back

```
# matrix multiply and its adjoint
#
subroutine matmult( adj, bb,         nx,x,  ny,y)
integer ix, iy,    adj,             nx,    ny
real                    bb(ny,nx), x(nx), y(ny)
call adjnull(      adj, 0,          x,nx,  y,ny)
do ix= 1, nx {
do iy= 1, ny {
        if( adj == 0 )
                        y(iy) = y(iy) + bb(iy,ix) * x(ix)
        else
                        x(ix) = x(ix) + bb(iy,ix) * y(iy)
        }}
return; end
```

Back

output-erasing subroutine `adjnull()` below. adjnull The subroutine `matmult()` /prog:matmult for matrix multiply and its adjoint exhibits a style that we will use repeatedly. matmult

5.1.1. Transient convolution

When the matrix has a special form, such as

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} b_1 & 0 & 0 & 0 & 0 \\ b_2 & b_1 & 0 & 0 & 0 \\ b_3 & b_2 & b_1 & 0 & 0 \\ 0 & b_3 & b_2 & b_1 & 0 \\ 0 & 0 & b_3 & b_2 & b_1 \\ 0 & 0 & 0 & b_3 & b_2 \\ 0 & 0 & 0 & 0 & b_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \tag{5.1}$$

then the matrix multiplication and transpose multiplication still fit easily in the same computational framework. The operation **Bx** convolves b_t with x_t, whereas the operation $\mathbf{B}'\mathbf{y}$ **crosscorrelates** b_t with y_t. I will leave it to you to verify the pseudocode

```
do ix = 1, nx {
do ib = 1, nb {
        iy = ib + ix – 1
        if operator itself (convolution)
                y(iy) = y(iy) + b(ib) × x(ix)
        if adjoint (correlation)
                x(ix) = x(ix) + b(ib) × y(iy)
}}
```

Again, notice that the “bottom line” in the program is that *x* and *y* are simply interchanged.

Equation (5.1) could be rewritten as

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} x_1 & 0 & 0 \\ x_2 & x_1 & 0 \\ x_3 & x_2 & x_1 \\ x_4 & x_3 & x_2 \\ x_5 & x_4 & x_3 \\ 0 & x_5 & x_4 \\ 0 & 0 & x_5 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{5.2}$$

which we abbreviate by $\mathbf{y} = \mathbf{Xb}$. So we can choose between $\mathbf{y} = \mathbf{Xb}$ and $\mathbf{y} = \mathbf{Bx}$. In one case the output **y** is dual to the filter **b**, and in the other case the output **y** is dual to the input **x**. In applications, sometimes we will solve for **b** and sometimes for **x**; so sometimes we will use equation (5.2) and sometimes (5.1).

The program `contran()` /prog:contran can be used with either equation (5.1) or equation (5.2), because the calling program can swap the `xx` and `bb` variables. The name `contran()` denotes convolution with "transpose" and with "transient" end effects. contran

```
#       Convolve and correlate (adjoint convolve).
#
subroutine contran( adj, add, nx, xx, nb, bb, yy)
integer ix, ib, ny, adj, add, nx,     nb
real    xx(nx)                  # input signal
real    bb(nb)                  # filter      (or output crosscorrelation)
real    yy(nx+nb-1)             # filtered signal (or second input signal)
ny = nx + nb - 1                # length of filtered signal
call adjnull(       adj, add,         bb, nb, yy, ny)
do ib= 1, nb {
do ix= 1, nx {
        if( adj == 0 )
                yy( ib+ix-1) = yy( ib+ix-1) + xx( ix) * bb(ib)
        else
                bb( ib)      = bb( ib)      + xx( ix) * yy( ib+ix-1)
        }}
return; end
```

Back

5.1.2. Zero padding is the transpose of truncation.

Surrounding a dataset by zeros (**zero pad**ding) is adjoint to throwing away the extended data (**truncation**). Let us see why this is so. Set a signal in a vector **x**, and then make a longer vector **y** by adding some zeros at the end of **x**. This zero padding can be regarded as the matrix multiplication

$$\mathbf{y} \quad = \quad \begin{bmatrix} \mathbf{I} \\ \mathbf{0} \end{bmatrix} \mathbf{x} \tag{5.3}$$

The matrix is simply an identity matrix **I** above a zero matrix **0**. To find the transpose to zero padding, we now transpose the matrix and do another matrix multiply:

$$\tilde{\mathbf{x}} \quad = \quad \begin{bmatrix} \mathbf{I} & \mathbf{0} \end{bmatrix} \mathbf{y} \tag{5.4}$$

So the transpose operation to zero padding data is simply *truncating* the data back to its original length.

5.1.3. Product of operators

We will look into details of Fourier transformation elsewhere. Here we use it as an example of any operator containing complex numbers. For now, we can think of

Fourier transform as a square matrix $\mathbf{F}$. We denote the complex-conjugate transpose (or **adjoint**) matrix with a prime, i.e., $\mathbf{F}'$. The adjoint arises naturally whenever we consider energy. The statement that Fourier transforms conserve energy is $\mathbf{y}'\mathbf{y} = \mathbf{x}'\mathbf{x}$ where $\mathbf{y} = \mathbf{F}\mathbf{x}$. Substituting gives $\mathbf{F}'\mathbf{F} = \mathbf{I}$ which shows that the inverse matrix to Fourier transform happens to be the complex conjugate of the transpose of $\mathbf{F}$.

With Fourier transforms, **zero pad**ding and **truncation** are particularly prevalent. Most programs transform a dataset of length of 2^n, whereas dataset lengths are often of length $m \times 100$. The practical approach is therefore to pad given data with zeros. Padding followed by Fourier transformation $\mathbf{F}$ can be expressed in matrix algebra as

$$\text{Program} \quad = \quad \mathbf{F} \begin{bmatrix} \mathbf{I} \\ \mathbf{0} \end{bmatrix} \tag{5.5}$$

According to matrix algebra, the transpose of a product, say $\mathbf{AB} = \mathbf{C}$, is the product $\mathbf{C}' = \mathbf{B}'\mathbf{A}'$ in reverse order. So the adjoint program is given by

$$\text{Program}' \quad = \quad \begin{bmatrix} \mathbf{I} & \mathbf{0} \end{bmatrix} \mathbf{F}' \tag{5.6}$$

Thus the adjoint program *truncates* the data *after* the inverse Fourier transform.

```
# signal advance:    y(iy) = x(iy+jump)
#
subroutine advance( adj, add, jump, nx, xx, ny, yy)
integer ix, iy,     adj, add, jump, nx,     ny
real                                        xx(nx), yy(ny)
call adjnull( adj, add,                     xx,nx,  yy,ny)
do iy= 1, ny {
        ix = iy + jump
        if(          ix >= 1 )
                if( ix <= nx )
                        if( adj == 0 )
                                yy( iy) = yy( iy) + xx( ix)
                        else
                                xx( ix) = xx( ix) + yy( iy)
        }
return; end
```

Back

5.1.4. Convolution end effects

In practice, filtering generally consists of three parts: (1) **convolution**, (2) **shifting** to some preferred time alignment, and (3) **truncating** so the output has the same length as the input. An adjoint program for this task, is easily built from an earlier program. We first make a simple time-shift program `advance()`. advance Although the code is bulky for such a trivial program, it is easy to read, works for any size of array, and works whether the shift is positive or negative. Since filtering ordinarily delays, the `advance()` routine generally compensates.

Merging `advance()` with the earlier program `contran()` according to the transpose rule $(\mathbf{AB})' = \mathbf{B}'\mathbf{A}'$, we get `contrunc()`. contrunc For a symmetrical filter, a `lag` parameter half of the filter length would be specified. The output of a minimum-phase filter is defined to be at the beginning of the filter, `ff(1)`, so then `lag=1`. The need for an adjoint filtering program will be apparent later, when we design filters for prediction and interpolation. The program variable `add` happens to be useful when there are many signals. Our first real use of `add` will be found in the subroutine `stack1()` /prog:stack1.

Another goal of convolution programs is that zero data not be assumed beyond the interval for which the data is given. This can be important in filter design and

```
#       Convolve, shift, and truncate output.
#
subroutine contrunc( conj, add, lag,  np,pp,  nf,ff,  nq,qq)
integer ns,          conj, add, lag,  np,     nf,     nq
real    pp(np)                          # input data
real    ff(nf)                          # filter (output at ff(lag))
real    qq(nq)                          # filtered data
temporary real ss( np+nf-1)
ns = np + nf - 1
if( conj == 0 ) {
        call contran( 0, 0, np,pp, nf,ff, ss)
                                call advance( 0, add, lag-1, ns,ss, nq,qq)
        }
else {                          call advance( 1,   0, lag-1, ns,ss, nq,qq)
        call contran( 1, add, np,pp, nf,ff, ss)
        }
return; end
```

Back

```
#       Convolve and correlate with no assumptions off end of data.
#
subroutine convin( adj, add, nx, xx, nb, bb, yy)
integer ib, iy,ny, adj, add, nx,     nb
real    xx(nx)                  # input signal
real    bb(nb)                  # filter      (or output crosscorrelation)
real    yy(nx-nb+1)             # filtered signal (or second input signal)
ny = nx - nb + 1                # length of filtered signal
if( ny < 1 )  call erexit('convin() filter output negative length.')
call adjnull(      adj, add,          bb, nb, yy, ny)
if( adj == 0 )
        do iy= 1, ny {
        do ib= 1, nb {
                        yy( iy) = yy( iy) + bb(ib) * xx( iy-ib+nb)
                        }}
else
        do ib= 1, nb {
        do iy= 1, ny {
                        bb( ib) = bb( ib) + yy(iy) * xx( iy-ib+nb)
                        }}
return; end
```

Back

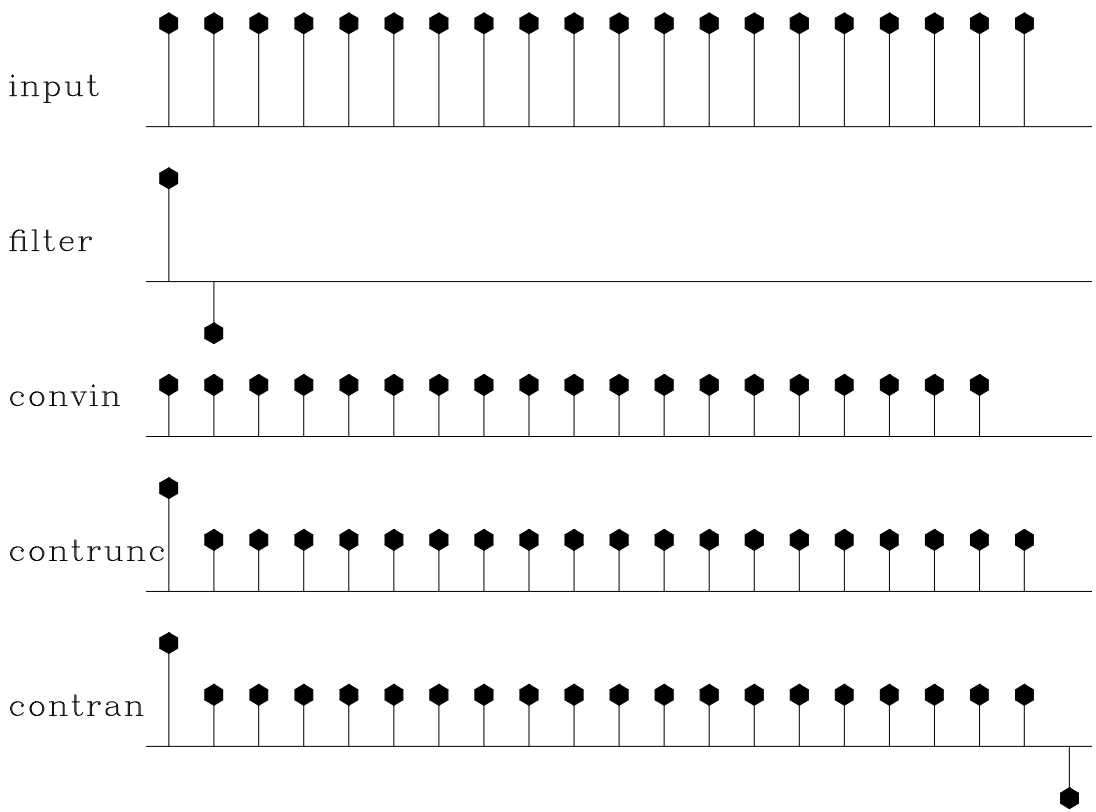


Figure 5.1: Example of convolution end effects. From top to bottom: (1) input; (2) filter; (3) output of `convin()`; (4) output of `contrunc()` with no lag (`lag=1`); and (5) output of `contran()`. conj-conv [ER]

spectral estimation, when we do not want the **truncation** at the end of the data to have an effect. Thus the output data is shorter than the input signal. To meet this goal, I prepared subroutine `convin()`. convin By now you are probably tired of looking at so many variations on convolution; but convolution is the computational equivalent of ordinary differential equations, its applications are vast, and end effects are important. The **end effect**s of the convolution programs are summarized in Figure 5.1.

5.1.5. Kirchhoff modeling and migration

Components of a vector can be summed into a scalar. The adjoint is taking the scalar and distributing it out to a vector (also called "**scatter**ing" or "**spray**ing"). Alternately, values to be summed can come from a trajectory in a plane, such as a hyperbolic trajectory.

When reflectors in the earth are dipping, or broken into point scatterers, time-to-depth conversion is not simply a stretching of the time axis. Modeling is done in a variety of ways, one of which is to model each point in the depth (x, z)-plane by a hyperbola in the data (x, t)-plane. The adjoint operation consumes much com-

puter power in the petroleum-prospecting industry and is called “migration." Many migration methods exist, most of which are taken up in IEI, but that book does not describe the adjoint property I discuss below.

Hyperbola superposition is the adjoint to hyperbola recognition by summing along hyperbolas. The summing is called “**Kirchhoff migration**” or “**imaging**,” and the spraying is called “Kirchhoff modeling." The name comes from Kirchhoff’s diffraction integral.

In the pseudocode below, the parameter `ih` refers to the separation of a point on a hyperbola from its top at `ix`. Ignoring “if index off data” tests, I show Kirchhoff modeling and migration in the pseudocode following:

```
do iz = 1,nz
    do ix = 1,nx
        do ih = –25, 25
            it = sqrt( iz∗iz + ih∗ih )/velocity
            ig = ix + ih
            if not adjoint
                zz(iz,ix) = zz(iz,ix) + tt(it,ig) # imaging
            if adjoint
                tt(it,ig) = tt(it,ig) + zz(iz,ix) # modeling
```

We can speed up the program by moving the `ix` loop to the inside of the square root and interpolation overheads.

5.1.6. Migration defined

"**Migration**" is a word in widespread use in reflection seismology to define any data-processing program that converts a data plane to an image. IEI offers several descriptive definitions of migration. Here I offer you a mathematical definition of

a migration operator: given any (**diffraction**) **modeling** operator $\mathbf{B}$, its adjoint $\mathbf{B}'$ defines a migration operator. This raises the interesting question, what is the inverse to $\mathbf{B}$, and how does it differ from the adjoint $\mathbf{B}'$?

An adjoint operator is not the same as an inverse operator. Most people think of migration as the *inverse* of modeling, but mathematically it is the *adjoint* of modeling. In many wave-propagation problems, $\mathbf{B}^{-1}$ and $\mathbf{B}'$ are nearly the same. A formula for $\mathbf{B}^{-1}$ (from (5.14)) is $\mathbf{B}^{-1} = (\mathbf{B}'\mathbf{B})^{-1}\mathbf{B}'$. So the difference between $\mathbf{B}'$ and $\mathbf{B}^{-1}$ is in the factor $\mathbf{B}'\mathbf{B}$. Theoreticians that work in the continuum find something like $\mathbf{B}'\mathbf{B}$ in the form of a weighting function in the physical domain or a weighting function in the spectral domain or both. Since it is merely a weighting function, it is not very exciting to practitioners who are accustomed to weighting functions in both domains for other purposes, principally for enhancing data display. Indeed, it could be a pitfall to introduce the weighting function of inversion, because it could interfere with the data display. The opportunity that I see for inversion lies in practice where $\mathbf{B}'\mathbf{B}$ is quite far from an identity matrix for another reason—that data is not a continuum and has aliasing, truncation, and noise.

A curious aspect of migration arises from the reflection **amplitude versus offset** (**AVO**) along the hyperbola. The effect of changing AVO is to change the dip

filtering. Notice that effort expended to get the correct AVO in the modeling operator affects the migration operator (the adjoint) without necessarily making it closer to the inverse. It is a **pitfall** to imagine that carefully constructing the correct amplitude versus offset in a diffraction operator will make the corresponding migration operator more effective. The question of whether an inverse operator is better than an adjoint has no simple answer; its answer depends on circumstances. So the phrase "true amplitude migration" has questionable meaning.

You might look at the Kirchhoff migration code above and ask, what is the modelling matrix that is transposed? We don't see it. We started by defining "adjoint operator" as the transpose of a matrix, but now we seem to be defining it by a certain programming style. The abstract vector for Kirchhoff migration is packed with data values from a two-dimensional (t, x)-plane. The abstract matrix is hard to visualize. How can we know whether we have defined the adjoint operator correctly? The answer is given next by the dot-product test.

5.2. ADJOINT DEFINED: DOT-PRODUCT TEST

There is a huge gap between the conception of an idea and putting it into practice. During development, things fail far more often than not. Often, when something fails, many tests are needed to track down the cause of failure. Maybe the cause cannot even be found. More insidiously, failure may be below the threshold of detection and poor performance suffered for years. I find the **dot-product test** to be an extremely valuable checkpoint.

Conceptually, the idea of matrix transposition is simply $a'_{ij} = a_{ji}$. In practice, however, we often encounter matrices far too large to fit in the memory of any computer. Sometimes it is also not obvious how to formulate the process at hand as a matrix multiplication. What we find in practice is that an application and its adjoint amounts to two subroutines. The first subroutine amounts to the matrix multiplication $\mathbf{Ax}$. The adjoint subroutine computes $\mathbf{A}'\mathbf{y}$, where $\mathbf{A}'$ is the transpose matrix. In a later chapter we will be solving huge sets of simultaneous equations. Then both subroutines are required. We are doomed from the start if the practitioner provides an inconsistent pair of subroutines. The dot product test is a simple test for verifying that the two subroutines are adjoint to each other.

The associative property of linear algebra says that we do not need parenthe-

ses in a vector-matrix-vector product like $\mathbf{y}'\mathbf{A}\mathbf{x}$ because we get the same result no matter where we put the parentheses. They serve only to determine the sequence of computation. Thus,

$$\mathbf{y}'(\mathbf{A}\mathbf{x}) = (\mathbf{y}'\mathbf{A})\mathbf{x} \tag{5.7}$$

$$\mathbf{y}'(\mathbf{A}\mathbf{x}) = (\mathbf{A}'\mathbf{y})'\mathbf{x} \tag{5.8}$$

(In general, the matrix is not square.) For the dot-product test, load the vectors $\mathbf{x}$ and $\mathbf{y}$ with random numbers. Compute the vector $\tilde{\mathbf{y}} = \mathbf{A}\mathbf{x}$ using your program for $\mathbf{A}$, and compute $\tilde{\mathbf{x}} = \mathbf{A}'\mathbf{y}$ using your program for $\mathbf{A}'$. Inserting these into equation (5.8) gives you two scalars that should be equal.

$$\mathbf{y}'(\mathbf{A}\mathbf{x}) = \mathbf{y}'\tilde{\mathbf{y}} = \tilde{\mathbf{x}}'\mathbf{x} = (\mathbf{A}'\mathbf{y})'\mathbf{x} \tag{5.9}$$

The left and right sides of this equation will be computationally equal only if the program doing $\mathbf{A}'$ is indeed adjoint to the program doing $\mathbf{A}$ (unless the random numbers do something miraculous).

I tested (5.9) on many operators and was surprised and delighted to find that it is often satisfied to an accuracy near the computing precision. More amazing is that on some computers, equation (5.9) was sometimes satisfied down to and including *the*

least significant bit. I do not doubt that larger rounding errors could occur, but so far, every time I encountered a relative discrepancy of 10^{-5} or more, I was later able to uncover a conceptual or programming error. Naturally, when I do dot-product tests, I scale the implied matrix to a small dimension in order to speed things along, and to be sure that boundaries are not overwhelmed by the much larger interior.

Do not be alarmed if the operator you have defined has truncation errors. Such errors in the definition of the original operator should be identically matched by **truncation** errors in the adjoint. If your code passes the **dot-product test**, then you really have coded the adjoint operator. In that case, you can take advantage of the standard methods of mathematics to obtain inverse operators.

We can speak of a continuous function $f(t)$ or a discrete one f_t. For continuous functions we use integration, and for discrete ones we use summation. In formal mathematics the dot-product test *defines* the adjoint operator, except that the summation in the dot product may need to be changed to an integral. The input or the output or both can be given either on a continuum or in a discrete domain. So the dot-product test $\mathbf{y}'\tilde{\mathbf{y}} = \tilde{\mathbf{x}}'\mathbf{x}$ could have an integration on one side of the equal sign and a summation on the other. Linear-operator theory is rich with concepts, but I will not develop it here. I assume that you studied it before you came to read this

book, and that it is my job to show you how to use it.

5.2.1. What is an adjoint operator?

In mathematics the word "**adjoint**" has three meanings. One of them, the so-called Hilbert adjoint, is the one generally found in Physics and Engineering and it is the one used in this book. In Linear Algebra is a different matrix, called the **adjugate** matrix. It is a matrix whose elements are signed cofactors (minor determinants). For invertible matrices, this matrix is the determinant times the inverse matrix. It is computable without ever using division, so potentially the adjugate can be useful in applications where an inverse matrix cannot. Unfortunately, the adjugate matrix is sometimes called the adjoint matrix particularly in the older literature. Because of the confusion of multiple meanings of the word adjoint, in the first printing of this book I avoided the use of the word, substituting the definition, "conjugate transpose". Unfortunately this was often abbreviated to "conjugate" which caused even more confusion.

EXERCISES:

1 Suppose a linear operator **A** has its input in the discrete domain and its output in the continuum. How does the operator resemble a matrix? Describe the operator $\mathbf{A}'$ which has its output in the discrete domain and its input in the continuum. To which do you apply the words "scales and adds some functions," and to which do you apply the words "does a bunch of integrals"? What are the integrands?

2 Examine the end effects in the programs `contran()` and `convin()`. Interpret differences in the adjoints.

3 An operator is "**self-adjoint**" if it equals its adjoint. Only square matrices can be **self-adjoint**. Prove by a numerical test that subroutine `leaky()` /prog:leaky is **self-adjoint**.

4 Prove by a numerical test that the subroutine `triangle()` /prog:triangle, which convolves with a **triangle** and then folds boundary values back inward, is **self-adjoint**.

5.3. NORMAL MOVEOUT AND OTHER MAPPINGS

Many times we simply deform or stretch a wave field or a map. A curious mapping I once made was a transformation of world topography (including ocean depth). **Great circles** play an important role in global surface-wave propagation because waves travel on the great circles. In my transformed map, the great circle from Stanford University to the east is plotted as an equator on a Mercator projection. North at Stanford is plotted vertically as usual. Figure 5.2 shows it.

Deformations can either stretch or shrink or both, and different practical problems arise in each of these cases.

5.3.1. Nearest-neighbor interpolation

Deformations begin from the task of selecting a value `val` from an array `vec(ix), ix=1,nx`. The points of the array are at locations `x = x0+dx*(ix-1)`. Given the location `x` of the desired value we backsolve for `ix`. In Fortran, conversion of a real value to an integer is done by truncating the fractional part of the real value.

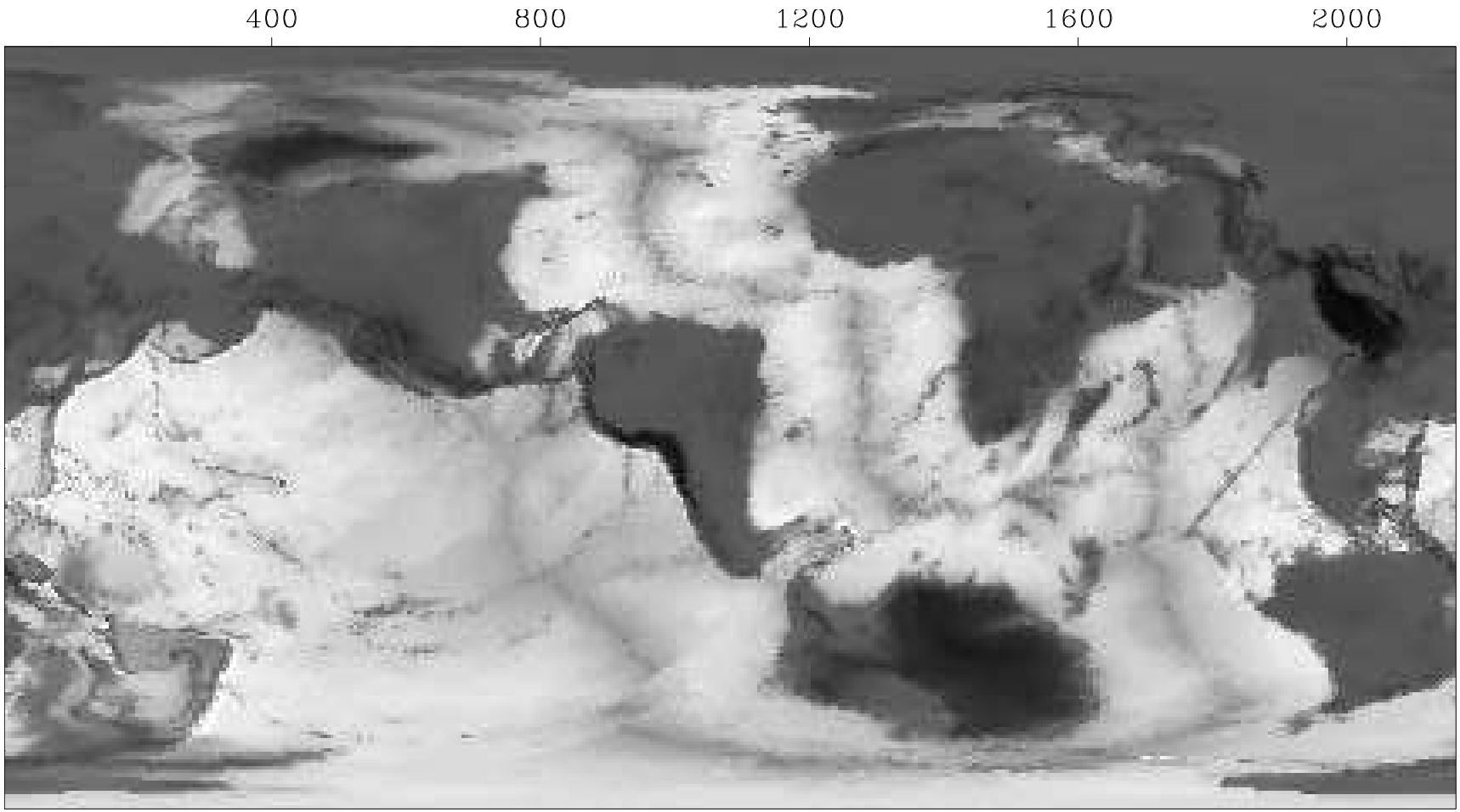


Figure 5.2: The world as Gerhardus Mercator might have drawn it if he had lived at Stanford University. Press button for movie (and be patient). conj-great [NR,M]

```
# Nearest neighbor interpolation, essentially:   val = vec( 1.5 + (t-t0)/dt)
#
subroutine spot0( adj, add, nt,t0,dt, t, val,    vec    )
integer it,       adj, add, nt
real                           t0,dt, t, val,    vec( nt)
call adjnull(     adj, add,                 val, 1, vec, nt)
it = 1.5 + (t-t0) / dt
if( 0 < it && it <= nt)
        if( adj == 0 )                                  # add value onto vector
                vec( it) = vec( it) + val
        else                                            # take value from vector
                val      = val      + vec( it)
return; end
```

Back

To get rounding up as well as down, we add a half before conversion to an integer, namely `ix=int(1.5+(x-x0)/dx)`. This gives the nearest neighbor. The adjoint to extracting a value from a vector is putting it back. A convenient subroutine for nearest-neighbor interpolation is `spot0()`. spot0

Recall subroutine `advance()` /prog:advance. For `jump==0` its matrix equivalent is an identity matrix. For other values of `jump`, the identity matrix has its diagonal shifted up or down. Now examine subroutine `spot0()` /prog:spot0 and think about its matrix equivalent. Since its input is a single value and its output is a vector, that means its matrix is a column vector so the adjoint operator is a row vector. The vector is all zeros except for somewhere where there is a "1".

5.3.2. A family of nearest-neighbor interpolations

Let an integer k range along a survey line, and let data values x_k be packed into a vector $\mathbf{x}$. (Each data point x_k could also be a seismogram.) We plan to resample the data more densely, say from 4 to 6 points. For illustration, I follow a crude **nearest-neighbor interpolation** scheme by sprinkling ones along the diagonal of a

rectangular matrix that is

$$\mathbf{y} \quad = \quad \mathbf{B}\mathbf{x} \tag{5.10}$$

where

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \tag{5.11}$$

The interpolated data is simply $\mathbf{y} = (x_1, x_2, x_2, x_3, x_4, x_4)$. The matrix multiplication (5.11) would not be done in practice. Instead there would be a loop running over the space of the outputs **y** that picked up values from the input.

- **Looping over input space**

The obvious way to program a deformation is to take each point from the *input* space and find where it goes on the output space. Naturally, many points could land in the same place, and then only the last would be seen. Alternately, we could first erase the output space, then add in points, and finally divide by the number of points

that ended up in each place. The biggest aggravation is that some places could end up with no points. This happens where the transformation **stretch**es. There we need to decide whether to interpolate the missing points, or simply low-pass filter the output.

- **Looping over output space**

The alternate method that is usually preferable to looping over input space is that our program have a loop over the space of the *outputs*, and that each output find its input. The matrix multiply of (5.11) can be interpreted this way. Where the transformation **shrink**s is a small problem. In that area many points in the input space are ignored, where perhaps they should somehow be averaged with their neighbors. This is not a serious problem unless we are contemplating iterative transformations back and forth between the spaces.

We will now address interesting questions about the reversibility of these deformation transforms.

5.3.3. Formal inversion

We have thought of equation (5.10) as a formula for finding **y** from **x**. Now consider the opposite problem, finding **x** from **y**. Begin by multiplying equation (5.11) by the **transpose matrix** to define a new quantity $\tilde{\mathbf{x}}$:

$$\begin{bmatrix} \tilde{x}_1 \\ \tilde{x}_2 \\ \tilde{x}_3 \\ \tilde{x}_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} \tag{5.12}$$

Obviously, $\tilde{\mathbf{x}}$ is not the same as **x**, but at least these two vectors have the same dimensionality. This turns out to be the first step in the process of finding **x** from **y**. Formally, the problem is

$$\mathbf{y} = \mathbf{B}\mathbf{x} \tag{5.13}$$

And the formal solution to the problem is

$$\mathbf{x} = (\mathbf{B}'\mathbf{B})^{-1}\mathbf{B}'\mathbf{y} \tag{5.14}$$

Formally, we verify this solution by substituting (5.13) into (5.14).

$$\mathbf{x} \quad = \quad (\mathbf{B}'\mathbf{B})^{-1}(\mathbf{B}'\mathbf{B})\mathbf{x} \quad = \quad \mathbf{I}\mathbf{x} \quad = \quad \mathbf{x} \tag{5.15}$$

In applications, the possible nonexistance of an inverse for the matrix $(\mathbf{B}'\mathbf{B})$ is always a topic for discussion. For now we simply examine this matrix for the interpolation problem. We see that it is diagonal:

$$\mathbf{B}'\mathbf{B} \quad = \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad = \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix} \tag{5.16}$$

So, $\tilde{\mathbf{x}}_1 = \mathbf{x}_1$; but $\tilde{\mathbf{x}}_2 = 2\mathbf{x}_2$. To recover the original data, we need to divide $\tilde{\mathbf{x}}$ by the diagonal matrix $\mathbf{B}'\mathbf{B}$. Thus, matrix inversion is easy here.

Equation (5.14) has an illustrious reputation, which arises in the context of "least squares." **Least squares** is a general method for solving sets of equations that have more equations than unknowns.

Recovering **x** from **y** using equation (5.14) presumes the existence of the inverse of $\mathbf{B}'\mathbf{B}$. As you might expect, this matrix is nonsingular when **B** *stretches* the data, because then a few data values are distributed among a greater number of locations. Where the transformation *squeezes* the data, $\mathbf{B}'\mathbf{B}$ must become singular, since returning uniquely to the uncompressed condition is impossible.

We can now understand why an adjoint operator is often an approximate inverse. This equivalency happens in proportion to the nearness of the matrix $\mathbf{B}'\mathbf{B}$ to an identity matrix. The interpolation example we have just examined is one in which $\mathbf{B}'\mathbf{B}$ differs from an identity matrix merely by a scaling.

5.3.4. Nearest-neighbor NMO

Normal-moveout correction (**NMO**) is a geometrical correction of reflection data that stretches the time axis so that data recorded at nonzero separation x_0 of shot and receiver, after stretching, appears to be at $x_0 = 0$. See Figure 5.3. NMO correction is roughly like time-to-depth conversion with the equation $v^2t^2 = z^2 + x_0^2$. After the data at x_0 is stretched from t to z, it should look like stretched data from any other x (assuming plane horizontal reflectors, etc.). In practice, z is not used; rather,

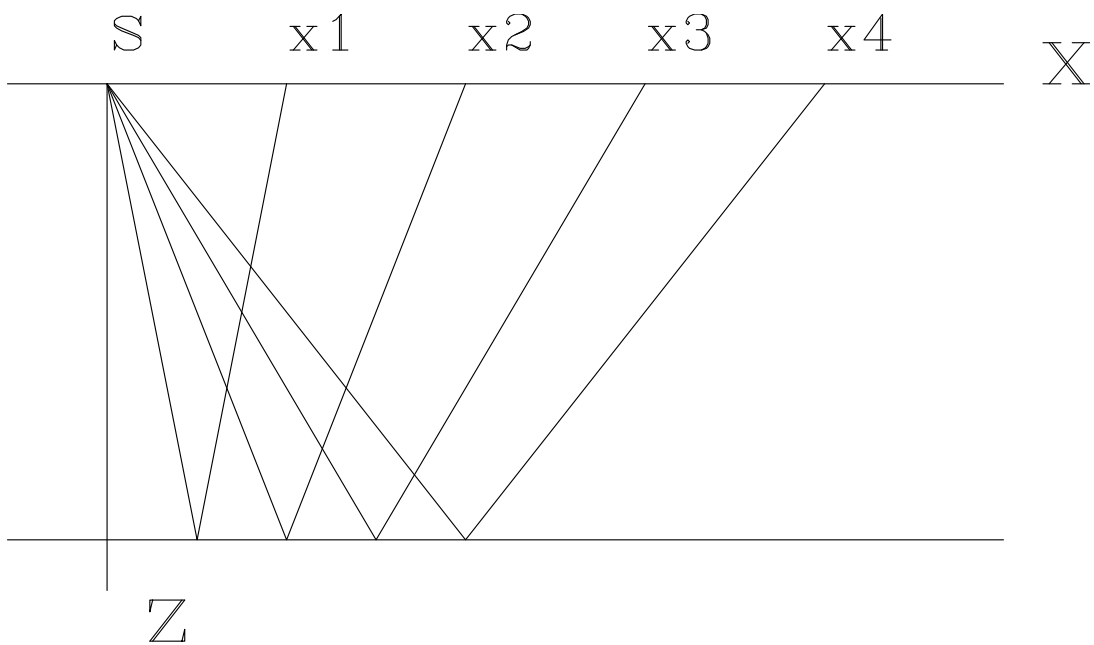


Figure 5.3: A sound emitter at location s on the earth's surface $z = 0$, and rays from a horizontal reflector at depth z reflecting back to surface locations x_i. conj-geometry [ER]

traveltime depth τ is used, where $\tau = z/v$; so $t^2 = \tau^2 + x_0^2/v^2$.

To show how surfaces deform under moveout correction, I took a square of text and deformed it according to the NMO correction equation and its inverse. This is shown in Figure 5.4. The figure assumes a velocity of unity, so the asymptotes of the

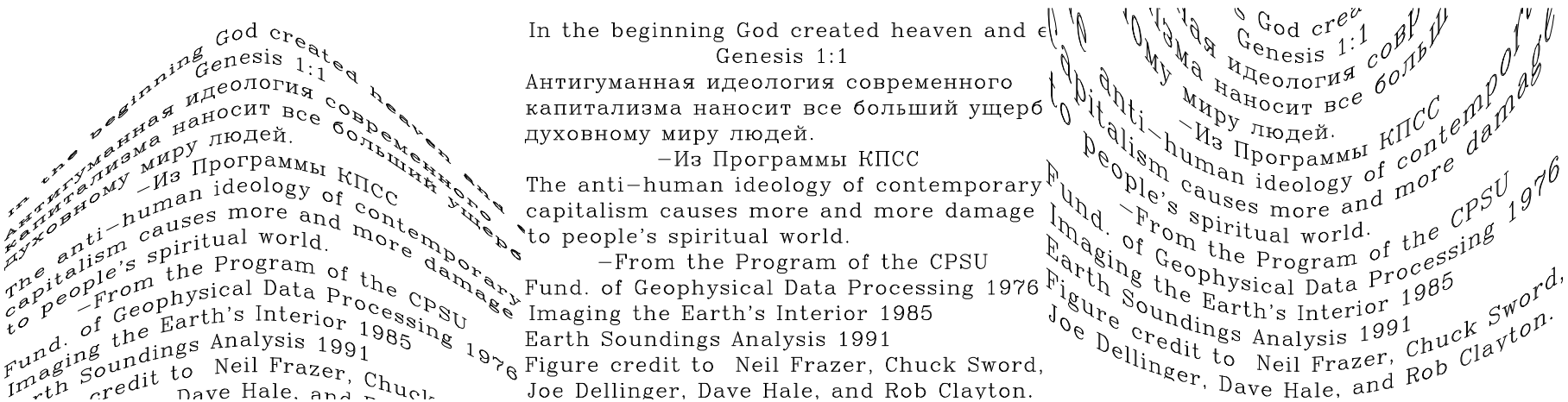


Figure 5.4: Roughly, NMO takes each panel to the one on its right. conj-frazer [ER]

hyperbolas lie at 45°. The main thing to notice is that **NMO stretch**es information at wide offsets and early time, whereas modeling, its inverse, squeezes it. More

precisely, starting from the center panel, adjoint NMO created the left panel, and NMO created the right panel. Notice that adjoint NMO throws away data at late time, whereas NMO itself throws away data at early time. Otherwise, adjoint NMO in this example is the same as inverse NMO.

Normal moveout is a linear operation. This means that data can be decomposed into any two parts, early and late, high frequency and low, smooth and rough, steep and shallow dip, etc.; and whether the two parts are NMO'ed either separately or together, the result is the same, i.e., $\mathbf{N}(\mathbf{a}+\mathbf{b}) = \mathbf{N}\mathbf{a}+\mathbf{N}\mathbf{b}$.

Figure 5.5 shows a marine dataset before and after NMO correction at the water velocity. You can notice that the wave packet reflected from the ocean bottom is approximately a constant width on the raw data. After NMO, however, this waveform broadens considerably—a phenomenon known as "NMO stretch."

The NMO transformation $\mathbf{N}$ is representable as a square matrix. The matrix $\mathbf{N}$ is a (τ, t)-plane containing all zeros except an interpolation operator centered along the hyperbola. The dots in the matrix below are zeros. The input signal x_t is put into the vector $\mathbf{x}$. (This x_t should not be confused with the x_0 denoting distance in the hyperbola $t^2 = \tau^2 + x_0^2/v^2$.) The output vector $\mathbf{y}$—i.e., the NMO'ed signal—is simply $(x_6, x_6, x_6, x_7, x_7, x_8, x_8, x_9, x_{10}, 0)$. In real life, the subscript would go up to

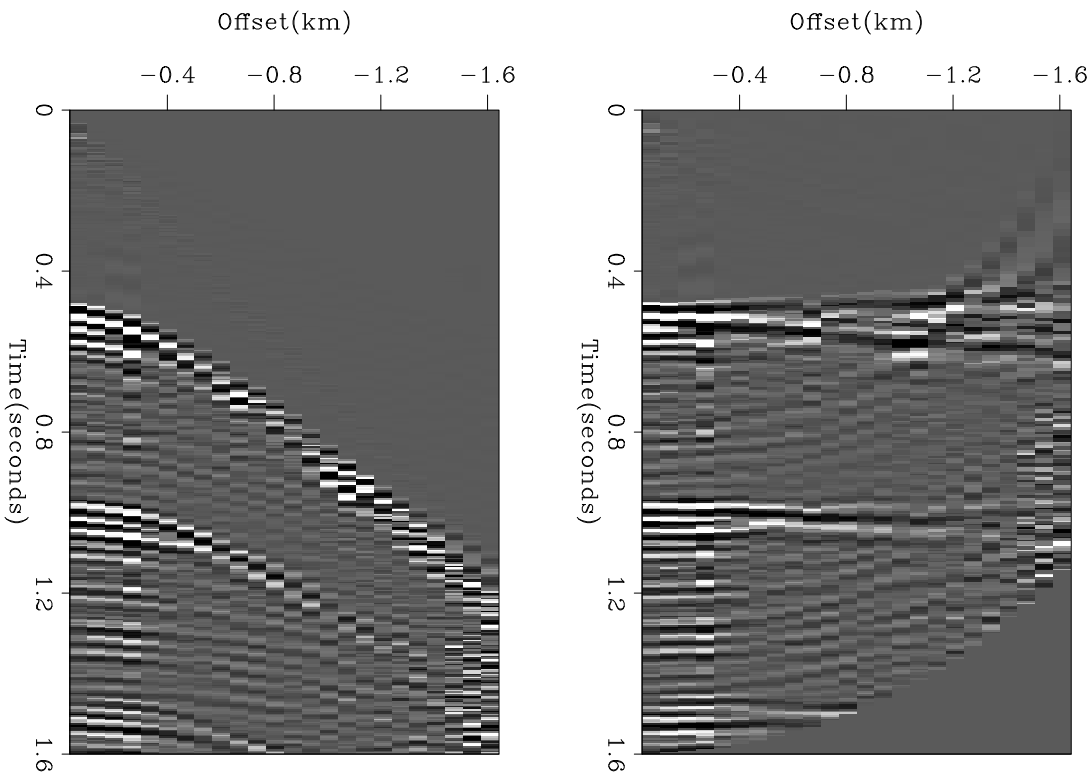


Figure 5.5: Marine data moved out with water velocity. Input on the left, output on the right. Press button for movie sweeping through velocity (actually through slowness squared). conj-stretch [ER,M]

about one thousand instead of merely to ten.

$$\mathbf{y} \quad = \quad \mathbf{N}\mathbf{x} \quad = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \end{bmatrix} = \begin{bmatrix} . & . & . & . & . & 1 & . & . & . & . \\ . & . & . & . & . & 1 & . & . & . & . \\ . & . & . & . & . & 1 & . & . & . & . \\ . & . & . & . & . & . & 1 & . & . & . \\ . & . & . & . & . & . & 1 & . & . & . \\ . & . & . & . & . & . & . & 1 & . & . \\ . & . & . & . & . & . & . & 1 & . & . \\ . & . & . & . & . & . & . & . & 1 & . \\ . & . & . & . & . & . & . & . & . & 1 \\ . & . & . & . & . & . & . & . & . & . \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \end{bmatrix} \tag{5.17}$$

You can think of the matrix as having a horizontal t-axis and a vertical τ-axis. The 1's in the matrix are arranged on the hyperbola $t^2 = \tau^2 + x_0^2/v^2$. The transpose matrix defining some $\tilde{\mathbf{x}}$ from $\mathbf{y}$ gives pseudodata $\tilde{\mathbf{x}}$ from the zero-offset (or stack)

model **y**, namely,

$$\tilde{\mathbf{x}} \quad = \quad \mathbf{N}'\mathbf{y} \quad = \quad \begin{bmatrix} \tilde{x}_1 \\ \tilde{x}_2 \\ \tilde{x}_3 \\ \tilde{x}_4 \\ \tilde{x}_5 \\ \tilde{x}_6 \\ \tilde{x}_7 \\ \tilde{x}_8 \\ \tilde{x}_9 \\ \tilde{x}_{10} \end{bmatrix} \quad = \quad \begin{bmatrix} . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ 1 & 1 & 1 & . & . & . & . & . & . & . \\ . & . & . & 1 & 1 & . & . & . & . & . \\ . & . & . & . & . & 1 & 1 & . & . & . \\ . & . & . & . & . & . & . & 1 & . & . \\ . & . & . & . & . & . & . & . & 1 & . \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \end{bmatrix} \tag{5.18}$$

A program for **nearest-neighbor normal moveout** as defined by equations (5.17) and (5.18) is `nmo1()`. Because of the limited alphabet of programming languages, I used the keystroke `z` to denote τ. nmo1

```
subroutine nmo1( adj, add, slow,    x, t0, dt, n,zz,  tt )
integer  it, iz, adj, add,                        n
real  xs, t , z,           slow(n), x, t0, dt, zz(n), tt(n), wt
call adjnull(    adj, add,                        zz,n,  tt,n)
do iz= 1, n {   z = t0 + dt*(iz-1)
                xs = x * slow(iz)
                t = sqrt ( z * z + xs * xs) + 1.e-20
                wt = z/t * (1./sqrt(t))                  #  weighting function
                it = 1 + .5 + (t - t0) / dt
                if( it <= n )
                        if( adj == 0 )
                                tt(it) = tt(it) + zz(iz) * wt
                        else
                                zz(iz) = zz(iz) + tt(it) * wt
            }
return; end
```

Back

5.3.5. Stack

Typically, many receivers record every shot. Each seismogram can be transformed by NMO and the results all added. This is called "**stack**ing" or "**NMO stack**ing." The adjoint to this operation is to begin from a model that is identical to the near-offset trace and spray this trace to all offsets. There is no "official" definition of which operator of an operator pair is the operator itself and which is the adjoint. On the one hand, I like to think of the modeling operation itself as *the* operator. On the other hand, the industry machinery keeps churning away at many processes that have well-known names, so I often think of one of them as *the* operator. Industrial data-processing operators are typically **adjoint**s to modeling operators.

Figure 5.6 illustrates the operator pair, consisting of spraying out a zero-offset trace (the model) to all offsets and the adjoint of the spraying, which is **stack**ing. The moveout and stack operations are in subroutine `stack1()`. stack1 Let **S** denote NMO, and let the stack be defined by invoking `stack1()` with the `conj=0` argument. Then $\mathbf{S}'$ is the modeling operation defined by invoking `stack1()` with the `conj=1` argument. Figure 5.6 illustrates both. Notice the roughness on the waveforms caused by different numbers of points landing in one place. Notice also the increase of **AVO** as the waveform gets compressed into a smaller space. Finally, no-

```
subroutine stack1( adj, add, slow,     t0,dt, x0,dx, nt,nx, stack, gather)
integer ix,        adj, add,                         nt,nx
real     x,                 slow(nt), t0,dt, x0,dx,   stack(nt), gather(nt,nx)
call adjnull(      adj, add,                           stack,nt,  gather,nt*nx)
do ix= 1, nx {
        x = x0 + dx * (ix-1)
        call nmo1( adj, 1, slow, x, t0,dt, nt, stack, gather(1,ix))
        }
return; end
```

Back

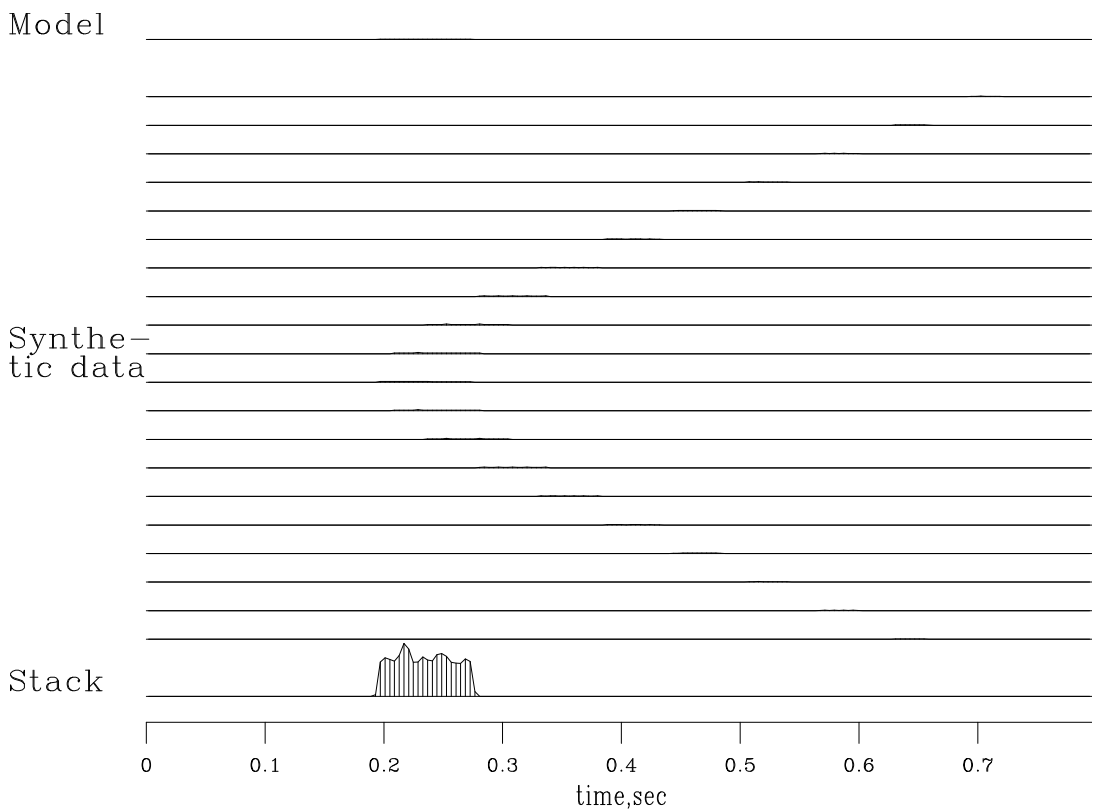


Figure 5.6: Top is a model trace **m**. Center shows the spraying to synthetic traces, $\mathbf{S'm}$. Bottom is the stack of the synthetic data, $\mathbf{SS'm}$. conj-stack [ER]

tice that the stack is a little rough, but the energy is all in the desired time window.

We notice a contradiction of aspirations. On the one hand, an operator has smooth outputs if it "loops over output space" and finds its input where-ever it may. On the other hand, it is nice to have modeling and processing be exact adjoints of each other. Unfortunately, we cannot have both. If you loop over the output space of an operator, then the adjoint operator has a loop over input space and a consequent roughness of its output.

Unfortunately, the adjoint operator $\mathbf{N}'$ defined by the subroutine `nmo1()` /prog:nmo1 is not a good operator for seismogram modeling—notice the roughness of the synthetic seismograms in Figure 5.6. This roughness is *not* an inevitable consequence of nearest-neighbor interpolation. It is a consequence of defining the NMO program as a loop over the output space τ. Instead, we can define inverse NMO as a loop over *its* output space, which is not τ but t. This is done in `imo1()` /prog:imo1.

imo1 imospray

```
subroutine imo1( adj, add, xs, t0, dt, nt, zz,     tt )
integer          adj, add,             nt,                                  it, iz
real                                t0, dt,    zz(nt), tt(nt),    t, xs, zsquared
call adjnull(    adj, add,                     zz,nt,  tt,nt)
do it= 1, nt {  t = t0 + dt*(it-1)
        zsquared =  t * t - xs * xs
        if ( zsquared >= 0.) {  iz = 1.5 + (sqrt( zsquared) - t0) /dt
                if ( iz > 0 ) { if( adj == 0 )
                                        tt(it) = tt(it) + zz(iz)
                                else
                                        zz(iz) = zz(iz) + tt(it)
                                }
                }
        }
return; end
```

Back

```
# inverse moveout and spray into a gather.
#
subroutine imospray( adj, add, slow, x0,dx, t0,dt, nx,nt, stack, gather)
integer ix,          adj, add,                     nx,nt
real xs,                          slow, x0,dx, t0,dt,    stack(nt), gather( nt,nx)
call adjnull(        adj, add,                           stack,nt,  gather, nt*nx)
do ix= 1, nx {
        xs = (x0 + dx * (ix-1)) * slow
        call imo1( adj, 1, xs, t0, dt, nt,             stack,     gather(1,ix))
        }
return; end
```

Back

5.3.6. Pseudoinverse to nearest-neighbor NMO

Examine the matrix $\mathbf{N}'\mathbf{N}$:

$$\mathbf{N}'\mathbf{N} = \begin{bmatrix} . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & 3 & . & . & . & . \\ . & . & . & . & . & . & 2 & . & . & . \\ . & . & . & . & . & . & . & 2 & . & . \\ . & . & . & . & . & . & . & . & 1 & . \\ . & . & . & . & . & . & . & . & . & 1 \end{bmatrix} \tag{5.19}$$

Any mathematician will say that equation (5.19) is not invertible because the zeros on the diagonal make it singular. But as a geophysicist, you know better. Our

inverse, called a "**pseudoinverse**," is

$$(\mathbf{N}'\mathbf{N})^{-1} = \begin{bmatrix} . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & \frac{1}{3} & . & . & . & . \\ . & . & . & . & . & . & \frac{1}{2} & . & . & . \\ . & . & . & . & . & . & . & \frac{1}{2} & . & . \\ . & . & . & . & . & . & . & . & 1 & . \\ . & . & . & . & . & . & . & . & . & 1 \end{bmatrix} \tag{5.20}$$

We could write code for inverse NMO, which is an easy task, or we could try to write code for inverse NMO and stack, which has no clean solution known to me. Instead, we move to other topics.

5.3.7. Null space and inconsistency

The normal-moveout transformation is a textbook example of some of the pathologies of simultaneous equation solving. Reexamine equation (5.17), thinking of it as a set of simultaneous equations for x_i given y_i. First, (5.17) shows that there may exist a set of y_i for which no solution x_i is possible—any set containing $y_{10} \neq 0$, for example. This is an example of **inconsistency** in simultaneous equations. Second, there are **x** vectors that satisfy $\mathbf{Nx} = \mathbf{0}$, so any number of such vectors can be added into any solution and it remains a solution. These solutions are called the "**null space**." Here these solutions are the arbitrary values of x_1, x_2, x_3, x_4, and x_5 that obviously leave **y** unaffected. Typical matrices disguise their inconsistencies and null spaces better than does the NMO transformation. To make such a transformation, we could start from the NMO transformation and apply any coordinate transformation to the vectors **x** and **y**.

EXERCISES:

1 A succession of normal-moveout operators is called "**cascaded NMO**." Consider NMO from time t'' to traveltime depth t' by $t''^2 = t'^2 + x^2/v_2^2$, followed by another NMO transform which uses the transformation equation

$t'^2 = t^2 + x^2/v_1^2$. Show that the overall transformation is another NMO transformation. What is its velocity? Notice that cascaded NMO can be used to correct an NMO velocity. Thus it can be called residual velocity analysis or residual normal moveout.

5.3.8. NMO with linear interpolation

NMO with **linear interpolation** implies that the matrix **N** is a two-band matrix. Each row has exactly two elements that interpolate between two elements on the input. I will sketch the appearance of the matrix, using the letters *a* and *b* for the

elements. Each a and b is different numerically, but on a given row, $a+b=1$.

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \end{bmatrix} = \begin{bmatrix} . & . & . & . & a & b & . & . & . & . \\ . & . & . & . & a & b & . & . & . & . \\ . & . & . & . & a & b & . & . & . & . \\ . & . & . & . & . & a & b & . & . & . \\ . & . & . & . & . & a & b & . & . & . \\ . & . & . & . & . & . & a & b & . & . \\ . & . & . & . & . & . & a & b & . & . \\ . & . & . & . & . & . & . & a & b & . \\ . & . & . & . & . & . & . & . & a & b \\ . & . & . & . & . & . & . & . & . & a \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \end{bmatrix} \tag{5.21}$$

Here the matrix $\mathbf{N}'\mathbf{N}$ is tridiagonal, but I am going to let you work out the details by yourself. The original data can be recovered by solving the tridiagonal system. This method can be used to program an invertible NMO or to program an invertible trace interpolation. I do not want to clutter this book with the many details. Instead, I present `spot1()`, a convenient subroutine for linear interpolation that can be used in many applications. spot1

```
# Nearest neighbor interpolation would do this:    val = vec( 1.5 + (t-t0)/dt)
# This is the same but with _linear_ interpolation.
#
subroutine spot1( adj, add, nt,t0,dt, t, val,    vec    )
integer it, itc,  adj, add, nt
real tc, fraction,             t0,dt, t, val,    vec(nt)
call adjnull(    adj, add,                val, 1, vec,nt)
tc      = (t-t0) / dt
itc     =  tc
it      = 1 + itc;                      fraction = tc - itc
if( 1 <= it  &&  it < nt)
        if( adj == 0) {                                  # add value onto vector
                vec(it  ) = vec(it  ) + (1.-fraction) * val
                vec(it+1) = vec(it+1) +   fraction    * val
                }
        else                                             # take value from vector
                val = val + (1.-fraction) * vec(it)  +  fraction * vec(it+1)
return; end
```

Back

5.4. DERIVATIVE AND INTEGRAL

Differentiation and integration are very basic operations. Their adjoints are best understood when they are represented in the sampled-time domain, rather than the usual time continuum.

5.4.1. Adjoint derivative

Given a sampled signal, its time derivative can be estimated by convolution with the filter $(1,-1)/\Delta t$. This could be done with any convolution program. For example if we choose to ignore end effects we might select `convin()` /prog:convin. This example arises so frequently that I display the matrix multiply below:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} -1 & 1 & . & . & . & . \\ . & -1 & 1 & . & . & . \\ . & . & -1 & 1 & . & . \\ . & . & . & -1 & 1 & . \\ . & . & . & . & -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \tag{5.22}$$

The filter impulse response is seen in any column in the middle of the matrix, namely $(1, -1)$. In the transposed matrix the filter impulse response is time reversed to $(-1, 1)$. So, mathematically, we can say that the adjoint of the time derivative operation is the negative time derivative. This corresponds also to the fact that the complex conjugate of $-i\omega$ is $i\omega$. We can also speak of the adjoint of the boundary conditions: we might say the adjoint of "no boundary condition" is "specified value" boundary conditions.

Banded matrices like in (5.21) and (5.22) arise commonly, and subroutines like `convin()` /prog:convin are awkward and over-general because they sum with a `do` loop where a mere statement of the two terms is enough. This is illustrated in subroutine `ruffen1()`. Notice the adjoint calculation resembles that in `spot1()` /prog:spot1. ruffen1

5.5. CAUSAL INTEGRATION RECURSION

Causal integration is defined as

$$y(t) \quad = \quad \int_{-\infty}^{t} x(t)\, dt \tag{5.23}$$

```
subroutine ruffen1( adj,    n, xx, yy     )
integer i,          adj,       n
real                           xx(n), yy( n-1)
call adjnull(       adj, 0, xx,n,  yy, n-1)
do i= 1, n-1 {
        if( adj == 0 )
                yy(i) = yy(i) + xx(i+1) - xx(i)
        else {
                xx(i+1) = xx(i+1) + yy(i)
                xx(i  ) = xx(i  ) - yy(i)
                }
        }
return; end
```

Back

Sampling the time axis gives a matrix equation which we should call causal summation, but we often call it causal integration.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix} \tag{5.24}$$

(In some applications the 1 on the diagonal is replaced by 1/2.) Causal integration is the simplest prototype of a recursive operator. The coding is trickier than operators we considered earlier. Notice when you compute y_5 that it is the sum of 6 terms, but that this sum is more quickly computed as $y_5 = y_4 + x_5$. Thus equation (5.24)

is more efficiently thought of as the recursion

$$y_t \quad = \quad y_{t-1} + x_t \qquad \text{for increasing } t \tag{5.25}$$

(which may also be regarded as a numerical representation of the differential equation $dy/dt = x$.)

When it comes time to think about the adjoint, however, it is easier to think of equation (5.24) than of (5.25). Let the matrix of equation (5.24) be called $\mathbf{C}$. Transposing to get $\mathbf{C}'$ and applying it to $\mathbf{y}$ gives us something back in the space of $\mathbf{x}$, namely $\tilde{\mathbf{x}} = \mathbf{C}'\mathbf{y}$. From it we see that the adjoint calculation, if done recursively, needs to be done backwards like

$$\tilde{x}_{t-1} \quad = \quad \tilde{x}_t + y_{t-1} \qquad \text{for decreasing } t \tag{5.26}$$

We can sum up by saying that the adjoint of causal integration is anticausal integration.

A subroutine to do these jobs is `causint()` /prog:causint. The code for anticausal integration is not obvious from the code for integration and the adjoint coding tricks we learned earlier. To understand the adjoint, you need to inspect the detailed form of the expression $\tilde{\mathbf{x}} = \mathbf{C}'\mathbf{y}$ and take care to get the ends correct. causint

```
# causal integration  (1's on diagonal)
#
subroutine causint( adj, add,         n,xx,  yy    )
integer i,        n, adj, add;   real xx(n), yy(n )
temporary real tt( n)
call adjnull(        adj, add,         xx,n,  yy,n )
if( adj == 0){                                    tt(1) = xx(1)
                                             do i= 2, n
                                                  tt(i) = tt(i-1) + xx(i)
        do i= 1, n
                yy(i) = yy(i) + tt(i)
        }
else {                                            tt(n) = yy(n)
                                             do i= n, 2, -1
                                                  tt(i-1) = tt(i) + yy(i-1)
        do i= 1, n
                xx(i) = xx(i) + tt(i)
        }
return; end
```

Back

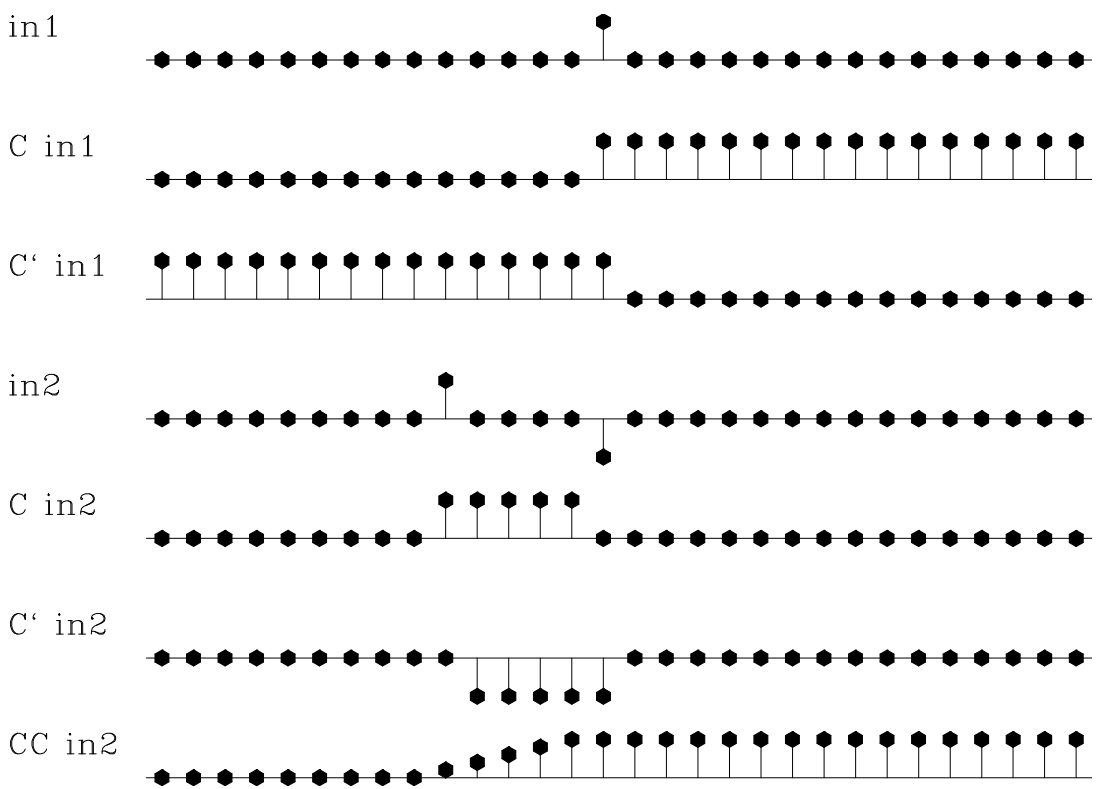


Figure 5.7: `in1` is an input pulse. `C in1` is its causal integral. `C' in1` is the anticausal integral of the pulse. `in2` is a separated doublet. Its causal integration is a box and its anticausal integration is the negative. `CC in2` is the double causal integral of `in2`. How can a triangle be built? conj-causint [ER]

Later we will consider equations to march wavefields up towards the earth's surface, a layer at a time, an operator for each layer. Then the adjoint will start from the earth's surface and march down, a layer at a time, into the earth.

EXERCISES:

1 Modify the calculation in Figure 5.7 to make a triangle waveform on the bottom row.

5.5.1. Readers' guide

Now we have completed our discussion of most of the essential common features of adjoint operators. You can skim forward to the particular operators of interest to you without fear of missing anything essential.

5.6. UNITARY OPERATORS

The nicest operators are unitary. Let us examine the difference between a unitary operator and a nonunitary one.

5.6.1. Meaning of B'B

A matrix operation like $\mathbf{B}'\mathbf{B}$ arises whenever we travel from one space to another and back again. The inverse of this matrix arises when we ask to return from the other space with no approximations. In general, $\mathbf{B}'\mathbf{B}$ can be complicated beyond comprehension, but we have seen some recurring features. In some cases this matrix turned out to be a diagonal matrix which is a scaling function in the physical domain. With banded matrices, the $\mathbf{B}'\mathbf{B}$ matrix is also a banded matrix, being tridiagonal for $\mathbf{B}$ operators of both (5.22) and (5.21). The banded matrix for the derivative operator (5.22) can be thought of as the frequency domain weighting factor ω^2. We did not examine $\mathbf{B}'\mathbf{B}$ for the filter operator, but if you do, you will see that the rows (and the columns) of $\mathbf{B}'\mathbf{B}$ are the **autocorrelation** of the filter. A filter in the time domain is simply a weighting function in the frequency domain.

The tridiagonal banded matrix for linearly-interpolated NMO is somewhat more complicated to understand, but it somehow represents the smoothing inherent to the composite process of NMO followed by adjoint NMO, so although we may not fully understand it, we can think of it as some multiplication in the spectral domain as well as some rescaling in the physical domain. Since $\mathbf{B}'\mathbf{B}$ clusters on the main diagonal, it never has a "time-shift" behavior.

5.6.2. Unitary and pseudounitary transformation

A so-called **unitary** transformation $\mathbf{U}$ conserves energy. In other words, if $\mathbf{v} = \mathbf{U}\mathbf{x}$, then $\mathbf{x}'\mathbf{x} = \mathbf{v}'\mathbf{v}$, which requires $\mathbf{U}'\mathbf{U} = \mathbf{I}$. Imagine an application where the transformation seems as if it should not destroy information. Can we arrange it to conserve energy? The conventional inversion

$$\mathbf{y} = \mathbf{B}\mathbf{x} \tag{5.27}$$

$$\mathbf{x} = (\mathbf{B}'\mathbf{B})^{-1}\mathbf{B}'\mathbf{y} \tag{5.28}$$

can be verified by direct substitution. Seeking a more symmetrical transformation between $\mathbf{y}$ and $\mathbf{x}$ than the one above, we define

$$\mathbf{U} = \mathbf{B}(\mathbf{B}'\mathbf{B})^{-1/2} \tag{5.29}$$

and the transformation pair

$$\mathbf{v} = \mathbf{U}\mathbf{x} \tag{5.30}$$

$$\mathbf{x} = \mathbf{U}'\mathbf{v} \tag{5.31}$$

where we can easily verify that $\mathbf{x}'\mathbf{x} = \mathbf{v}'\mathbf{v}$ by direct substitution. In practice, it would often be found that $\mathbf{v}$ is a satisfactory substitute for $\mathbf{y}$, and further that the unitary

property is often a significant advantage.

Is the operator $\mathbf{U}$ unitary? It would not be unitary for NMO, because equation (5.19) is not invertible. Remember that we lost (x_1, x_2, x_3, x_4, and x_5) in (5.17). $\mathbf{U}$ *is* unitary, however, except for lost points, so we call it "**pseudounitary**." A trip into and back from the space of a pseudounitary operator is like a pass through a bandpass filter. Something is lost the first time, but no more is lost if we do it again. Thus, $\mathbf{x} \neq \mathbf{U}'\mathbf{U}\mathbf{x}$, but $\mathbf{U}'\mathbf{U}\mathbf{x} = \mathbf{U}'\mathbf{U}(\mathbf{U}'\mathbf{U}\mathbf{x})$ for any $\mathbf{x}$. Furthermore, $(\mathbf{U}'\mathbf{U})^2 = \mathbf{U}'\mathbf{U}$, but $\mathbf{U}'\mathbf{U} \neq \mathbf{I}$. In mathematics the operators $\mathbf{U}'\mathbf{U}$ and $\mathbf{U}\mathbf{U}'$ are called "**idempotent**" operators. Another example of an idempotent operator is that of subroutine `advance()`
/prog:advance

5.6.3. Pseudounitary NMO with linear interpolation

It is often desirable to work with transformations that are as nearly unitary as possible, i.e., their transpose is their pseudoinverse. These transformations are pseudounitary. Let us make NMO with *linear* interpolation into a pseudounitary transformation. We need to factor the tridiagonal matrix $\mathbf{N}'\mathbf{N} = \mathbf{T}$ into bidiagonal parts, $\mathbf{T} = \mathbf{B}'\mathbf{B}$. One such factorization is the well-known **Cholesky decomposition**;

which is like spectral factorization. (We never really need to look at square roots of matrices). Then we will define **pseudounitary NMO** as $\mathbf{U} = \mathbf{N}\mathbf{B}^{-1}$. To confirm the unitary property, we check that $\mathbf{U}'\mathbf{U} = \mathbf{B}'^{-1}\mathbf{N}'\mathbf{N}\mathbf{B}^{-1} = \mathbf{B}'^{-1}\mathbf{B}'\mathbf{B}\mathbf{B}^{-1} = \mathbf{I}$. An **all-pass filter** is a ratio of two terms, both with the same color, the denominator minimum phase, and the numerator not. Analogously, in $\mathbf{U} = \mathbf{N}\mathbf{B}^{-1}$, the numerator time shifts, and the denominator corrects the numerator's color.

EXERCISES:

1 Explain why normal moveout is not generally invertible where velocity depends on depth.

2 What adaptations should be made to equation (5.17) to make it pseudounitary?

3 Extend subroutine `wavekill1()` /prog:wavekill1 to include the adjoint considering the wave *input* to be dual to its output (not considering the *filter* to be dual to the output).

5.7. VELOCITY SPECTRA

An important transformation in exploration geophysics is from data as a function of shot-receiver offset to data as a function of apparent velocity. To go from offset to velocity, the transformation sums along hyperbolas of many velocities. The adjoint is a superposition of hyperbolas of all the different velocities. Pseudocode for these transformations is

```
do v
    do τ
        do x
            t = sqrt(τ^2 + x^2/v^2)
            if hyperbola superposition
                data(t,x)= data(t,x) + vspace(τ,v)
            else if velocity analysis
                vspace(τ,v)=vspace(τ,v)+data(t,x)
```

5.8. INTRODUCTION TO TOMOGRAPHY

Tomography is the reconstruction of a function from line integrals through the function. Tomography has become a routine part of medicine, and an experimental part of earth sciences. For illustration, a simple arrangement is well-to-well tomography. A sound source can be placed at any depth in one well and receivers placed at any depth in another well. At the sender well, we have sender depths s, and at the receiver well, we have receiver depths g. Our data is a table $t(s,g)$ of traveltimes from s to g. The idea is to try to map the area between the wells. We divide the area between wells into cells in (x,z)-space. The map could be one of material velocities or one of absorptivity. The traveltime of a ray increases by adding the *slownesses* of cells traversed by the ray. Our model is a table $s(x,z)$ of slownesses in the plane between wells. (Alternately, the logarithm of the amplitude of the ray is a summation of absorptivities of the cells traversed.) The pseudocode is

do s = range of sender locations
do g = range of receiver locations
$z = z(s)$ # depth of sender.
$\theta = \theta$(s,g) # ray take-off angle.
do x = range from senders to receivers.
$z = z + \Delta x \tan\theta$ # ray tracing
if modeling
$t_{sg} = t_{sg} + s_{xz}\ \Delta x / \cos\theta$
else tomography
$s_{xz} = s_{xz} + t_{sg}\ \Delta x / \cos\theta$

In the pseudocode above, we assumed that the rays were straight lines. The problem remains one of linear operators even if the rays curve, making ray tracing more complicated. If the solution $s(x,z)$ is used to modify the ray tracing then the problem becomes nonlinear, requiring the complexities of nonlinear optimization theory.

5.8.1. Units

Notice that the physical units of an operator (such as the meters or feet implied by Δx) are the *same* as the physical units of the adjoint operator. The units of an inverse operator, however, are *inverse* to the units of the original operator. Thus it is hard to imagine that an adjoint operator could ever be a satisfactory approximation to the inverse. We know, however, that adjoints often are a satisfactory approximation to an inverse, which means then that either (1) such operators do not have physical units, or (2) a scaling factor in the final result is irrelevant. With the tomographic operator, the adjoint is quite far from the inverse so practicioners typically work from the adjoint toward the inverse.

Some operators are arrays with different physical units for different array elements. For these operators the adjoint is unlikely to be a satisfactory approximation to the inverse since changing the units changes the adjoint. A way to bring all components to the same units is to redefine each member of data space and model space to be itself divided by its variance. Alternately, again we can abandon the idea of finding immediate utility in the adjoint of an operator and and we could progress from the adjoint toward the inverse.

EXERCISES:

1 Show how to adapt tomography for "fat" rays of thickness N_z points along the z-axis.

5.9. STOLT MIGRATION

NMO is based on the quadratic equation $v^2t^2 = z^2 + x^2$ (as explained in IEI). **Stolt migration** is based on the quadratic equation $\omega^2/v^2 = k_z^2 + k_x^2$, which is the dispersion relation of the scalar wave equation. Stolt migration is NMO in the Fourier domain (see IEI). Denote the Fourier transform operator by **F** and the Stolt operator by **S**, where

$$\mathbf{S} \quad = \quad \mathbf{F}'\mathbf{N}\mathbf{F} \tag{5.32}$$

A property of matrix adjoints is $(\mathbf{ABC})' = \mathbf{C}'\mathbf{B}'\mathbf{A}'$. We know the transpose of NMO, and we know that the adjoint of Fourier transformation is inverse Fourier transformation. So

$$\mathbf{S}' \quad = \quad \mathbf{F}'\mathbf{N}'\mathbf{F} \tag{5.33}$$

We see then that the transpose to Stolt modeling is Stolt migration. (There are a few more details with Stolt's **Jacobian**.)

5.10. References

Nolet, G., 1985, Solving or resolving inadequate and noisy tomographic systems: J. Comp. Phys., **61**, 463-482.

Thorson, J.R., 1984, Velocity stack and slant stack inversion methods: Ph.D. thesis, Stanford University.

Chapter 6

Model fitting by least squares

The first level of computer use in science and engineering is “**modeling**." Beginning from physical principles and design ideas, the computer mimics nature. After this, the worker looks at the result and thinks a while, then alters the modeling program

and tries again. The next, deeper level of computer use is that the computer itself examines the results of modeling and reruns the modeling job. This deeper level is variously called "**fitting**" or "**inversion**." The term "**processing**" is also used, but it is broader, including the use of adjoint operators (as discussed in chapter 5). Usually people are more effective than computers at fitting or inversion, but some kinds of fitting are more effectively done by machines. A very wide range of methods comes under the heading of "**least squares**," and these methods are the topic of this chapter and chapters 7 through **??**.

A part of basic education in mathematics is the fitting of scattered points on a plane to a straight line. That is a simple example of inversion, a topic so grand and broad that some people think of learning to do inversion as simply "learning." Although I will be drawing many examples from my area of expertise, namely, earth soundings analysis, the methods presented here are much more widely applicable.

6.1. MULTIVARIATE LEAST SQUARES

As described at the beginning of chapter 4, signals and images will be specified here by numbers packed into abstract vectors. We consider first a hypothetical applica-

tion with one data vector $\mathbf{d}$ and two fitting vectors $\mathbf{b}_1$ and $\mathbf{b}_2$. Each fitting vector is also known as a "**regressor**." Our first task is to try to approximate the data vector $\mathbf{d}$ by a scaled combination of the two regressor vectors. The scale factors x_1 and x_2 should be chosen so that the model matches the data, i.e.,

$$\mathbf{d} \quad \approx \quad \mathbf{b}_1 x_1 + \mathbf{b}_2 x_2 \tag{6.1}$$

For example, if I print the characters "**P**" and "**b**" on top of each other, I get "**B**," which looks something like an image of the letter "**B**." This is analogous to $\mathbf{d} \approx \mathbf{b}_1 + \mathbf{b}_2$. More realistically, $\mathbf{d}$ could contain a sawtooth function of time, and $\mathbf{b}_1$ and $\mathbf{b}_2$ could be sinusoids. Still more realistically, $\mathbf{d}$ could be an observed 2-D wave field, and $\mathbf{b}_1$ and $\mathbf{b}_2$ could be theoretical data in two parts, where the contribution of each part is to be learned by fitting. (One part could be primary reflections and the other multiple reflections.)

Notice that we could take the partial derivative of the data in (6.1) with respect to an unknown, say x_1, and the result is the regressor $\mathbf{b}_1$.

The **partial derivative** of *all* data with respect to *any* model parameter gives a **regressor**. A **regressor** is a column in the partial-derivative matrix.

Equation (6.1) is often expressed in the more compact mathematical matrix notation $\mathbf{d} \approx \mathbf{Bx}$, but in our derivation here we will keep track of each component explicitly and use mathematical matrix notation to summarize the final result. Fitting the data $\mathbf{d}$ to its two theoretical components can be expressed as minimizing the length of the residual vector $\mathbf{r}$, where

$$\mathbf{r} = \mathbf{d} - \mathbf{b}_1 x_1 - \mathbf{b}_2 x_2 \tag{6.2}$$

So we construct a sum of squares (also called a "**quadratic form**") of the components of the residual vector by using a dot product:

$$Q(x_1, x_2) = \mathbf{r} \cdot \mathbf{r} \tag{6.3}$$

$$= (\mathbf{d} - \mathbf{b}_1 x_1 - \mathbf{b}_2 x_2) \cdot (\mathbf{d} - \mathbf{b}_1 x_1 - \mathbf{b}_2 x_2) \tag{6.4}$$

The gradient of $Q(x_1, x_2)/2$ is defined by its two components:

$$\frac{\partial Q}{\partial x_1} = -\mathbf{b}_1 \cdot (\mathbf{d} - \mathbf{b}_1 x_1 - \mathbf{b}_2 x_2) - (\mathbf{d} - \mathbf{b}_1 x_1 - \mathbf{b}_2 x_2) \cdot \mathbf{b}_1 \tag{6.5}$$

$$\frac{\partial Q}{\partial x_2} = -\mathbf{b}_2 \cdot (\mathbf{d} - \mathbf{b}_1 x_1 - \mathbf{b}_2 x_2) - (\mathbf{d} - \mathbf{b}_1 x_1 - \mathbf{b}_2 x_2) \cdot \mathbf{b}_2 \tag{6.6}$$

Setting these derivatives to zero and using $(\mathbf{b}_1 \cdot \mathbf{b}_2) = (\mathbf{b}_2 \cdot \mathbf{b}_1)$ etc., we get

$$(\mathbf{b}_1 \cdot \mathbf{d}) = (\mathbf{b}_1 \cdot \mathbf{b}_1)x_1 + (\mathbf{b}_1 \cdot \mathbf{b}_2)x_2 \tag{6.7}$$

$$(\mathbf{b}_2 \cdot \mathbf{d}) = (\mathbf{b}_2 \cdot \mathbf{b}_1)x_1 + (\mathbf{b}_2 \cdot \mathbf{b}_2)x_2 \tag{6.8}$$

which two equations we can use to solve for the two unknowns x_1 and x_2. Writing this expression in matrix notation, we have

$$\begin{bmatrix} (\mathbf{b}_1 \cdot \mathbf{d}) \\ (\mathbf{b}_2 \cdot \mathbf{d}) \end{bmatrix} = \begin{bmatrix} (\mathbf{b}_1 \cdot \mathbf{b}_1) & (\mathbf{b}_1 \cdot \mathbf{b}_2) \\ (\mathbf{b}_2 \cdot \mathbf{b}_1) & (\mathbf{b}_2 \cdot \mathbf{b}_2) \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{6.9}$$

It is customary to use matrix notation without dot products. For this we need some additional definitions. To clarify these definitions, I choose the number of components in the vectors $\mathbf{b}_1$, $\mathbf{b}_2$, and $\mathbf{d}$ to be three. Thus I can explicitly write a matrix $\mathbf{B}$ in full as

$$\mathbf{B} = [\mathbf{b}_1 \quad \mathbf{b}_2] = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} \tag{6.10}$$

Likewise, the *transposed* matrix $\mathbf{B}'$ is defined by

$$\mathbf{B}' = \begin{bmatrix} b_{11} & b_{21} & b_{31} \\ b_{12} & b_{22} & b_{32} \end{bmatrix} \tag{6.11}$$

The matrix in equation (6.9) contains dot products. Matrix multiplication is an abstract way of representing the dot products:

$$\begin{bmatrix} (\mathbf{b}_1 \cdot \mathbf{b}_1) & (\mathbf{b}_1 \cdot \mathbf{b}_2) \\ (\mathbf{b}_2 \cdot \mathbf{b}_1) & (\mathbf{b}_2 \cdot \mathbf{b}_2) \end{bmatrix} = \begin{bmatrix} b_{11} & b_{21} & b_{31} \\ b_{12} & b_{22} & b_{32} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} \tag{6.12}$$

Thus, equation (6.9) without dot products is

$$\begin{bmatrix} b_{11} & b_{21} & b_{31} \\ b_{12} & b_{22} & b_{32} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} b_{11} & b_{21} & b_{31} \\ b_{12} & b_{22} & b_{32} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{6.13}$$

which has the matrix abbreviation

$$\mathbf{B}'\mathbf{d} = (\mathbf{B}'\,\mathbf{B})\mathbf{x} \tag{6.14}$$

Equation (6.14) is the classic result of least-squares fitting of data to a collection of regressors. Obviously, the same matrix form applies when there are more than two regressors and each vector has more than three components. Equation (6.14) leads to an analytic solution for **x** using an inverse matrix. To solve formally for the unknown **x**, we premultiply by the inverse matrix $(\mathbf{B}'\,\mathbf{B})^{-1}$:

$$\mathbf{x} \quad = \quad (\mathbf{B}'\,\mathbf{B})^{-1}\,\mathbf{B}'\mathbf{d} \tag{6.15}$$

Equation (6.15) is the central result of **least-squares** analysis. We see it everywhere.

Equation (6.12) is an example of what is called a "**covariance matrix**." Such matrices usually need to be inverted, and in equation (6.15) you already see an example of the occurrence of an inverse covariance matrix. Any description of an application of least-squares fitting will generally include some discussion of the covariance matrix—how it will be computed, assumed, or estimated, and how its inverse will be found or approximated. In chapter 4 we found the need to weight residuals by the inverse of their scale. That was our first example of the occurrence of an inverse covariance matrix—although in that case the matrix size was only

1×1.

In our first manipulation of matrix algebra, we move around some parentheses in (6.14):

$$\mathbf{B}'\mathbf{d} \quad = \quad \mathbf{B}'\,(\mathbf{B}\mathbf{x}) \tag{6.16}$$

Moving the parentheses implies a regrouping of terms or a reordering of a computation. You can verify the validity of moving the parentheses by writing (6.16) in full as the set of two equations it represents. Equation (6.14) led to the "analytic" solution (6.15). In a later section on conjugate gradients, we will see that equation (6.16) expresses better than (6.15) the philosophy of computation.

Notice how equation (6.16) invites us to cancel the matrix $\mathbf{B}'$ from each side. We cannot do that of course, because $\mathbf{B}'$ is not a number, nor is it a square matrix with an inverse. If you really want to cancel the matrix $\mathbf{B}'$, you may, but the equation is then only an approximation that restates our original goal (6.1):

$$\mathbf{d} \quad \approx \quad \mathbf{B}\mathbf{x} \tag{6.17}$$

A speedy problem solver might ignore the mathematics covering the previous page, study his or her application until he or she is able to write the statement of wishes (6.17) = (6.1), premultiply by $\mathbf{B}'$, replace $\approx$ by =, getting (6.14), and take

(6.14) to a simultaneous equation-solving program to get $\mathbf{x}$.

The formal literature does not speak of "statement of wishes" but of "**regression**," which is the same concept. In a regression, there is an abstract vector called the residual $\mathbf{r} = \mathbf{d} - \mathbf{B}\mathbf{x}$ whose components should all be small. Formally this is often written as:

$$\min_{\mathbf{x}} \ ||\mathbf{d} - \mathbf{B}\mathbf{x}|| \tag{6.18}$$

The notation above with two pairs of vertical lines looks like double absolute value, but we can understand it as a reminder to square and sum all the components. This notation is more explicit about what is being minimized, but I often find myself sketching out applications in the form of a "statement of wishes," which I call a "**regression**."

6.1.1. Inverse filter example

Let us take up a simple example of **time-series analysis**. Given the input, say $(\cdots, 0, 0, 2, 1, 0, 0, \cdots)$, to some filter, say $\mathbf{f} = (f_0, f_1)$, then the output is necessarily $\mathbf{c} = (2f_0, \ f_0 + 2f_1, \ f_1)$. To design an inverse filter, we would wish to have $\mathbf{c}$ come

out as close as possible to $(1,0,0)$. So the statement of wishes (6.17) is

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \approx \begin{bmatrix} 2 & 0 \\ 1 & 2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \end{bmatrix} \tag{6.19}$$

The method of solution is to premultiply by the matrix $\mathbf{B}'$, getting

$$\begin{bmatrix} 2 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 & 1 & 0 \\ 0 & 2 & 1 \end{bmatrix} \begin{bmatrix} 2 & 0 \\ 1 & 2 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \end{bmatrix} \tag{6.20}$$

Thus,

$$\begin{bmatrix} 2 \\ 0 \end{bmatrix} = \begin{bmatrix} 5 & 2 \\ 2 & 5 \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \end{bmatrix} \tag{6.21}$$

and the **inverse filter** comes out to be

$$\begin{bmatrix} f_0 \\ f_1 \end{bmatrix} = \frac{1}{21} \begin{bmatrix} 5 & -2 \\ -2 & 5 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \end{bmatrix} = \begin{bmatrix} \frac{10}{21} \\ -\frac{4}{21} \end{bmatrix} \tag{6.22}$$

Inserting this value of (f_0, f_1) back into (6.19) yields the actual output $(\frac{20}{21}, +\frac{2}{21}, -\frac{4}{21})$, which is not a bad approximation to $(1,0,0)$.

6.1.2. Normal equations

The basic least-squares equations are often called the "**normal**" equations. The word "normal" means perpendicular. We can rewrite equation (6.16) to emphasize the perpendicularity. Bring both terms to the left, and recall the definition of the residual **r** from equation (6.2):

$$\mathbf{B}'(\mathbf{d}-\mathbf{B}\mathbf{x}) = \mathbf{0} \tag{6.23}$$

$$\mathbf{B}'\mathbf{r} = \mathbf{0} \tag{6.24}$$

Equation (6.24) says that the **residual** vector **r** is perpendicular to each row in the $\mathbf{B}'$ matrix. These rows are the **fitting function**s. Therefore, the residual, after it has been minimized, is perpendicular to the fitting functions.

6.1.3. Differentiation by a complex vector

Complex numbers frequently arise in physical problems, particularly with Fourier series. Let us extend the multivariable least-squares theory to the use of complex-valued unknowns **x**. First recall how complex numbers were handled with single-variable least squares, i.e., as in the discussion leading up to equation (**??**). Use

prime, such as $\mathbf{x}'$, to denote the complex conjugate of the transposed vector $\mathbf{x}$. Now write the positive **quadratic form** as

$$Q(\mathbf{x}', \mathbf{x}) \quad = \quad (\mathbf{B}\mathbf{x} - \mathbf{d})'(\mathbf{B}\mathbf{x} - \mathbf{d}) \quad = \quad (\mathbf{x}'\mathbf{B}' - \mathbf{d}')(\mathbf{B}\mathbf{x} - \mathbf{d}) \tag{6.25}$$

In chapter 4 (after equation (4.16)), we minimized a quadratic form $Q(\bar{X}, X)$ by setting to zero both $\partial Q/\partial \bar{X}$ and $\partial Q/\partial X$. We noted that only one of $\partial Q/\partial \bar{X}$ and $\partial Q/\partial X$ is necessary because they are conjugates of each other. Now take the derivative of Q with respect to the (possibly complex, row) vector $\mathbf{x}'$. Notice that $\partial Q/\partial \mathbf{x}'$ is the complex conjugate transpose of $\partial Q/\partial \mathbf{x}$. Thus, setting one to zero sets the other also to zero. Setting $\partial Q/\partial \mathbf{x}' = \mathbf{0}$ gives the normal equations:

$$\mathbf{0} \quad = \quad \frac{\partial Q}{\partial \mathbf{x}'} \quad = \quad \mathbf{B}'(\mathbf{B}\mathbf{x} - \mathbf{d}) \tag{6.26}$$

The result is merely the complex form of our earlier result (6.14). Therefore, differentiating by a complex vector is an abstract concept, but it gives the same set of equations as differentiating by each scalar component, and it saves much clutter.

6.1.4. Time domain versus frequency domain

Equation (**??**) is a frequency-domain quadratic form that we minimized by varying a single parameter, a Fourier coefficient. Now we will look at the same problem in the time domain. The time domain offers new flexibility with boundary conditions, constraints, and weighting functions. The notation will be that a filter f_t has input x_t and output y_t. In Fourier space this is $Y = XF$. There are two problems to look at, unknown filter F and unknown input X.

- **Unknown filter**

Given inputs and outputs, the problem of finding an unknown filter appears to be overdetermined, so we write $\mathbf{y} \approx \mathbf{Xf}$ where the matrix $\mathbf{X}$ is a matrix of downshifted columns like (6.19). Thus the quadratic form to be minimized is a restatement of equation (6.25) using filter definitions:

$$Q(\mathbf{f}', \mathbf{f}) \quad = \quad (\mathbf{Xf} - \mathbf{y})'(\mathbf{Xf} - \mathbf{y}) \tag{6.27}$$

The solution $\mathbf{f}$ is found just as we found (6.26), and it is the set of simultaneous equations $\mathbf{0} = \mathbf{X}'(\mathbf{Xf} - \mathbf{y})$.

- **Unknown input: deconvolution with a known filter**

For the unknown input problem we put the known filter f_t in a matrix of downshifted columns $\mathbf{F}$. Our statement of wishes is now to find x_t so that $\mathbf{y} \approx \mathbf{F}\mathbf{x}$. We can expect to have trouble finding unknown filter inputs x_t when we are dealing with certain kinds of filters, such as bandpass filters. If the output is zero in a frequency band, we will never be able to find the input in that band and will need to prevent x_t from diverging there. We do this by the statement that we wish $\mathbf{0} \approx \epsilon\,\mathbf{x}$, where ϵ is a parameter that is small and whose exact size will be chosen by experimentation. Putting both wishes into a single, partitioned matrix equation gives

$$\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \approx \begin{bmatrix} \mathbf{r}_1 \\ \mathbf{r}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{y} \\ \mathbf{0} \end{bmatrix} - \begin{bmatrix} \mathbf{F} \\ \epsilon\,\mathbf{I} \end{bmatrix} \mathbf{x} \tag{6.28}$$

To minimize the residuals $\mathbf{r}_1$ and $\mathbf{r}_2$, we can minimize the scalar $\mathbf{r}'\mathbf{r} = \mathbf{r}'_1\mathbf{r}_1 + \mathbf{r}'_2\mathbf{r}_2$. This is

$$\begin{aligned} Q(\mathbf{x}', \mathbf{x}) &= (\mathbf{F}\mathbf{x} - \mathbf{y})'(\mathbf{F}\mathbf{x} - \mathbf{y}) + \epsilon^2 \mathbf{x}'\mathbf{x} \\ &= (\mathbf{x}'\mathbf{F}' - \mathbf{y}')(\mathbf{F}\mathbf{x} - \mathbf{y}) + \epsilon^2 \mathbf{x}'\mathbf{x} \end{aligned} \tag{6.29}$$

We have already solved this minimization in chapter 4 in the frequency domain (beginning from equation (4.16)).

Formally the solution is found just as with equation (6.26), but this solution looks unappealing in practice because there are so many unknowns and because the problem can be solved much more quickly in the Fourier domain. To motivate ourselves to solve this problem in the time domain, we need either to find an approximate solution method that is much faster, or to discover that constraints or time-variable weighting functions are required in some applications.

EXERCISES:

1 Try other lags in (6.19) such as $(0,1,0)'$ and $(0,0,1)'$. Which works best? Why?

2 Using matrix algebra, what value of $\mathbf{x}$ minimizes the **quadratic form** $Q(\mathbf{x}) = (\mathbf{y}-\mathbf{A}\mathbf{x})'\mathbf{M}_{nn}^{-1}(\mathbf{y}-\mathbf{A}\mathbf{x}) + (\mathbf{x}-\mathbf{x}_0)'\mathbf{M}_{xx}^{-1}(\mathbf{x}-\mathbf{x}_0)$? In applications, $\mathbf{x}_0$ is called the prior model, $\mathbf{M}_{xx}$ its **covariance matrix**, and $\mathbf{M}_{nn}$ the noise **covariance matrix**.

3 Let $y(t)$ constitute a complex-valued function at successive integer values of t. Fit $y(t)$ to a least-squares straight line $y(t) \approx \alpha + \beta t$, where $\alpha = \alpha_r + i\alpha_t$

and $\beta = \beta_r + i\beta_t$. Do it two ways: (a) assume α_r, α_t, β_i, and β_r are four independent variables, and (b) assume α, $\overline{\alpha}$, β, and β are independent variables. (Leave the answer in terms of $s_n = \sum_t t^n$.)

4 Ocean tides fit sinusoidal functions of known frequencies quite accurately. Associated with the tide is an earth tilt. A complex time series can be made from the north-south tilt plus $\sqrt{-1}$ times the east-west tilt. The observed complex time series can be fitted to an analytical form $\sum_{j=1}^{N} A_j e^{i\omega_j t}$. Find the set of equations which can be solved for the A_j that gives the best fit of the formula to the data. Show that some elements of the normal equation matrix are sums that can be summed analytically.

5 The general solution to Laplace's equation in cylindrical coordinates (r, θ) for a potential field P which vanishes at $r = \infty$ is given by

$$P(r,\theta) = \Re \sum_{m=0}^{\infty} A_m \frac{e^{im\theta}}{r^{m+1}}$$

Find the potential field surrounding a square object at the origin which is at unit potential. Do this by finding N of the coefficients A_m by minimizing

the squared difference between $P(r,\theta)$ and unity integrated around the square. Give the answer in terms of an inverse matrix of integrals. Which coefficients A_m vanish exactly by symmetry?

6.2. ITERATIVE METHODS

The solution time for simultaneous linear equations grows cubically with the number of unknowns. There are three regimes for solution; which one is applicable depends on the number of unknowns n. For n three or less, we use analytical methods. We also sometimes use analytical methods on matrices of size 4×4 if the matrix contains many zeros. For $n < 500$ we use exact numerical methods such as Gauss reduction. A 1988 vintage workstation solves a 100×100 system in a minute, but a 1000×1000 system requires a week. At around $n = 500$, exact numerical methods must be abandoned and **iterative method**s must be used.

An example of a geophysical problem with $n > 1000$ is a missing seismogram. Deciding how to handle a missing seismogram may at first seem like a question of missing *data*, not excess numbers of *model* points. In fitting wave-field data to a consistent model, however, the missing data is seen to be just more unknowns.

In real life we generally have not one missing seismogram, but many. Theory in 2-D requires that seismograms be collected along an infinite line. Since any data-collection activity has a start and an end, however, practical analysis must choose between falsely asserting zero data values where data was not collected, or implicitly determining values for unrecorded data at the ends of a survey.

A numerical technique known as the "**conjugate-gradient method**" (**CG**) works well for all values of n and is our subject here. As with most simultaneous equation solvers, an exact answer (assuming exact arithmetic) is attained in a finite number of steps. And if n is too large to allow n^3 computations, the CG method can be interrupted at any stage, the partial result often proving useful. Whether or not a partial result actually is useful is the subject of much research; naturally, the results vary from one application to the next.

The simple form of the CG algorithm covered here is a sequence of steps. In each step the minimum is found in the plane given by two vectors: the gradient vector and the vector of the previous step.

6.2.1. Method of random directions and steepest descent

Let us minimize the sum of the squares of the components of the **residual** vector given by

$$\text{residual} \quad = \quad \text{data space} \quad - \quad \text{transform} \quad \text{model space} \tag{6.30}$$

$$\begin{bmatrix} \\ R \\ \\ \end{bmatrix} \quad = \quad \begin{bmatrix} \\ Y \\ \\ \end{bmatrix} \quad - \quad \begin{bmatrix} \\ \mathbf{A} \\ \\ \end{bmatrix} \begin{bmatrix} \\ x \\ \\ \end{bmatrix} \tag{6.31}$$

Fourier-transformed variables are often capitalized. Here we capitalize vectors transformed by the **A** matrix. A matrix such as **A** is denoted by **boldface** print.

A **contour** plot is based on an altitude function of space. The altitude is the dot product $R \cdot R$. By finding the lowest altitude we are driving the residual vector R as close as we can to zero. If the residual vector R reaches zero, then we have solved the simultaneous equations $Y = \mathbf{A}x$. In a two-dimensional world the vector x has two components, (x_1, x_2). A contour is a curve of constant $R \cdot R$ in (x_1, x_2)-space. These contours have a statistical interpretation as contours of uncertainty in (x_1, x_2), given measurement errors in Y.

Starting from $R = Y - \mathbf{A}x$, let us see how a random search direction can be used to try to reduce the residual. Let g be an abstract vector with the same number of components as the solution x, and let g contain arbitrary or random numbers. Let us add an unknown quantity α of vector g to vector x, thereby changing x to $x + \alpha g$. The new residual $R + dR$ becomes

$$\begin{aligned} R + dR &= Y - \mathbf{A}(x + \alpha g) && (6.32) \\ &= Y - \mathbf{A}x - \alpha \mathbf{A}g && (6.33) \\ &= R - \alpha G && (6.34) \end{aligned}$$

We seek to minimize the dot product

$$(R + dR) \cdot (R + dR) = (R - \alpha G) \cdot (R - \alpha G) \quad (6.35)$$

Setting to zero the derivative with respect to α gives

$$\alpha = \frac{(R \cdot G)}{(G \cdot G)} \quad (6.36)$$

Geometrically and algebraically the new residual $R_+ = R - \alpha G$ is perpendicular to the "fitting function" G. (We confirm this by substitution leading to $R_+ \cdot G = 0$.)

In practice, random directions are rarely used. It is more common to use the **gradient** vector. Notice also that a vector of the size of x is

$$g \quad = \quad \mathbf{A}' R \tag{6.37}$$

Notice also that this vector can be found by taking the gradient of the size of the residuals:

$$\frac{\partial}{\partial x'} \, R \cdot R \quad = \quad \frac{\partial}{\partial x'} \, (Y' - x' \mathbf{A}')(Y - \mathbf{A} x) \quad = \quad -\mathbf{A}' \, R \tag{6.38}$$

Descending by use of the gradient vector is called “the method of **steepest descent**."

6.2.2. Conditioning the gradient

Often people do calculations by the method of steepest descent without realizing it. Often a result is improved in a single step, or with a small number of steps, many fewer than the number needed to achieve convergence. This is especially true with images where the dimensionality is huge and where a simple improvement to the adjoint operator is sought. Three-dimensional migration is an example. In these cases it may be worthwhile to make some ad hoc improvements to the gradient that

acknowledge the gradient will be a perturbation to the image **x** and so should probably have an amplitude and spectrum like that of **x**. A more formal mathematical discussion of preconditioning is on page 347.

6.2.3. Why steepest descent is so slow

Before we can understand why the **conjugate-gradient method** is so fast, we need to see why the **steepest-descent method** is so slow. The process of selecting α is called "**line search**," but for a linear problem like the one we have chosen here, we hardly recognize choosing α as searching a line. A more graphic understanding of the whole process is possible in a two-dimensional space where the vector of unknowns x has just two components, x_1 and x_2. Then the size of the residual vector $R \cdot R$ can be displayed with a contour plot in the plane of (x_1, x_2). Visualize a contour map of a mountainous terrain. The gradient is perpendicular to the contours. Contours and gradients are *curved lines*. In the steepest-descent method we start at a point and compute the gradient direction at that point. Then we begin a *straight-line* descent in that direction. The gradient direction curves away from our direction of travel, but we continue on our straight line until we have stopped descending and

are about to ascend. There we stop, compute another gradient vector, turn in that direction, and descend along a new straight line. The process repeats until we get to the bottom, or until we get tired.

What could be wrong with such a direct strategy? The difficulty is at the stopping locations. These occur where the descent direction becomes *parallel* to the contour lines. (There the path becomes horizontal.) So after each stop, we turn 90°, from parallel to perpendicular to the local contour line for the next descent. What if the final goal is at a 45° angle to our path? A 45° turn cannot be made. Instead of moving like a rain drop down the centerline of a rain gutter, we move along a fine-toothed zigzag path, crossing and recrossing the centerline. The gentler the slope of the rain gutter, the finer the teeth on the zigzag path.

6.2.4. Conjugate gradient

In the **conjugate-gradient method**, not a line, but rather a plane, is searched. A plane is made from an arbitrary linear combination of two vectors. One vector will be chosen to be the gradient vector, say g. The other vector will be chosen to be the previous descent step vector, say $s = x_j - x_{j-1}$. Instead of αg we need

a linear combination, say $\alpha g + \beta s$. For minimizing quadratic functions the plane search requires only the solution of a two-by-two set of linear equations for α and β. The equations will be specified here along with the program. (For *nonquadratic* functions a plane search is considered intractable, whereas a line search proceeds by bisection.)

6.2.5. Magic

Some properties of the conjugate-gradient approach are well known but hard to explain. D. G. Luenberger's book, *Introduction to Linear and Nonlinear Programming*, is a good place to look for formal explanations of this magic. (His book also provides other forms of the conjugate-gradient algorithm.) Another helpful book is Strang's *Introduction to Applied Mathematics*. Known properties follow:

1. The **conjugate-gradient method** gets the exact answer (assuming exact arithmetic) in n descent steps (or less), where n is the number of unknowns.

2. Since it is helpful to use the previous step, you might wonder why not use the previous two steps, since it is not hard to solve a three-by-three set of

simultaneous linear equations. It turns out that the third direction does not help: the distance moved in the extra direction is zero.

6.2.6. Conjugate-gradient theory for programmers

Define the solution, the solution step (from one iteration to the next), and the gradient by

$$X = \mathbf{A}\,x \tag{6.39}$$

$$S_j = \mathbf{A}\,s_j \tag{6.40}$$

$$G_j = \mathbf{A}\,g_j \tag{6.41}$$

A linear combination in solution space, say $s+g$, corresponds to $S+G$ in the conjugate space, because $S+G = \mathbf{A}s + \mathbf{A}\mathbf{g} = \mathbf{A}(s+g)$. According to equation (6.31), the residual is

$$R = Y - \mathbf{A}\,x = Y - X \tag{6.42}$$

The solution x is obtained by a succession of steps s_j, say

$$x = s_1 + s_2 + s_3 + \cdots \tag{6.43}$$

The last stage of each iteration is to update the solution and the residual:

$$\begin{array}{ll} \text{solution update:} & x \leftarrow x + s \\ \text{residual update:} & R \leftarrow R - S \end{array}$$

The *gradient* vector g is a vector with the same number of components as the solution vector x. A vector with this number of components is

$$g = \mathbf{A}' R = \text{gradient} \tag{6.44}$$

$$G = \mathbf{A}\, g = \text{conjugate gradient} \tag{6.45}$$

The gradient g in the transformed space is G, also known as the "**conjugate gradient**."

The minimization (6.35) is now generalized to scan not only the line with α, but simultaneously another line with β. The combination of the two lines is a plane:

$$Q(\alpha,\beta) = (R-\alpha G-\beta S)\cdot(R-\alpha G-\beta S) \tag{6.46}$$

The minimum is found at $\partial Q/\partial\alpha = 0$ and $\partial Q/\partial\beta = 0$, namely,

$$0 = G\cdot(R-\alpha G-\beta S) \tag{6.47}$$

$$0 = S\cdot(R-\alpha G-\beta S) \tag{6.48}$$

The solution is

$$\begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \frac{1}{(G\cdot G)(S\cdot S)-(G\cdot S)^2} \begin{bmatrix} (S\cdot S) & -(S\cdot G) \\ -(G\cdot S) & (G\cdot G) \end{bmatrix} \begin{bmatrix} (G\cdot R) \\ (S\cdot R) \end{bmatrix} \tag{6.49}$$

6.2.7. First conjugate-gradient program

The **conjugate-gradient program** can be divided into two parts: an inner part that is used almost without change over a wide variety of applications, and an outer part containing the initializations. Since **Fortran** does not recognize the difference between upper- and lower-case letters, the conjugate vectors G and S in the program are denoted by `gg` and `ss`. The inner part of the conjugate-gradient task is in subroutine `cgstep()`. cgstep

This program was used to produce about 50 figures in this book. The first example of its use is the solution of the 5×4 set of simultaneous equations below. Observe that the "exact" solution is obtained in the last step. Since the data and answers are integers, it is quick to check the result manually.

```
y transpose
```

```
# A step of conjugate-gradient descent.
#
subroutine cgstep( iter,   n, x, g, s,   m, rr, gg, ss)
integer i,         iter,   n,            m
real    x(n), rr(m)     # solution, residual
real    g(n), gg(m)     # gradient, conjugate gradient
real    s(n), ss(m)     # step,     conjugate step
real dot, sds, gdg, gds, determ, gdr, sdr, alfa, beta
if( iter == 0 ) {
        do i= 1, n
                s(i) = 0.
        do i= 1, m
                ss(i) = 0.
        if( dot(m,gg,gg)==0 )   call erexit('cgstep: grad vanishes identically')
        alfa =  dot(m,gg,rr) / dot(m,gg,gg)
        beta = 0.
        }
else {                          # search plane by solving 2-by-2
        gdg = dot(m,gg,gg)      #  G . (R - G*alfa - S*beta) = 0
        sds = dot(m,ss,ss)      #  S . (R - G*alfa - S*beta) = 0
        gds = dot(m,gg,ss)
        determ = gdg * sds - gds * gds + (.00001 * (gdg * sds) + 1.e-15)
        gdr = dot(m,gg,rr)
        sdr = dot(m,ss,rr)
        alfa = ( sds * gdr - gds * sdr ) / determ
        beta = (-gds * gdr + gdg * sdr ) / determ
        }
do i= 1, n                                      # s = model step
        s(i)  = alfa * g(i)  + beta * s(i)
do i= 1, m                                      # ss = conjugate
        ss(i) = alfa * gg(i) + beta * ss(i)
do i= 1, n                                      # update solution
        x(i)  =  x(i) +  s(i)
do i= 1, m                                      # update residual
        rr(i) = rr(i) - ss(i)
return; end

real function dot( n, x, y )
```

```
      3.00      3.00      5.00      7.00      9.00

A transpose
      1.00      1.00      1.00      1.00      1.00
      1.00      2.00      3.00      4.00      5.00
      1.00      0.00      1.00      0.00      1.00
      0.00      0.00      0.00      1.00      1.00
for iter = 0, 4
x    0.43457383  1.56124675  0.27362058  0.25752524
res  0.73055887 -0.55706739 -0.39193439  0.06291389  0.22804642
x    0.51313990  1.38677311  0.87905097  0.56870568
res  0.22103608 -0.28668615 -0.55250990  0.37106201  0.10523783
x    0.39144850  1.24044561  1.08974123  1.46199620
res  0.27836478  0.12766024 -0.20252618  0.18477297 -0.14541389
x    1.00001717  1.00006616  1.00001156  2.00000978
res -0.00009474 -0.00014952 -0.00022683 -0.00029133 -0.00036907
```

```
x    0.99999994  1.00000000  1.00000036  2.00000000
res -0.00000013 -0.00000003  0.00000007  0.00000018 -0.00000015
```

Initialization of the **conjugate-gradient method** typically varies from one application to the next, as does the setting up of the transformation and its adjoint. The problem above was set up with the `matmul()` program given in chapter 5. The program `cgmeth()` below initializes a zero solution and the residual of a zero solution. cgmeth Then it loops over iterations, invoking matrix multiply, conjugate transpose multiply, and the conjugate-gradient stepper. In subroutine `cgmeth()`, the variable `dx` is like g in equation (6.44), and the variable `dr` is like G in equation (6.45).

6.2.8. Preconditioning

Like steepest descent, CG methods can be accelerated if a nonsingular matrix $\mathbf{M}$ with known inverse can be found to approximate $\mathbf{A}$. Then, instead of solving $\mathbf{Ax} \approx \mathbf{y}$, we solve $\mathbf{M}^{-1}\mathbf{Ax} \approx \mathbf{M}^{-1}\mathbf{y} = \mathbf{c}$, which should converge much faster since $\mathbf{M}^{-1}\mathbf{A} \approx \mathbf{I}$. This is called "**preconditioning**."

In my experience the matrix $\mathbf{M}$ is rarely available, except in the crude approxi-

```
# setup of conjugate gradient descent, minimize  SUM rr(i)**2
#                    nx
# rr(i)  = yy(i) - sum aaa(i,j) * x(j)
#                   j=1
subroutine cgmeth( nx,x, nr,yy,rr, aaa, niter)
integer i, iter,   nx,   nr,            niter
real    x(nx), yy(nr), rr(nr), aaa(nr,nx)
temporary real dx(nx), sx(nx), dr(nr), sr(nr)
do i= 1, nx
        x(i) = 0.
do i= 1, nr
        rr(i) = yy(i)
do iter= 0, niter {
        call matmult( 1, aaa, nx,dx, nr,rr)              # dx= dx(aaa,rr)
        call matmult( 0, aaa, nx,dx, nr,dr)              # dr= dr(aaa,dx)
        call cgstep( iter, nx, x,dx,sx, _
                           nr,rr,dr,sr)                # x=x+s; rr=rr-ss
        }
return; end
```

Back

mation of scaling columns, so the unknowns have about equal magnitude. As with signals and images, spectral balancing should be helpful.

EXERCISES:

1 Remove lines from the conjugate-gradient program to convert it to a program that solves simultaneous equations by the method of steepest descent. Per iteration, how many dot products are saved, and how much is the memory requirement reduced?

2 A precision problem can arise with the CG method when many iterations are required. What happens is that R drifts away from $\mathbf{A}r$ and X drifts away from $\mathbf{A}x$. Revise the program `cgmeth()` to restore consistency every twentieth iteration.

6.3. INVERSE NMO STACK

Starting from a hypothetical, ideal, zero-offset model, Figure 5.6 shows synthetic data and the result of adjoint modeling (back projection), which reconstructs an

imperfect model. Inversion should enable us to reconstruct the original model. Let us see how ***back projection*** can be upgraded towards ***inversion***.

Unfortunately, the adjoint operator $\mathbf{N}'$ defined by the subroutine `nmo1()` /prog:nmo1 is not a good operator for seismogram modeling—notice the roughness of the synthetic seismograms in Figure 5.6. This roughness is *not* an inevitable consequence of nearest-neighbor interpolation. It is a consequence of defining the NMO program as a loop over the output space τ. Instead, we can define inverse NMO as a loop over *its* output space, which is not τ but t. This is done in `imo1()` /prog:imo1 and `imospray()` /prog:imospray.

If we plan an upgrade from back projection towards inversion, we must be aware that the accuracy of the original modeling operator can become an issue.

The new seismograms at the bottom of Figure 6.1 show the first four iterations of conjugate-gradient inversion. You see the original rectangle-shaped waveform returning as the iterations proceed. Notice also on the **stack** that the early and late events have unequal amplitudes, but after iteration they are equal, as they began. Mathematically, we can denote the top trace as the model **m**, the synthetic data sig-

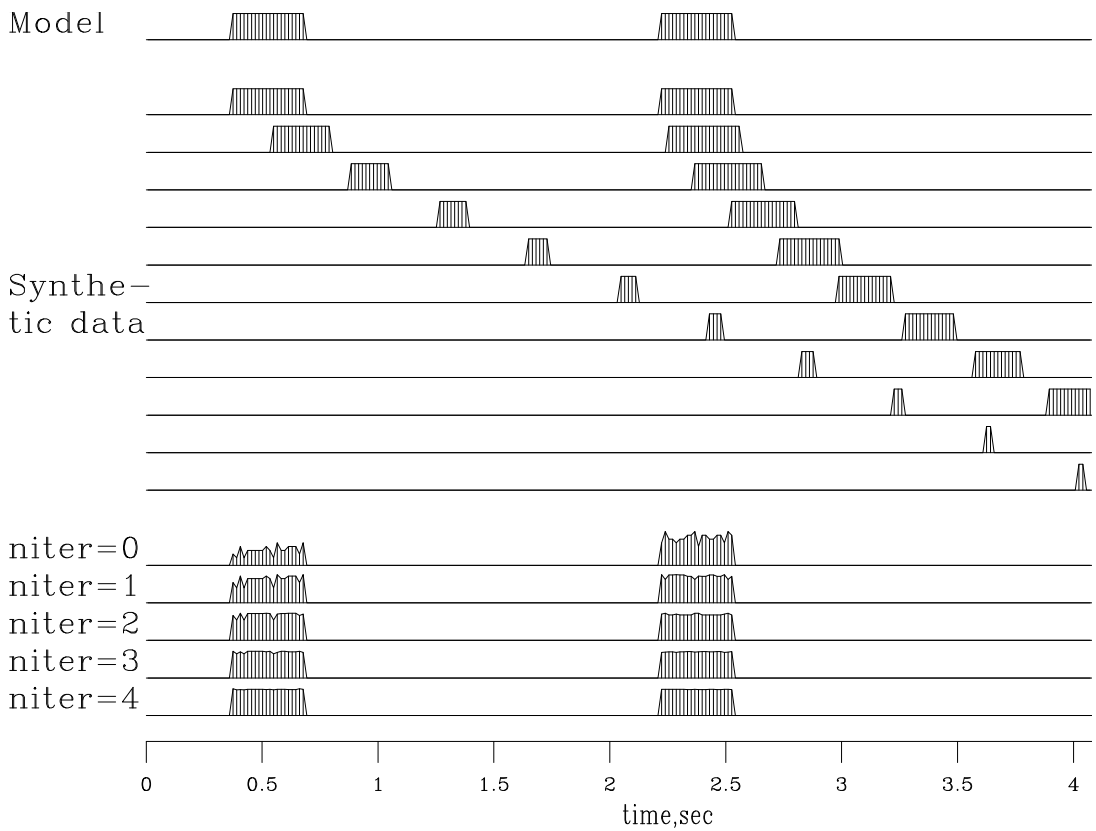


Figure 6.1: Top is a model trace **m**. Next are the synthetic data traces, $\mathbf{d} = \mathbf{Mm}$. Then, labeled `niter=0` is the **stack**, a result of processing by adjoint modeling. Increasing values of `niter` show **x** as a function of iteration count in the regression $\mathbf{d} \approx \mathbf{Mx}$. (Carlos Cunha-Filho) ls-invstack [ER]

```
# NMO stack by inverse of forward modeling
#
subroutine invstack(  nt,model,nx,gather,rr,t0,x0,dt,dx,slow,niter)
integer it, ix, iter, nt,      nx,                              niter
real    t0,x0,dt,dx,slow, gather(nt,nx), rr(nt,nx), model(nt)
temporary real  dmodel(nt), smodel(nt), dr(nt,nx), sr(nt,nx)
do it= 1, nt
        model(it) = 0.0
do it= 1, nt
        do ix= 1, nx
                rr(it,ix) = gather(it,ix)
do iter = 0, niter {
        call imospray( 1,0,slow,x0,dx,t0,dt,nx,nt,dmodel,rr)    # nmo-stack
        call imospray( 0,0,slow,x0,dx,t0,dt,nx,nt,dmodel,dr)    # modeling
        call cgstep(iter,   nt, model, dmodel, smodel, _
                         nt*nx, rr,    dr,     sr)
        }
return; end
```

Back

nals as $\mathbf{d} = \mathbf{Mm}$, and the stack as $\mathbf{M}'\mathbf{d}$. The conjugate-gradient algorithm solves the regression $\mathbf{d} \approx \mathbf{Mx}$ by variation of $\mathbf{x}$, and the figure shows $\mathbf{x}$ converging to $\mathbf{m}$. Since there are 256 unknowns in $\mathbf{m}$, it is gratifying to see good convergence occurring after the first four iterations. The CG subroutine used is `invstack()`, which is just like `cgmeth()` /prog:cgmeth except that the matrix-multiplication operator `matmul()` /prog:matmul has been replaced by `imospray()` /prog:imospray. Studying the program, you can deduce that, except for a scale factor, the output at `niter=0` is identical to the stack $\mathbf{M}'\mathbf{d}$. All the signals in Figure 6.1 are intrinsically the same scale. invstack

This simple inversion is inexpensive. Has anything been gained over conventional stack? First, though we used nearest-neighbor interpolation, we managed to preserve the spectrum of the input, apparently all the way to the Nyquist frequency. Second, we preserved the true amplitude scale without ever bothering to think about (1) dividing by the number of contributing traces, (2) the amplitude effect of NMO stretch, or (3) event truncation.

With depth-dependent velocity, wave fields become much more complex at wide offset. NMO soon fails, but wave-equation forward modeling offers interesting opportunities for inversion.

6.4. MARINE DEGHOSTING

The marine **ghost** presents a problem that is essentially insoluble; but because it is always with us, we need to understand how to do the best we can with it. Even if an **airgun** could emit a perfect impulse, the impulse would reflect from the nearby water surface, thereby giving a second pulse of opposite **polarity**. The energy going down into the earth is therefore a doublet when we would prefer a single pulse. Likewise, **hydrophone**s see the upcoming wave once coming up, and an instant later they see the wave with opposite polarity reflecting from the water surface. Thus the combined system is effectively a second derivative wavelet $(1, -2, 1)$ that is convolved with signals of interest. Our problem is to remove this wavelet by **deconvolution**. It is an omnipresent problem and is cleanly exposed on marine data where the water bottom is hard and deep.

Theoretically, a double integration of the second derivative gives the desired pulse. A representation in the discrete time domain is the product of $(1-Z)^2$ with $1+2Z+3Z^2+4Z^3+5Z^4+\cdots$, which is 1. Double integration amounts to spectral division by $-\omega^2$. Mathematically the problem is that $-\omega^2$ vanishes at $\omega = 0$. In practice the problem is that dividing by ω^2 where it is small amplifies noises at those low frequencies. (Inversion theorists are even more frustrated because they

are trying to create something like a velocity profile, roughly a step function, and they need to do something like a third integration.) Old nuts like this illustrate the dichotomy between theory and practice.

Chapter 4 provides a theoretical solution to this problem in the Fourier domain. Here we will express the same concepts in the time domain. Define as follows:

y_t	Given data.
b_t	Known filter.
x_t	Excitation (to be found).
$n_t = y_t - x_t * b_t$	Noise: data minus filtered excitation.

With Z-transforms the problem is given by $Y(Z) = B(Z)X(Z) + N(Z)$. Our primary wish is $N \approx 0$. Our secondary wish is that X not be infinity as $X = Y/B$ threatens. This second wish is expressed as $\epsilon X \approx 0$ and is called "stabilizing" or "**damping**." In the Fourier domain the wishes are

$$Y \approx BX \tag{6.50}$$

$$0 \approx \epsilon X \tag{6.51}$$

The formal expression of the **regression** is

$$\min_{X} \; (\, ||Y - BX|| \, + \, \epsilon^2 ||X|| \,) \tag{6.52}$$

In the time domain the regression is much more explicit:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \approx \begin{bmatrix} 1 & . & . & . & . & . & . \\ -2 & 1 & . & . & . & . & . \\ 1 & -2 & 1 & . & . & . & . \\ . & 1 & -2 & 1 & . & . & . \\ . & . & 1 & -2 & 1 & . & . \\ . & . & . & 1 & -2 & 1 & . \\ . & . & . & . & 1 & -2 & 1 \\ \epsilon & . & . & . & . & . & . \\ . & \epsilon & . & . & . & . & . \\ . & . & \epsilon & . & . & . & . \\ . & . & . & \epsilon & . & . & . \\ . & . & . & . & \epsilon & . & . \\ . & . & . & . & . & \epsilon & . \\ . & . & . & . & . & . & \epsilon \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix} \tag{6.53}$$

```
subroutine ident( adj, add, epsilon, n, pp,    qq   )
integer i,        adj, add,          n
real                               epsilon,    pp(n), qq(n)  # equivalence (pp,qq) OK
if( adj == 0 ) {
        if( add == 0 ) { do i=1,n {  qq(i) =          epsilon * pp(i) } }
        else           { do i=1,n {  qq(i) = qq(i) + epsilon * pp(i) } }
        }
else {  if( add == 0 ) { do i=1,n {  pp(i) =          epsilon * qq(i) } }
        else           { do i=1,n {  pp(i) = pp(i) + epsilon * qq(i) } }
        }
return; end
```

Back

where "·" denotes a zero. Since it is common to add $\epsilon \mathbf{I}$ to an operator to stabilize it, I prepared subroutine `ident()` for this purpose. It is used so frequently that I coded it in a special way to allow the input and output to overlie one another. ident

We can use any convolution routine we like, but for simplicity, I selected `contrunc()` so the output would be the same length as the input. The two operators `ident()` and `contrunc()` could be built into a new operator. I found it easier to simply cascade them in the deghosting subroutine `deghost()` below. deghost

6.4.1. Synthetics

I made some synthetic marine data and added 5% noise. This, along with an attempted deconvolution, is shown in Figure 6.2. The plot in Figure 6.2 is for the value of ϵ that I subjectively regarded as best. The result is pleasing because the doublets tend to be converted to impulses. Unfortunately, the low-frequency noise in x_t is significantly stronger than that in y_t, as expected.

Taking ϵ larger will decrease $||X||$ but increase the explicit noise $||Y - BX||$. To decrease the explicit noise, I chose a tiny value of $\epsilon = .001$. Figure 6.3 shows the result. The explicit noise n_t appears to vanish, but the low-frequency noise implicit

```
# deghost:                    min       |rrtop|  =  | y - bb (contrunc) xx |
#                              x        |rrbot|     | 0 - epsilon  I    xx |
subroutine deghost( eps, nb,bb, n, yy, xx, rr, niter)
integer iter,            nb,    n,             niter
real    bb(nb), yy(n), eps              # inputs.  typically bb=(1,-2,1)
real    xx(n), rr(n+n)                  # outputs.
temporary real dx(n), sx(n), dr(n+n), sr(n+n)
call zero( n, xx)
call copy( n, yy, rr(1  ))          # top half of residual
call zero( n    , rr(1+n))          # bottom   of residual
do iter= 0, niter {
        call contrunc(1,0,1,nb,bb, n,dx,n,rr); call ident(1,1,eps, n,dx,rr(1+n))
        call contrunc(0,0,1,nb,bb, n,dx,n,dr); call ident(0,0,eps, n,dx,dr(1+n))
        call cgstep( iter,  n,xx,dx,sx, _
                          n+n,rr,dr,sr)
        }
return; end
```

Back

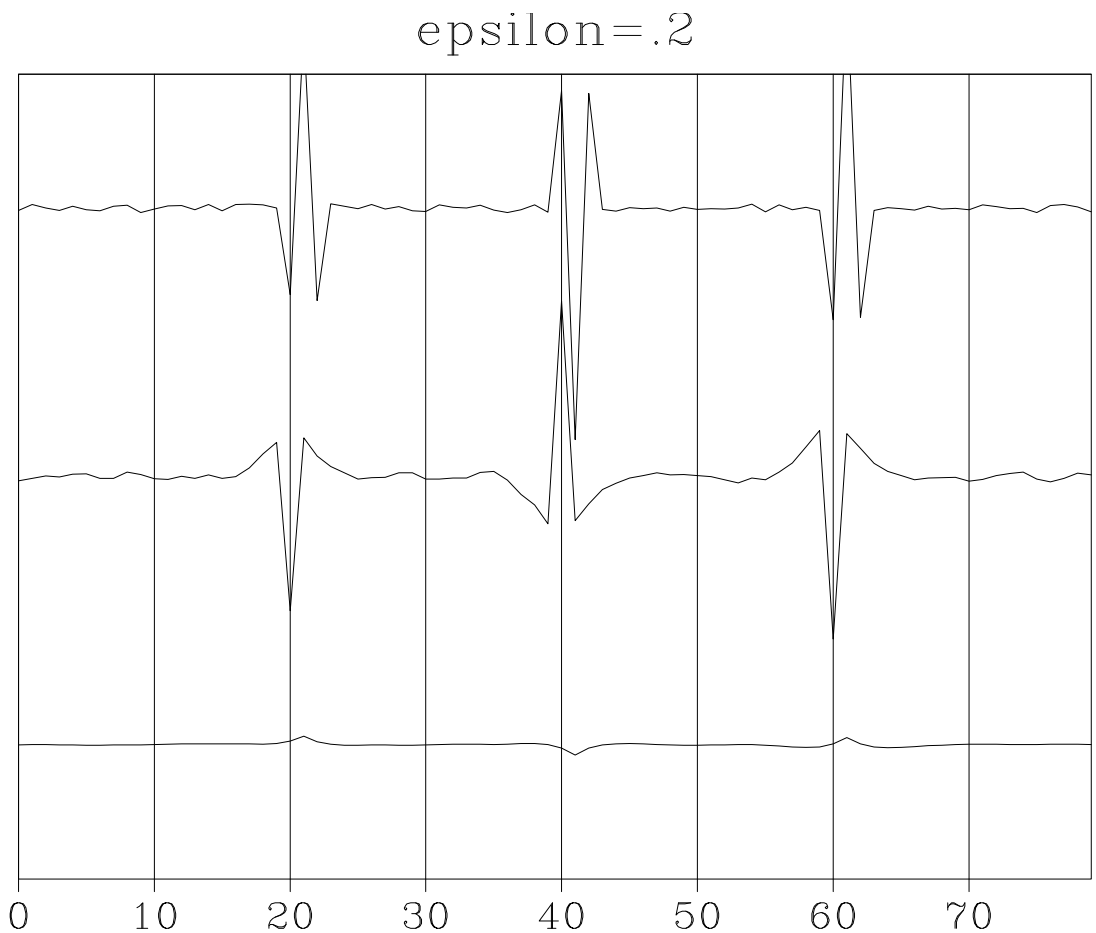
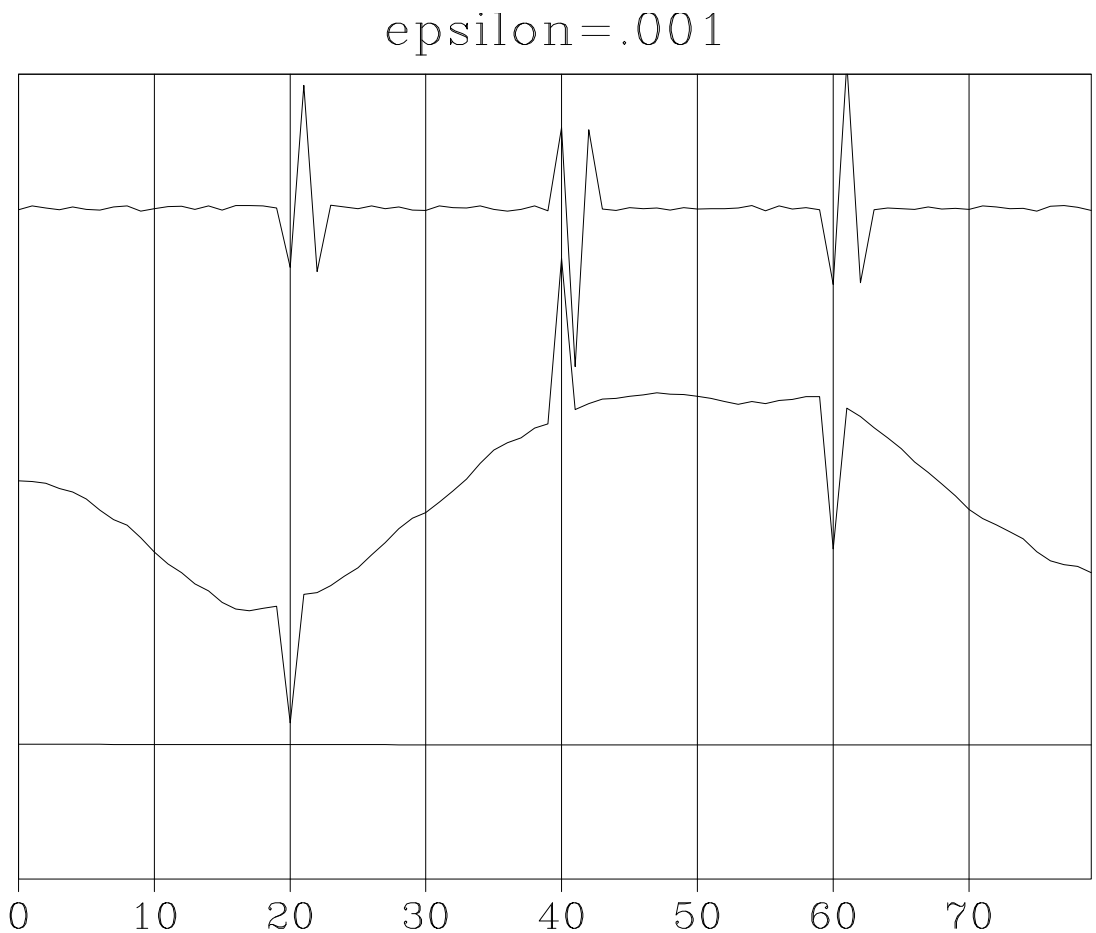


Figure 6.2: Top is the synthetic data y_t. Middle is the deconvolved data x_t. Bottom is the noise n_t. This choice of $\epsilon = .2$ gave my preferred result. ls-syn+.2 [ER]

Figure 6.3: As before, the signals from top to bottom are y_t, x_t, and n_t. Choosing a small $\epsilon = .001$ forces the noise mostly into the output x_t. Thus the noise is essentially implicit. ls-syn+.001 [ER]



to the deconvolved output x_t has grown unacceptably.

Finally, I chose a larger value of $\epsilon = .5$ to allow more noise in the explicit n_t, hoping to get a lower noise implicit to x_t. Figure 6.4 shows the result. Besides the growth of the explicit noise (which is disturbingly correlated to the input), the deconvolved signal has the same unfortunate wavelet on it that we are trying to remove from the input.

Results of a field-data test shown in Figure 6.5 do not give reason for encouragement.

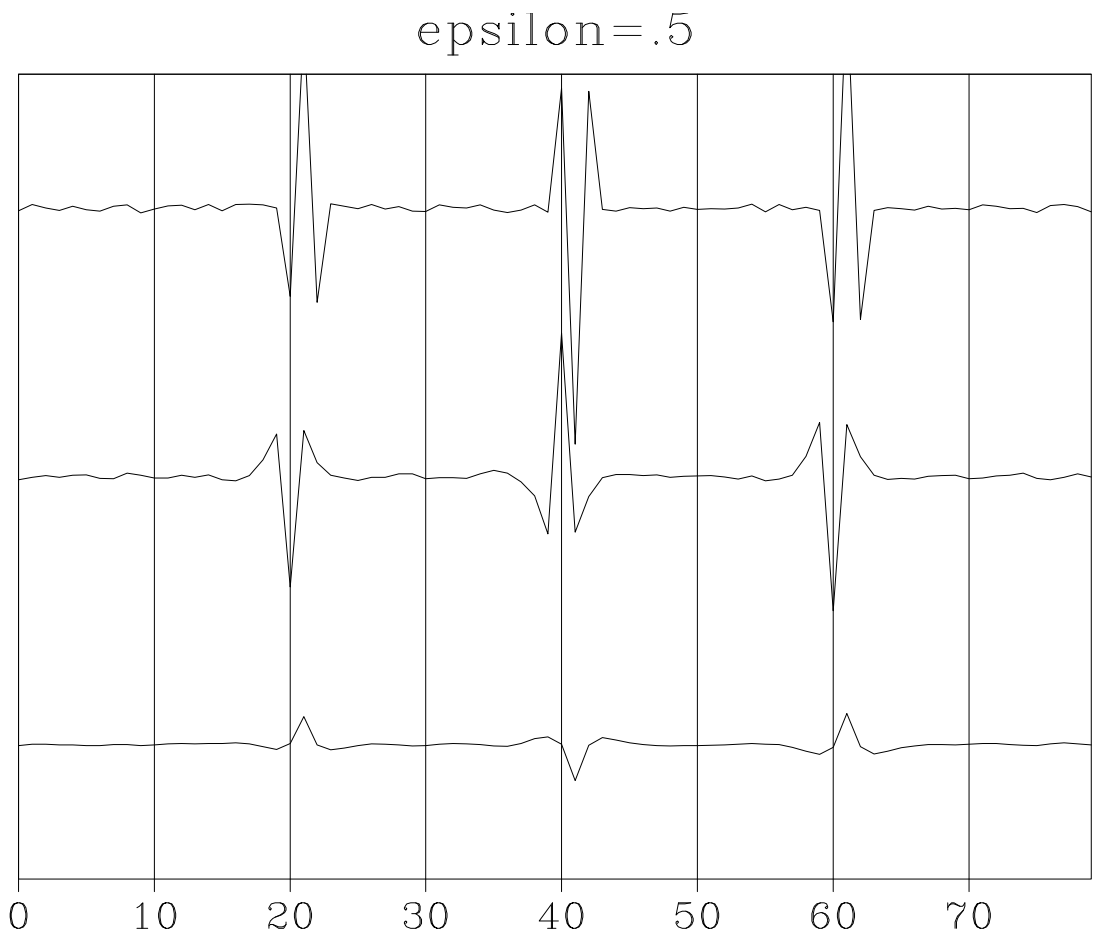
In conclusion, all data recording has an implicit filter, and this filter is arranged to make the data look good. Application of a theoretical filter, such as ω^{-2}, may achieve some theoretical goals, but it does not easily achieve practical goals.

EXERCISES:

1 The $(1, -2, 1)$ **signature** is an oversimplification. In routine operations the hydrophones are at a depth of 7-10 meters and the airgun is at a depth of 5-6 meters. Assuming a sampling rate of .004 s (4 milliseconds) and a water velocity of 1500 m/s, what should the wavelet be?

2 Rerun the figures with the revised wavelet of the previous exercise.

Figure 6.4: Noise essentially explicit. $\epsilon = .5$. ls-syn+.5 [ER]



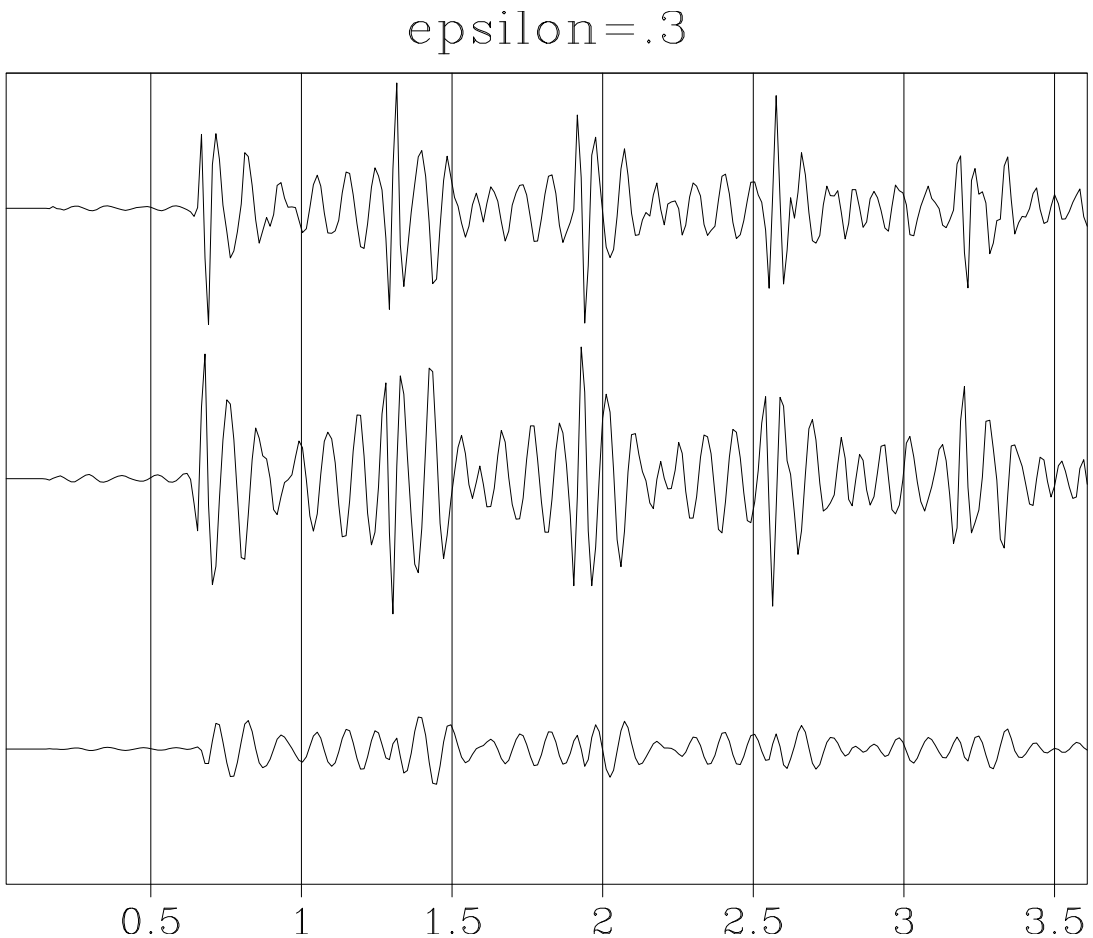


Figure 6.5: Field-data test. As before, the signals from top to bottom are y_t, x_t, and n_t. ls-field+.3 [ER]

6.5. CG METHODOLOGY

The conjugate-gradient method is really a family of methods. Mathematically these algorithms all converge to the answer in n (or fewer) steps where there are n unknowns. But the various methods differ in numerical accuracy, treatment of underdetermined systems, accuracy in treating ill-conditioned systems, space requirements, and numbers of dot products. I will call the method I use in this book the "**book3**" method. I chose it for its clarity and flexibility. I caution you, however, that among the various CG algorithms, it may have the least desirable numerical properties.

The "**conjugate-gradient method**" was introduced by **Hestenes** and Stiefel in 1952. A popular reference book, "Numerical Recipes," cites an algorithm that is useful when the weighting function does not factor. (Weighting functions are not easy to come up with in practice, and I have not found any examples of **nonfactorable weighting functions** yet.) A high-quality program with which my group has had good experience is the Paige and Saunders LSQR program. A derivative of the LSQR program has been provided by Nolet. A disadvantage of the **book3** method is that it uses more auxiliary memory vectors than other methods. Also, you have to tell the **book3** program how many iterations to make.

There are a number of **nonlinear optimization** codes that reduce to **CG** in the limit of a quadratic function.

According to Paige and Saunders, accuracy can be lost by explicit use of vectors of the form $\mathbf{A}'\mathbf{A}x$, which is how the **book3** method operates. An algorithm with better numerical properties, invented by **Hestenes** and Stiefel, can be derived by algebraic rearrangement. This rearrangement (adapted from Paige and Saunders by Lin Zhang) for the problem $\mathbf{A}x \approx Y$ is

- Set $x = 0$, $R = Y$, $g = \mathbf{A}'Y$, $s = g$ and $\gamma^- = \|g\|^2$.
- For each iteration, repeat the following:

$$\begin{aligned} S &= \mathbf{A}s \\ \alpha &= \gamma^- / \|S\|^2 \\ x &= x + \alpha s \\ R &= R - \alpha S \\ g &= \mathbf{A}'R \\ \gamma &= \|g\|^2 \\ \beta &= \gamma / \gamma^- \end{aligned}$$

$$\begin{aligned} \gamma^- &= \gamma \\ s &= g + \beta s \end{aligned}$$

where $\|v\|$ stands for the L_2 norm of vector v. A program that implements this algorithm in a manner consistent with a blending of `cgmeth()` /prog:cgmeth and `cgstep()` /prog:cgstep is `hestenes()`. hestenes I have not used the **Hestenes** and Stiefel version of the CG method for the tutorial programs in this book because I wish to isolate features of geophysical analysis from features of solution by the CG method. The blending of geophysical features with solution details is worsened by the combination of **Fortran** and this improved version of the **CG** algorithm.

6.5.1. Programming languages and this book

A valuable goal is to isolate the CG solving program from all physical aspects of the linear operator. We wish to abstract the CG solver, to put it in a compiled subroutine library where we can feed it inputs and get back the outputs, and never again to see the internals of how it works (like the fast Fourier transform). Unfortunately, the primitive nature of the **Fortran-Ratfor** language does not appear to allow this

```
subroutine hestenes( nx,x, nr,yy,rr, aaa, niter)
integer i, iter,     nx,   nr,            niter
real    alpha, beta, gamma, gammam
real    dot
real    x(nx), yy(nr), rr(nr), aaa(nr,nx)
temporary real g(nx), s(nx), ss(nr)
do i= 1, nx
        x(i) = 0.
do i= 1, nr
        rr(i) = yy(i)
call matmult( 1, aaa, nx,g, nr,yy)
do i= 1, nx
        s(i) = g(i)
gammam = dot(nx,g,g)
do iter= 0, niter {
        call matmult( 0, aaa, nx,s, nr,ss)
        alpha = gammam / dot(nr,ss,ss)
        do i = 1, nx
                x(i) = x(i) + alpha*s(i)
        do i = 1, nr
                rr(i) = rr(i) - alpha*ss(i)
        call matmult( 1, aaa, nx,g, nr,rr)
        gamma = dot(nx,g,g)
        beta = gamma / gammam
        gammam = gamma
        do i = 1, nx
                s(i) = g(i) + beta*s(i)
        }
return; end
```

Back

abstraction. The reason is that the CG program needs to call your linear operator routine and, to do this, it needs to know not only the name of your routine but how to supply its arguments. (I recall from years ago a **Fortran** where subroutines could have several entry points. This might help.) Thus, everywhere in this book where I solve a model-fitting problem, we must see some of the inner workings of CG. To keep this from becoming too objectionable, I found the nonstandard CG method we have been using and coupled it with Bill **Harlan**'s idea of isolating its inner workings into `cgstep()`. Because of this we can see complete code for many examples, and the code is not awfully cluttered. Unfortunately, my CG is not Hestenes' CG.

In many of the **Fortran** codes you see in this book it is assumed that the abstract vector input and vector output correspond to physical one-dimensional arrays. In real life, these abstract vectors are often matrices, or matrices in special forms, such as windows on a wall of data (nonpacked arrays), and they may contain complex numbers. Examining equation (6.53), we notice that the space of residuals for a damped problem is composed of two parts, the residual of the original problem and a part $\epsilon\mathbf{x}$ the size of the unknowns. These two parts are packed, somewhat uncomfortably, into the abstract residual vector.

A linear operator really consists of (at least) four subroutines, one for applying the operator, one for its adjoint, one for a dot product in the space of inputs, and one for a dot product in the space of outputs. Modern programming theory uses the terms “data abstraction” and “**object-oriented programming** (**OOP**)” to describe methods and languages that are well suited to the problems we are facing here. The linear-operator object is what the CG solver needs to be handed, along with an instance of the input abstract vector and a pointer to space for the output vector. (The linear-operator object, after it is created, could also be handed to a universal dot-product test routine. With **Fortran** I write a separate dot-product test program for each operator.)

Another abstraction that **Fortran** cannot cope with is this: the CG program must allocate space for the gradient and past-steps vectors. But the detailed form of these abstract vectors should not be known to the CG program. So the linear-operator object requires four more routines (called “methods" in OOP) that the CG routine uses to allocate and free memory (to *create* and *destroy objects* from the physical space of inputs and outputs). In this way OOP allows us to isolate concepts, so that each concept need only be expressed once. A single version of a concept can thus be reused without being replicated in a form blended with other concepts.

As I am going along generating examples for this book, and as the examples get more complicated, I am wondering just where I will drop the idea of exhibiting complete codes. Obviously, if I switched from **Fortran** to a more modern language, such as **C++**, I could get further. The disadvantages of C++ are that I am not experienced in it, few of my readers will know it, and its looping statements are cluttered and do not resemble mathematics. Instead of `do i=1,n`, C and C++ use `for( i=0; i<=n; i++)`. It would be fun to do the coding in a better way, but for now, I am having more fun identifying new problems to solve.

6.6. References

Gill, P.E., Murray, W., and Wright, M.H., 1981, Practical optimization: Academic Press.

Gorlen, K.E., Orlow, S.M., and Plexico, P.S., 1990, Data abstraction and object-oriented programming in C++: J. Wiley.

Hestenes, M.R., and Stiefel, E., 1952, Methods of conjugate gradients for solving linear systems: J. Res. Natl. Bur. Stand., **49**, 409-436.

Luenberger, D.G., 1973, Introduction to linear and nonlinear programming: Addison-Wesley.

Nolet, G., 1985, Solving or resolving inadequate and noisy tomographic systems: J. Comp. Phys., **61**, 463-482.

Paige, C.C., and Saunders, M.A., 1982a, LSQR: an algorithm for sparse linear equations and sparse least squares: Assn. Comp. Mach. Trans. Mathematical Software, **8,** 43-71.

Paige, C.C., and Saunders, M.A., 1982b, Algorithm 583, LSQR: sparse linear equations and least squares problems: Assn. Comp. Mach. Trans. Mathematical Software, **8,** 195-209.

Press, W.H. et al., 1989, Numerical recipes: the art of scientific computing: Cambridge University Press.

Strang, G., 1986, Introduction to applied mathematics: Wellesley-Cambridge Press.

Chapter 7

Time-series analysis

In chapter 5 we learned about many operators and how adjoints are a first approximation to inverses. In chapter 6 we learned how to make inverse operators from adjoint operators by the least-squares (LS) method using conjugate gradients (CG).

The many applications of least squares to the convolution operator constitute the subject known as "**time-series analysis**." In this chapter we examine applications of time-series analysis to reflection seismograms. These applications further illuminate the theory of least squares in the area of **weighting function**s and stabilization.

In the simplest applications, solutions can be most easily found in the frequency domain. When complications arise, it is better to use the time domain, directly applying the convolution operator and the method of least squares.

A first complicating factor in the frequency domain is a required boundary in the time domain, such as that between past and future, or requirements that a filter be nonzero in a stated time interval. Another factor that attracts us to the time domain rather than the Fourier domain is weighting functions. As we saw in the beginning of chapter 6 weighting functions are appropriate whenever a signal or image amplitude varies from place to place. Most of the literature on **time-series analysis** applies to the limited case of uniform weighting functions. Such time series are said to be "stationary." This means that their statistical properties do not change in time. In real life, particularly in the analysis of echos, signals are never stationary in time and space. A **stationarity** assumption is a reasonable starting assumption, but we should know how to go beyond it so we can take advantage of the many

opportunities that do arise. In order of increasing difficulty in the frequency domain are the following complications:

1. A time boundary such as between past and future.
2. More time boundaries such as delimiting a filter.
3. Nonstationary signal, i.e., time-variable weighting.
4. Time-axis stretching such as normal moveout.

We will not have difficulty with any of these complications here because we will stay in the time domain and set up and solve optimization problems using the conjugate-gradient method. Thus we will be able to cope with great complexity in problem formulation and get the right answer without approximations. By contrast, analytic or partly analytic methods can be more economical, but they generally solve somewhat different problems than those given to us by nature.

7.1. SHAPING FILTER

A shaping filter is a simple least-squares filter that converts one waveform to another. Shaping filters arise in a variety of contexts. In the simplest context, predicting one infinitely long time series from another, the shaping filter can be found in the Fourier domain.

7.1.1. Source waveform and multiple reflections

Figure 7.1 shows some reflection seismic data recorded at nearly vertical incidence from an **arctic ice** sheet. Apparently the initial waveform is somewhat complex, but the water-bottom reflection does not complicate it further. You can confirm this by noticing the water-bottom **multiple reflection**, i.e., the wave that bounces first from the water bottom, then from the water surface, and then a second time from the water bottom. This multiple reflection is similar to but has **polarity** opposite to the shape of the primary water-bottom reflection. (The opposite polarity results from the reflection at the ocean surface, where the acoustic pressure, the sum of the downgoing wave plus the upgoing wave, vanishes.)

Other data in water of similar depth shows a different reflection behavior. The

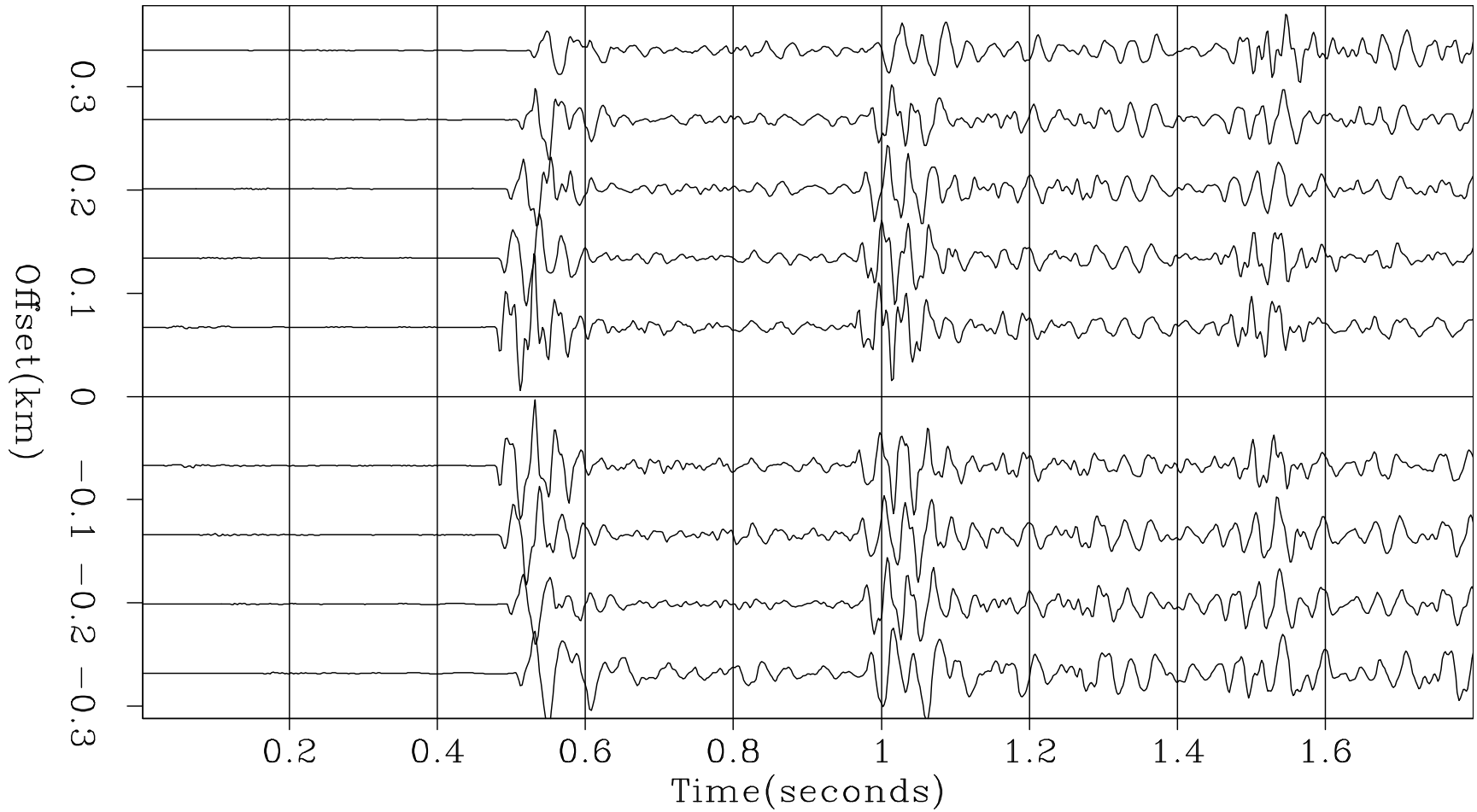


Figure 7.1: Some of the inner offset seismograms from Arctic dataset 24 (Yilmaz and Cumro). tsa-wz24 [NR]

bottom gives back not a single reflection, but a train of reflections. Let this train of reflections from the ocean *floor* be denoted by $F(Z)$. Instead of looking like $-F(Z)$, the first **multiple reflection** can look like $-F(Z)^2$. The ray sketch in Figure 7.2 shows a simple multiple reflection. There is only one water-bottom path, but there are two paths to a slightly deeper layer. I will call the first arrival the soft-mud arrival and the second one the **mudstone** arrival. If these two arrivals happen to have the same strength, an expression for $F(Z)$ is $1+Z$. The expression for the first multiple is $-F(Z)^2 = -(1+Z)^2 = -1+2Z-Z^2$ where the $2Z$ represents the two paths in Figure 7.2. The waveform of the second water bottom multiple is $(1-Z)^3$ in which the mudstone would be three times as strong as the soft mud. In the nth wave train the mudstone is n times as strong as the soft mud. Figure 7.3 is a textbook quality example of this simple concept.

Figures 7.3 and 7.1 illustrate how arctic data typically contrasts with data from temperate or tropic regions. The arctic water-bottom reflection is generally hard, indicating that the bottom is in a constant state of erosion from the scraping of the ice floes and the carrying away of sediments by the bottom currents. In temperate and tropical climates, the bottom is often covered with soft sediments: the top layer is unconsolidated mud, and deeper layers are mud consolidated into mudstone.

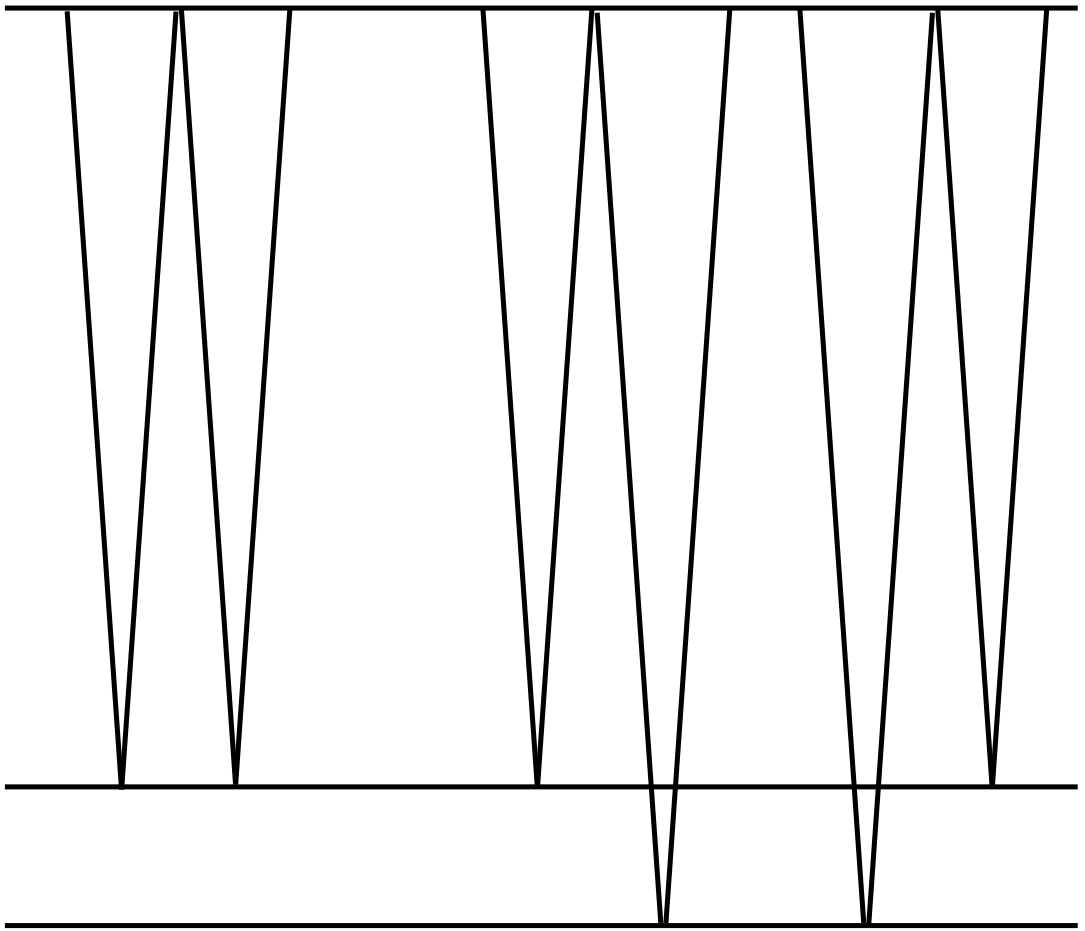


Figure 7.2: Water bottom soft-mud multiple (left) and similar travel times to mudstone (center and right). tsa-peg [NR]

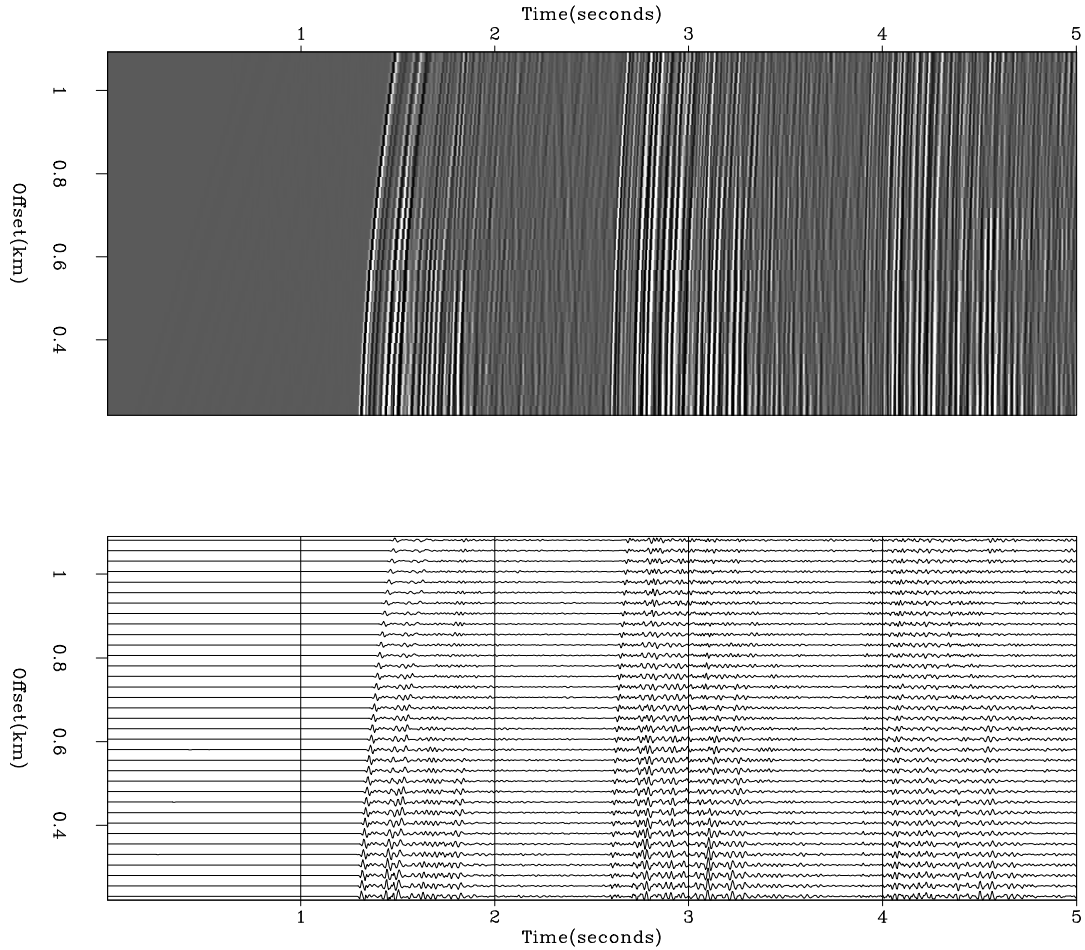


Figure 7.3: Two displays of a common-shot collection of seismograms from offshore Crete (Yilmaz and Cumro dataset 30). Top display is called "raster" and bottom display "wiggle." Raw data scaled by t^2. tsa-wz30 [NR]

Now we devise a simple mathematical model for the multiple reflections in Figures 7.1 and 7.3. There are two unknown waveforms, the source waveform $S(Z)$ and the ocean-floor reflection $F(Z)$. The water-bottom primary reflection $P(Z)$ is the convolution of the source waveform with the water-bottom response; so $P(Z) = S(Z)F(Z)$. The first multiple reflection $M(Z)$ sees the same source waveform, the ocean floor, a minus one for the free surface, and the ocean floor again. Thus the observations $P(Z)$ and $M(Z)$ as functions of the physical parameters are

$$P(Z) = S(Z)\,F(Z) \tag{7.1}$$

$$M(Z) = -S(Z)\,F(Z)^2 \tag{7.2}$$

In Figure 7.1 it appears that $F(Z)$ is nearly an impulse, whereas Figure 7.3 is dominated by the nonimpulsive aspect of $F(Z)$. Algebraically the solutions of equations (7.1) and (7.2) are

$$F(Z) = -M(Z)/P(Z) \tag{7.3}$$

$$S(Z) = -P(Z)^2/M(Z) \tag{7.4}$$

These solutions can be computed in the Fourier domain. The difficulty is that the divisors in equations (7.3) and (7.4) can be zero, or small. This difficulty can be

attacked by using a positive number ϵ to **stabilize** it. Equation (7.3), for example, could be written

$$F(Z) \quad = \quad - \frac{M(Z)P(1/Z)}{P(Z)P(1/Z)+\epsilon} \tag{7.5}$$

We can easily understand what this means as ϵ tends to infinity, where, because of the $1/Z$, the **matched filter** has a noncausal response. Thus, although the ϵ stabilization seems nice, it apparently produces a nonphysical model. For ϵ large or small, the time-domain response could turn out to be of much greater duration than is physically reasonable. This should not happen with perfect data, but in real life, data always has a limited spectral band of good quality.

Functions that are rough in the frequency domain will be long in the time domain. This suggests making a short function in the time domain by local smoothing in the frequency domain. Let the notation $< \cdots >$ denote smoothing by local averaging. Thus we can specify filters whose time duration is not unreasonably long by revising equation (7.5) to

$$F(Z) \quad = \quad - \frac{< M(Z)P(1/Z) >}{< P(Z)P(1/Z)+\epsilon >} \tag{7.6}$$

where it remains to specify the method and amount of smoothing.

These time-duration difficulties do not arise in a time-domain formulation. First express (7.3) and (7.4) as

$$P(Z)F(Z) \approx -M(Z) \tag{7.7}$$

$$M(Z)S(Z) \approx -P(Z)^2 \tag{7.8}$$

To imagine these in the time domain, refer back to equation (**??**). Think of **Pf** $\approx$ **m** where **f** is a column vector containing the unknown sea-floor filter, **m** is a column vector containing the portion of a seismogram in Figure 7.1 labeled "multiple," and **P** is a matrix of down-shifted columns, each column being the same as the signal labeled "primary" in Figure 7.1. The time-domain solutions are called "**shaping filters**." For a simple filter of two coefficients, f_0 and f_1, we solved a similar problem, equation (6.19), theoretically. With longer filters we use numerical methods.

In the time domain it is easy and natural to limit the duration and location of the nonzero coefficients in $F(Z)$ and $S(Z)$. The required program for this task is `shaper()`, which operates like `cgmeth()` /prog:cgmeth and `invstack()` /prog:invstack except that the operator needed here is `contran()` /prog:contran. shaper

The goal of finding the filters $F(Z)$ and $S(Z)$ is to best model the multiple reflections so that they can be subtracted from the data, enabling us to see what

```
# shaping filter
# minimize  SUM rr(i)**2  by finding ff and rr where
#
# rr = yy - xx (convolve) ff
#
subroutine shaper( nf,ff, nx,xx, ny, yy, rr, niter)
integer i, iter,   nf,    nx,    ny,         niter
real    ff(nf), xx(nx), yy(ny), rr(ny)
temporary real  df(nf), dr(ny), sf(nf), sr(ny)
if( ny != nx+nf-1)   call erexit('data length error')
do i= 1, nf
        ff(i) = 0.
do i= 1, ny
        rr(i) =  yy(i)
do iter= 0, niter {
        call contran( 1, 0, nx,xx,  nf,df,  rr)         # df=xx*rr
        call contran( 0, 0, nx,xx,  nf,df,  dr)         # dr=xx*df
        call cgstep( iter, nf,ff,df,sf,  ny,rr,dr,sr)   # rr=rr-dr; ff=ff+df
        }
return; end
```

Back

primary reflections have been hidden by the multiples. An important practical aspect is merging the analysis of many seismograms (see exercises).

Typical data includes not only that shown in Figures 7.1 and 7.3, but also wider source-receiver separation, as well as many other nearby shots and their receivers. Corrections need to be made for hyperbolic traveltime resulting from lateral separation between shot and receiver. Diffractions are a result of lateral imperfections in the generally flat sea floor. The spatial aspects of this topic are considered at great length in IEI. We will investigate them here in only a limited way.

7.1.2. Shaping a ghost to a spike

An exasperating problem in seismology is the "**ghost**" problem, in which a waveform is replicated a moment after it occurs because of a strong nearby reflection. In marine seismology the nearby reflector is the sea surface. Because the sea surface is near both the **airgun** and the **hydrophone**s, it creates two ghosts. Upgoing and downgoing waves at the sea surface have opposite polarity because their pressures combine to zero at the surface. Thus waves seen in the hydrophone encounter the ghost operator $g_t = (1, 0, 0, \cdots, -1)$ twice, once for the surface near the source and

once for the surface near the hydrophone. The number of zeros is typically small, depending on the depth of the device. The sound receivers can be kept away from surface-water wave noise by positioning them deeper, but that extends the ghost delay; and as we will see, this particular ghost is very hard to eliminate by processing. For simplicity, let us analyze just one of the two ghosts. Take it to be $G(Z) = 1 - Z^2$. Theoretically, the inverse is of infinite duration, namely, $(1, 0, 1, 0, 1, 0, 1, 0, 1, \cdots)$.

Since an infinitely long operator is not satisfactory, I used the program `shaper()` above to solve a least-squares problem for an antighost operator of finite duration. Since we know that the least-*squares* method abhors large errors and thus tends to equalize them, we should be able to guess the result.

The filter $(.9, .0, .8, .0, .7, .0, .6, .0, .5, .0, .4, .0, .3, .0, .2, .0, .1)$, when convolved with $(1, 0, -1)$, produces the desired **spike** (impulse) along with equal squared errors of .01 at each output time. Thus, the least-squares filter has the same problem as the analytical one—it is very long. This disappointment can be described in the Fourier domain by the many zeros in the spectrum of $(1, 0, -1)$. Since we cannot divide by zero, we should not try to divide by $1 - Z^n$, which has zeros uniformly distributed on the unit circle. The method of least squares prevents disaster, but it cannot perform miracles.

I consider ghosts to be a problem in search of a different solution. Ghosts also arise when seismograms are recorded in a shallow borehole. As mentioned, the total problem generally includes many waveforms propagating in more than one direction; thus it is not as one-dimensional as it may appear in Figures 7.3 and 7.1, in which I did not display the wide-offset signals.

EXERCISES:

1 What inputs to subroutine `shaper()` /prog:shaper give the filter $(.9, 0, .8, \cdots .1)$ mentioned above?

2 Figure 7.1 shows many seismograms that resemble each other but differ in the x location of the receiver. Sketch the overdetermined simultaneous equations that can be used to find the best-fitting source function $S(Z)$, where $M_x(Z)S(Z) \approx P_x(Z)^2$ for various x.

3 Continue solving the previous problem by defining a `contranx()` subroutine that includes several signals going through the same filter. In order to substitute your `contranx()` into `shaper()` /prog:shaper to replace `contran()` /prog:contran, you will need to be sure that the output and the *filter* are adjoint

(not the output and the *input*). Suggestion: define `real xx(nt,nx)`, etc.

7.2. SYNTHETIC DATA FROM FILTERED NOISE

A basic way to describe the random character of signals is to model them by putting random numbers into a filter. Practical work often consists of the reverse: deducing the filter and deconvolving it to see the input.

7.2.1. Gaussian signals versus sparse signals

Most theoretical work is based on random numbers from a Gaussian probability function. The basic theoretical model is that at *every* time point a **Gaussian** random number is produced. In real life we do observe such signals, but we also observe signals with less frequent noise bursts. Such signals, called "**sparse signal**s" or "**bursty signal**s," can be modeled in many ways, two of which are (1) that many points can have zero value (or a value that is smaller than expected from a Gaussian); and (2) that the Gaussian probability function describes the many smaller values, but some larger values also occur from time to time.

It turns out that the Gaussian probability function generates more cryptic signals than any other probability function. It also turns out that theory is best developed for the Gaussian case. Thus, Gaussian theory, which is the most pessimistic, tends to be applied to both Gaussian and sparser data. Sparse signals derive from diverse models, and usually there is not enough information to establish a convincing model. In practical work, "non-**Gaussian**" generally means "sparser than Gaussian." Figure 7.4 illustrates random signals from a Gaussian probability function and a sparser signal made by cubing the random numbers that emerge from a Gaussian random-number generator.

7.2.2. Random numbers into a filter

Figure 7.5 shows **random** numbers fed through **leaky integration** and the resulting spectral amplitude. The output spectral amplitude of an integrator should be $|\omega|^{-1}$, but the decay constant in the leaky integrator gives instead the amplitude $(\omega^2 + \epsilon^2)^{-1/2}$. Since the random numbers are sparse, you can see damped exponents in the data itself. This enables us to confirm the direction of the time axis. If the random numbers had been Gaussian, the spectrum would be the same, but we would

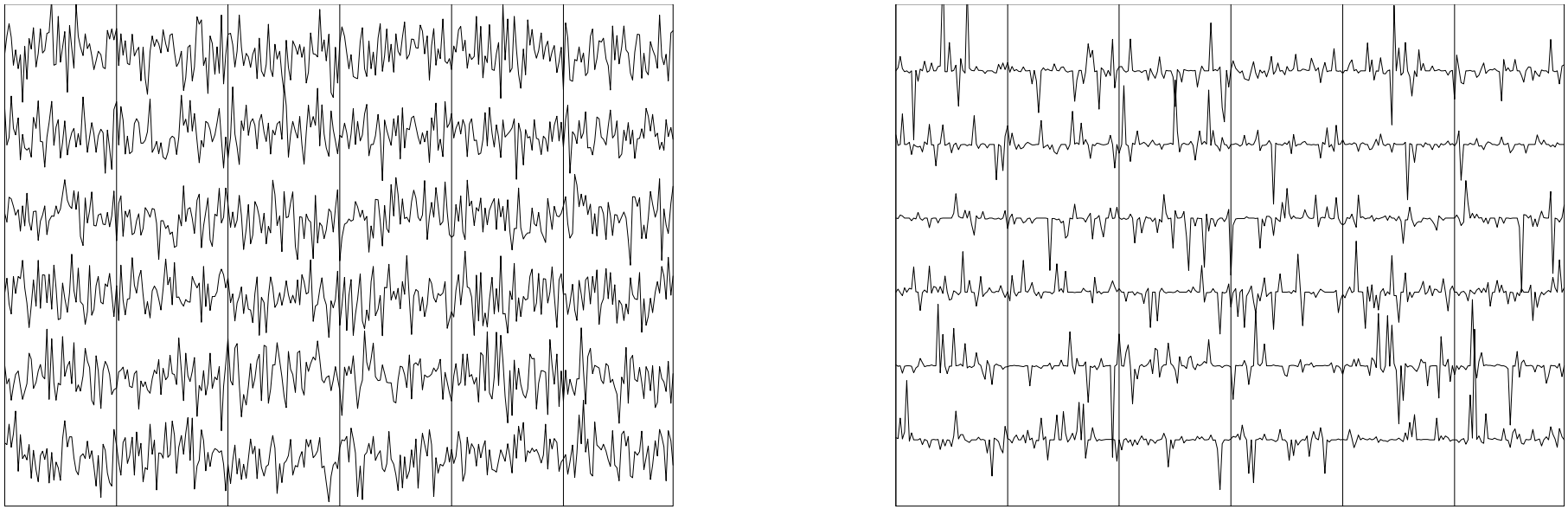


Figure 7.4: Left are random numbers from a Gaussian probability function. (The random numbers are connected by lines.) Right, the random numbers are cubed, making a signal in which large spikes are sparser. tsa-spikes [ER]

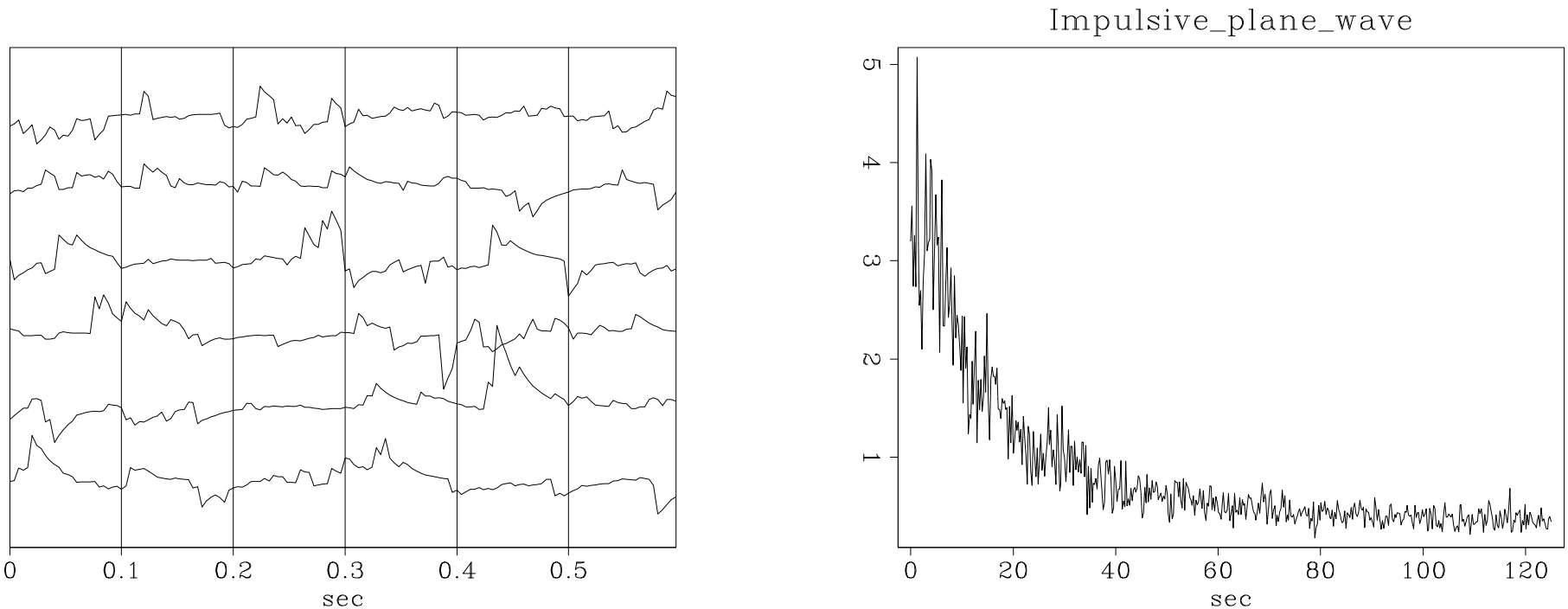


Figure 7.5: Left is sparse random noise passed through a leaky integrator. Right is the amplitude spectrum of the output. tsa-leaky [ER]

be able neither to see the damped exponents nor detect the direction of time.

7.2.3. Random numbers into the seismic spectral band

Figure 7.6 shows synthetic data designed to look like real seismic noise. Here some Gaussian random numbers were passed into a filter to simulate the seismic pass-band. Two five-term Butterworth filters (see chapter 10) were used, a highcut at .4 of the Nyquist and a lowcut at .1 of the Nyquist.

7.3. THE ERROR FILTER FAMILY

A simple regression for a **prediction filter** (f_1, f_2) is

$$\begin{bmatrix} x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \approx \begin{bmatrix} x_1 & x_0 \\ x_2 & x_1 \\ x_3 & x_2 \\ x_4 & x_3 \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} \tag{7.9}$$

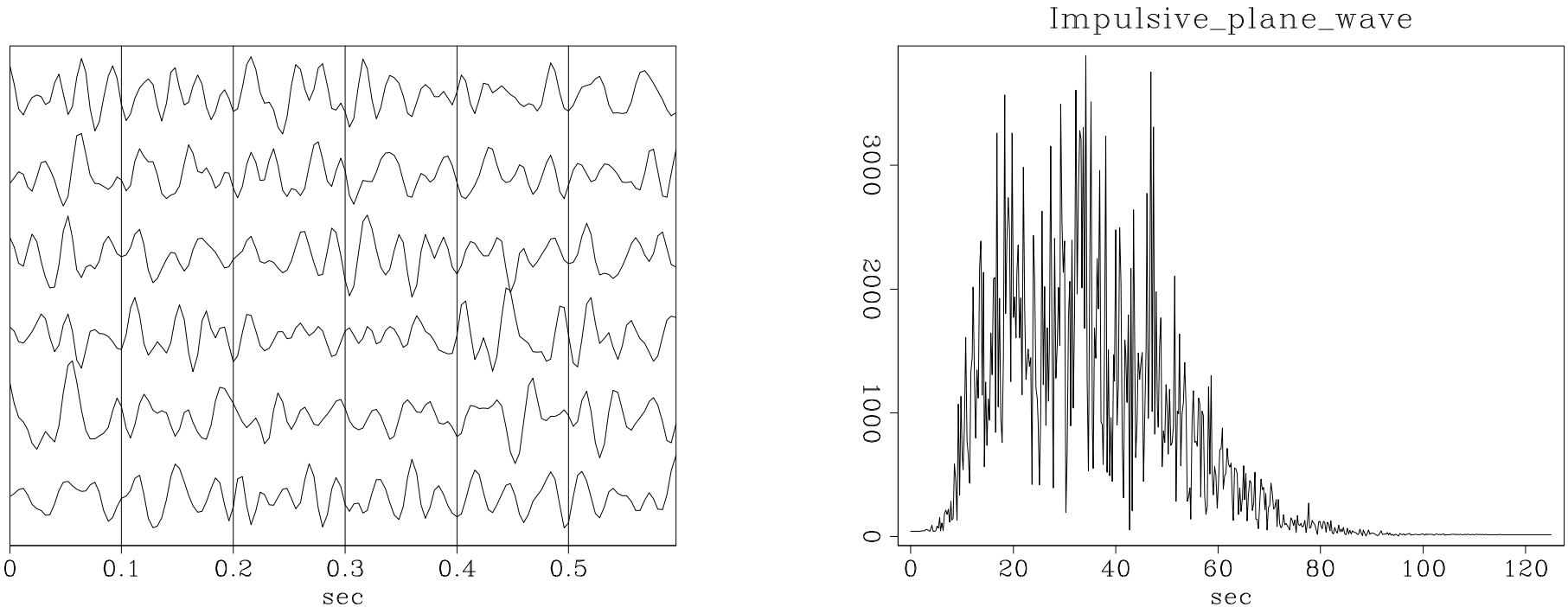


Figure 7.6: Left is Gaussian random noise passed through Butterworth filters to simulate the seismic passband. Right is the amplitude spectrum of the output. tsa-band [ER]

Notice that each row in this equation says that x_t fits a linear combination of x at earlier times; hence the description of f as a "prediction" filter. The error in the prediction is simply the left side minus the right side. Rearranging the terms, we get

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \approx \begin{bmatrix} x_2 & x_1 & x_0 \\ x_3 & x_2 & x_1 \\ x_4 & x_3 & x_2 \\ x_5 & x_4 & x_3 \end{bmatrix} \begin{bmatrix} 1 \\ -f_1 \\ -f_2 \end{bmatrix} \tag{7.10}$$

We have already written programs for regressions like (7.9). Regressions like (7.10), however, often arise directly in practice. They are easier to solve directly than by transforming them to resemble (7.9).

Multiple reflections are predictable. It is the unpredictable part of a signal, the prediction residual, that contains the primary information. The output of the filter $(1, -f_1, -f_2)$ is the unpredictable part of the input. This filter is a simple example of a "prediction-error" (PE) filter. It is one member of a family of filters called "error filters."

The error-filter family are filters with one coefficient constrained to be unity and various other coefficients constrained to be zero. Otherwise, the filter coefficients are chosen to have minimum power output. Names for various error filters follow:

$(1, a_1, a_2, a_3, \cdots, a_n)$	**prediction-error (PE) filter**
$(1, 0, 0, a_3, a_4, \cdots, a_n)$	gapped PE filter with a gap of 2
$(a_{-m}, \cdots, a_{-2}, a_{-1}, 1, a_1, a_2, a_3, \cdots, a_n)$	**interpolation-error (IE) filter**
$(a_{-m}, \cdots, a_{-4}, a_{-3}, 0, 0, 1, 0, 0, a_3, a_4, \cdots, a_n)$	a gapped IE filter

A program for computing all the error filters will be presented after we examine a collection of examples.

7.3.1. Prediction-error filters on synthetic data

The idea of using a **gap** in a prediction filter is to relax the goal of converting realistic signals into perfect impulses. Figure 7.7 shows synthetic data, sparse noise into a leaky integrator, and deconvolutions with prediction-error filters. Theoretically, the filters should turn out to be $1 - (.9Z)^{\text{gap}}$. Varying degrees of success are achieved by the filters obtained on the different traces, but overall, the results are good.

To see what happens when an unrealistic **deconvolution** goal is set for prediction error, we can try to compress a wavelet that is resistant to compression—for example, the impulse response of a Butterworth bandpass filter. The perfect filter to compress any wavelet is its inverse. But a wide region of the spectrum of a **Butter-**

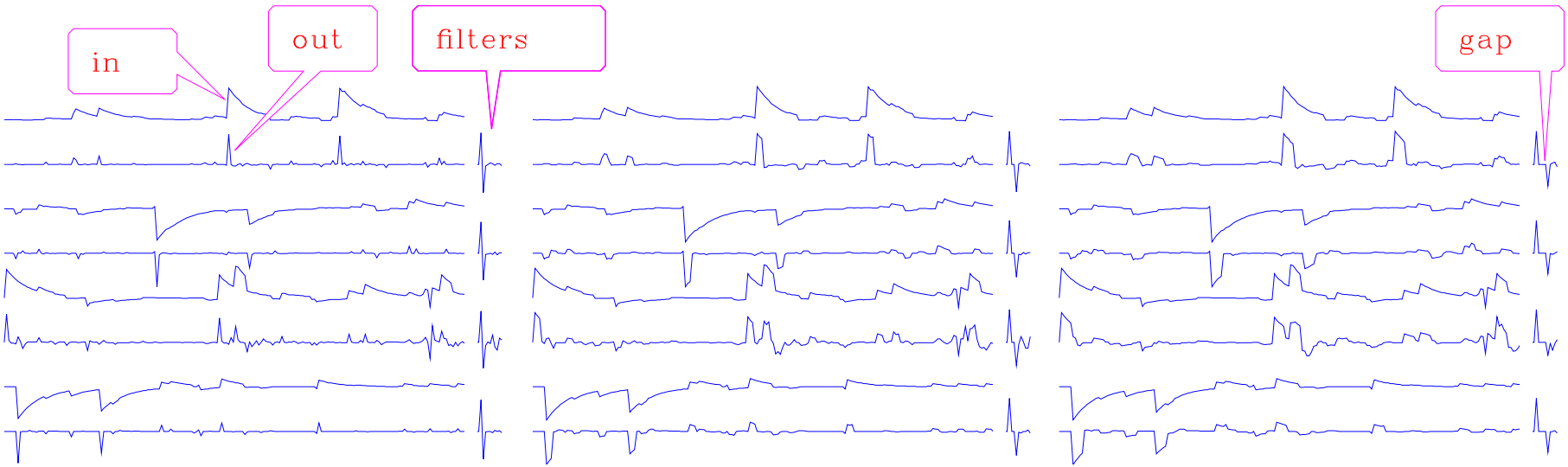


Figure 7.7: Deconvolution of leaky integrator signals with PE filters of various prediction-gap sizes. Inputs and outputs on alternate traces. Gap size increases from left to right. tsa-dleak [NR]

worth filter is nearly zero, so any presumed inverse must require nearly dividing by that range of zeros. Compressing a Butterworth filter is so difficult that I omitted the random numbers used in Figure 7.7 and applied prediction error to the Butterworth response itself, in Figure 7.8. Thus, we have seen that gapped PE filters sometimes

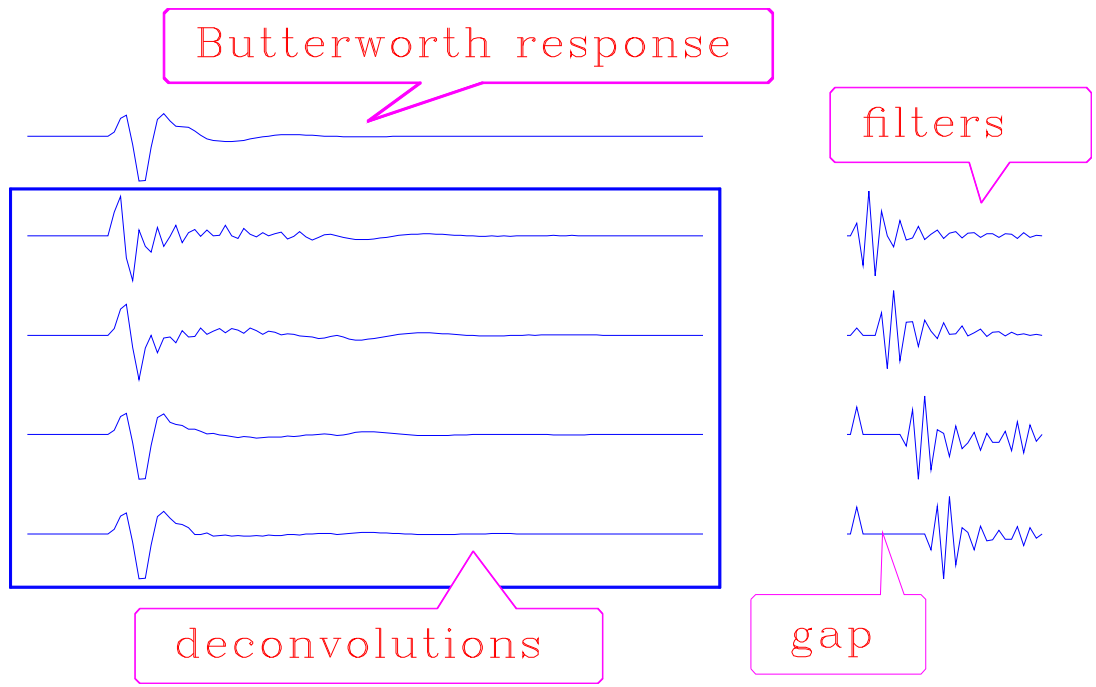


Figure 7.8: Butterworth deconvolution by prediction error. tsa-dbutter [NR]

are able to compress a wavelet, and sometimes are not. In real life, resonances arise

in the earth's shallow layers; and as we will see, the resonant filters can be shortened by PE filters.

7.3.2. PE filters on field data

Figure 7.9 is a nice illustration of the utility of **prediction-error filter**s. The input is quasi-sinusoidal, which indicates that **PE filter**ing should be successful. Indeed, some events are uncovered that probably would not have been identified on the input. In this figure, a separate problem is solved for each trace, and the resulting filter is shown on the right.

7.3.3. Prediction-error filter output is white.

The most important property of a **prediction-error filter** is that its output tends to a white spectrum. No matter what the input to this filter, its output tends to whiteness as the number of the coefficients $n \to \infty$ tends to infinity. Thus, the **PE filter** adapts itself to the input by absorbing all its **color**. If the input is already white, the a_j coefficients vanish. The PE filter is frustrated because with a white input it can

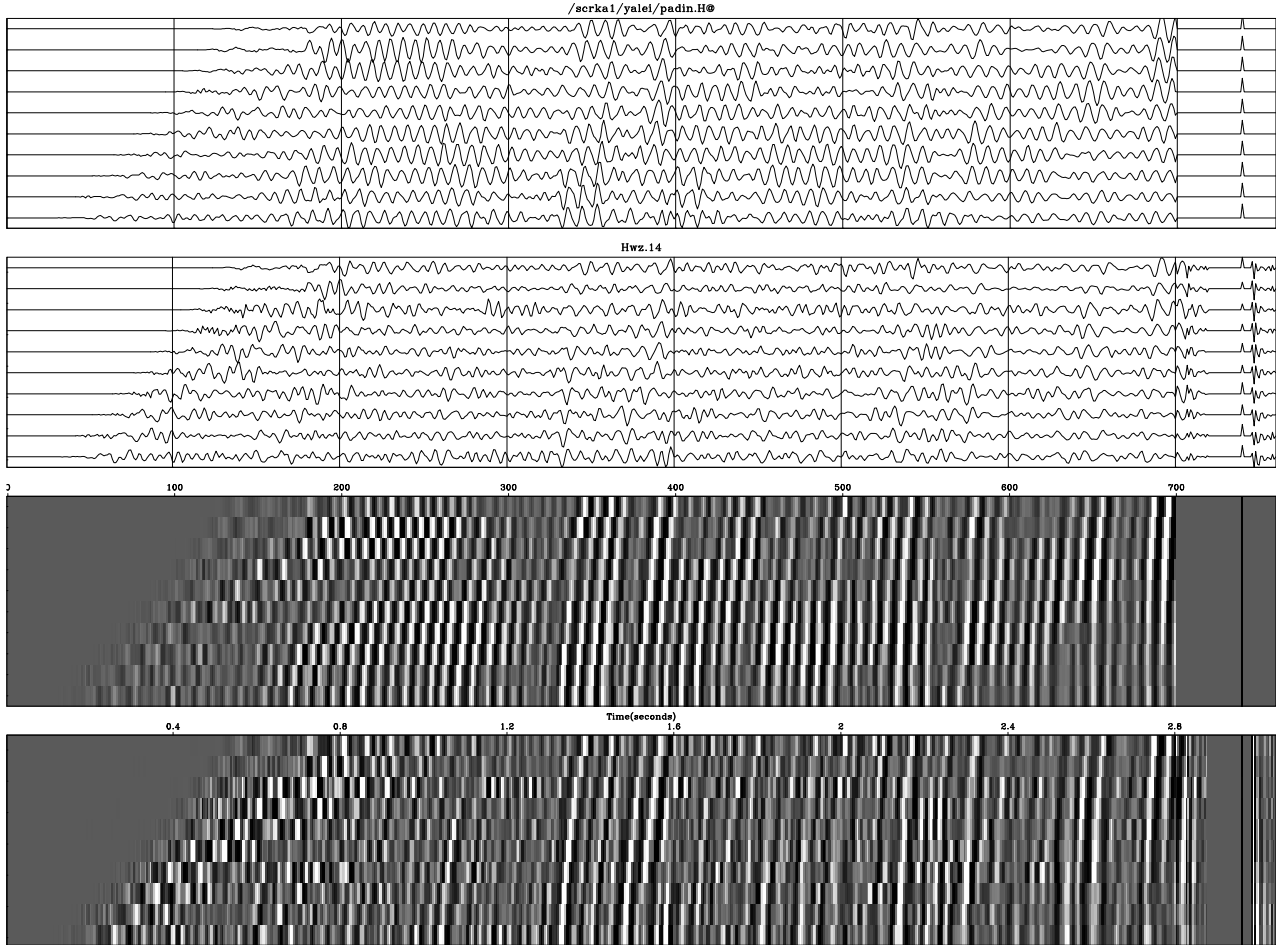


Figure 7.9: Data from offshore Texas (extracted from Yilmaz and Cumro dataset 14). Wiggle display above and raster below. Inputs above outputs. Filters displayed on the right. tsa-wz14 [NR]

predict nothing, so the output is the same as the input. Thus, if we were to cascade one PE filter after another, we would find that only the first filter does anything. If the input is a sinusoid, it is exactly predictable by a three-term recurrence relation, and all the color is absorbed by a three-term PE filter (see exercises). The power of a PE filter is that a short filter can often extinguish, and thereby represent the information in, a long filter.

That the output spectrum of a PE filter is **white** is very useful. Imagine the reverberation of the **soil** layer, highly variable from place to place, as the resonance between the surface and deeper consolidated rocks varies rapidly with surface location as a result of geologically recent fluvial activity. The spectral **color** of this erratic variation on surface-recorded seismograms is compensated for by a PE filter. Of course, we do not want PE-filtered seismograms to be white, but once they all have the same spectrum, it is easy to postfilter them to any desired spectrum.

Because the PE filter has an output spectrum that is white, the filter itself has a spectrum that is inverse to the input. Indeed, an effective mechanism of spectral estimation, developed by John P. **Burg** and described in **FGDP**, is to compute a PE filter and look at the inverse of its spectrum.

Another interesting property of the PE filter is that it is **minimum phase**.

The best proofs of this property are found in FGDP. These proofs assume uniform weighting functions.

7.3.4. Proof that PE filter output is white

[1] The basic idea of least-squares fitting is that the residual is orthogonal to the fitting functions. Applied to the PE filter, this idea means that the output of a PE filter is orthogonal to lagged inputs. The orthogonality applies only for lags in the past because prediction knows only the past while it aims to the future. What we want to show is different, namely, that the output is uncorrelated with *itself* (as opposed to the input) for lags in *both* directions; hence the output spectrum is **white**.

We are given a signal y_t and filter it by

$$x_t \quad = \quad y_t - \sum_{\tau>0} a_\tau y_{t-\tau} \tag{7.11}$$

[1] I would like to thank John P. **Burg** for this proof.

We found a_τ by setting to zero $d(\sum x_t^2)/da_\tau$:

$$\sum_t x_t y_{t-\tau} \quad = \quad 0 \qquad \text{for } \tau > 0 \tag{7.12}$$

We interpret this to mean that the residual is orthogonal to the fitting function, or the present PE filter output is orthogonal to its past inputs, or one side of the crosscorrelation vanishes. Taking an unlimited number of time lags and filter coefficients, the crosscorrelation vanishes not only for $\tau > 0$ but for larger values, say $\tau + s$ where $\tau \geq 0$ and $s > 0$. In other words, the future PE filter outputs are orthogonal to present and past inputs:

$$\sum_t x_{t+s} y_{t-\tau} \quad = \quad 0 \qquad \text{for } \tau \geq 0 \text{ and } s > 0 \tag{7.13}$$

Recall that if $\mathbf{r} \cdot \mathbf{u} = 0$ and $\mathbf{r} \cdot \mathbf{v} = 0$, then $\mathbf{r} \cdot (a_1 \mathbf{u} \pm a_2 \mathbf{v}) = 0$ for any a_1 and a_2. So for any a_τ we have

$$\sum_t x_{t+s} (y_t \pm a_\tau y_{t-\tau}) \quad = \quad 0 \qquad \text{for } \tau \geq 0 \text{ and } s > 0 \tag{7.14}$$

and for any linear combination we have

$$\sum_t x_{t+s}(y_t - \sum_{\tau>0} a_\tau y_{t-\tau}) \quad = \quad 0 \qquad \text{for } \tau \geq 0 \text{ and } s > 0 \tag{7.15}$$

Therefore, substituting from (7.11), we get

$$\sum_t x_{t+s} x_t \quad = \quad 0 \qquad \text{for } s > 0 \tag{7.16}$$

which is an **autocorrelation** function and must be symmetric. Thus,

$$\sum_t x_{t+s} x_t \quad = \quad 0 \qquad \text{for } s \neq 0 \tag{7.17}$$

Since the autocorrelation of the prediction-error output is an impulse, its spectrum is **white**. This has many interesting philosophical implications, as we will see next.

7.3.5. Nonwhiteness of gapped PE-filter output

When a PE filter is constrained so that a few near-zero-lag coefficients are zero, the output no longer tends to be **white** as the number of coefficients in the filter tends to

infinity. If f_1, the filter coefficient of $Z = e^{i\omega\Delta t}$, vanishes, then $F(\omega)$ lacks the slow variation in ω that this term provides. It lacks just the kind of spectral variation that could boost weak near-Nyquist noises up to the strength of the main passband. With such variation made absent by the constraint, the growth of Nyquist-region energy is no longer a necessary byproduct of PE filtering.

Figure 7.10 illustrates a **PE filter** with a long **gap**. (The gap was chosen to be a little less than the water depth.) This example nicely shows the suppression of some multiple reflections, but unfortunately I do not see that any primary reflections have been uncovered. Because the prediction gap is so long, the filter causes no visible change to the overall spectrum. Notice how much more the spectrum was broadened by the filter with a shorter gap in Figure 7.9. The theoretical association of prediction gap width with spectral broadening is examined next. Another interesting feature of Figure 7.10, which we will investigate later, is a geometrical effect. This shows up as poor multiple removal on and above the diagonal lines and happens because of the nonzero separation of the sound source and receiver.

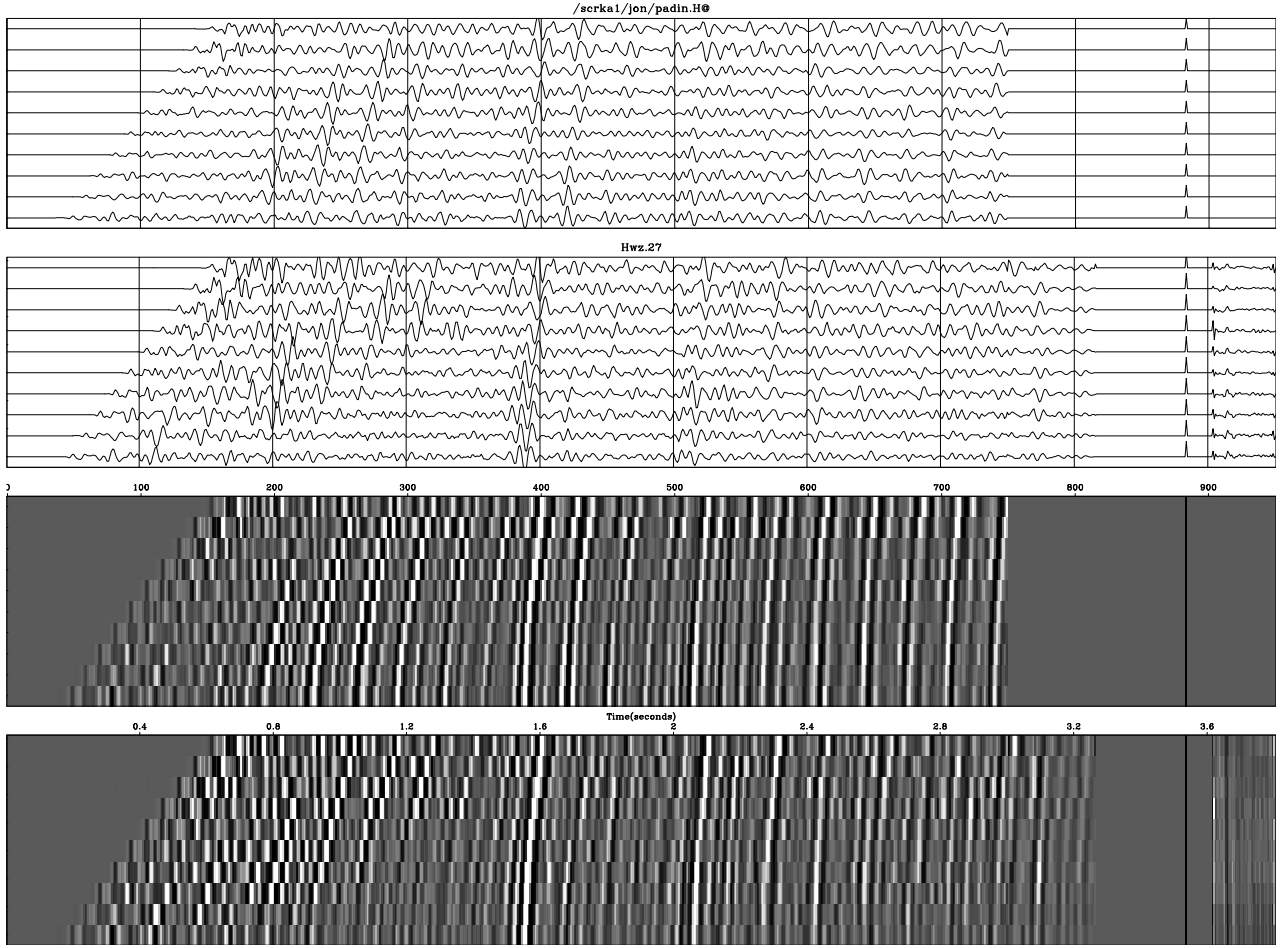


Figure 7.10: Data from offshore Canada (extracted from Yilmaz and Cumro dataset 27) processed by gapped prediction error. Inputs above outputs; filters displayed on the right. Nicely suppressed multiples appear in boxes. Badly suppressed multiples are shown above diagonal lines. tsa-wz27 [NR]

7.3.6. Postcoloring versus prewhitening

The output of a PE filter, as we know, is **white** (unless it is gapped), but people do not like to look at white signals. Signals are normally sampled at adequate density, which means that they are small anywhere near the Nyquist frequency. There is rarely energy above the half-Nyquist and generally little but marine noises above the quarter-Nyquist. To avoid boosting these noises, the ungapped PE filter is generally altered or accompanied by other filters. Three common approaches follow:

- Use a gapped filter.
- Deconvolve, then apply a filter with the desired spectrum $S(\omega)$.
- Prefilter the input with $S(\omega)^{-1}$, then deconvolve with an ungapped PE filter, and finally postfilter with $S(\omega)$.

The last process is called "**prewhitening**" for some complicated reasons: the idea seems to be that the prefilter removes known color so that the least-squares coefficients are not "wasted" on predicting what is already known. Thus the prefilter spectrum $S(\omega)^{-1}$ is theoretically the inverse of the prior estimate of the input spectrum. In real life, that is merely an average of estimates from other data. If the desired output spectrum does not happen to be $S(\omega)$, it does not matter, since any

final display filter can be used. Although this is a nice idea, I have no example to illustrate it.

There is also the question of what phase the postfilter should have. Here are some cautions against the obvious two choices:

- **Zero phase**: a symmetrical filter has a noncausal response.
- **Causal**: if a later step of processing is to make a coherency analysis for velocity versus time, then the effective time will be more like the signal maximum than the first break.

Since the postfilter is broadband, its phase is not so important as that of the deconvolution operator, which tries to undo the phase of a causal and resonant earth.

7.4. BLIND DECONVOLUTION

The **prediction-error filter** solves the "**blind-deconvolution**" problem. So far little has been said about the input data to the PE filter. A basic underlying model is that the input data results from white noise into a filter, where the filter is some process in nature. Since the output of the PE filter is white, it has the same spectrum as the

original white noise. The natural hypothesis is that the filter in nature is the inverse of our PE filter. Both filters are causal, and their amplitude spectra are mutually inverse. Theoretically, if the model filter were minimum phase, then its inverse would be causal, and it would be our PE filter. But if the model filter were an all-pass filter, or had an **all-pass filter** as a factor, then its inverse would not be causal, so it could not be our PE filter.

The blind-deconvolution problem can be attacked without PE filters by going to the frequency domain. Figure 7.11 shows sample spectra for the basic model. We see that the spectra of the random noise are random-looking. In chapter 11 we will study random noise more thoroughly; the basic fact important here is that the longer the random time signal is, the rougher is its spectrum. This applies to both the input and the output of the filter. Smoothing the very rough spectrum of the input makes it tend to a constant; hence the common oversimplification that the **spectrum** of random noise is a constant. Since for $Y(Z) = F(Z)X(Z)$ we have $|Y(\omega)| = |F(\omega)||X(\omega)|$, the spectrum of the output of random noise into a filter is like the spectrum of the filter, but the output spectrum is jagged because of the noise. To estimate the spectrum of the filter in nature, we begin with data (like the output in Figure 7.11) and smooth its spectrum, getting an approximation to that of the filter.

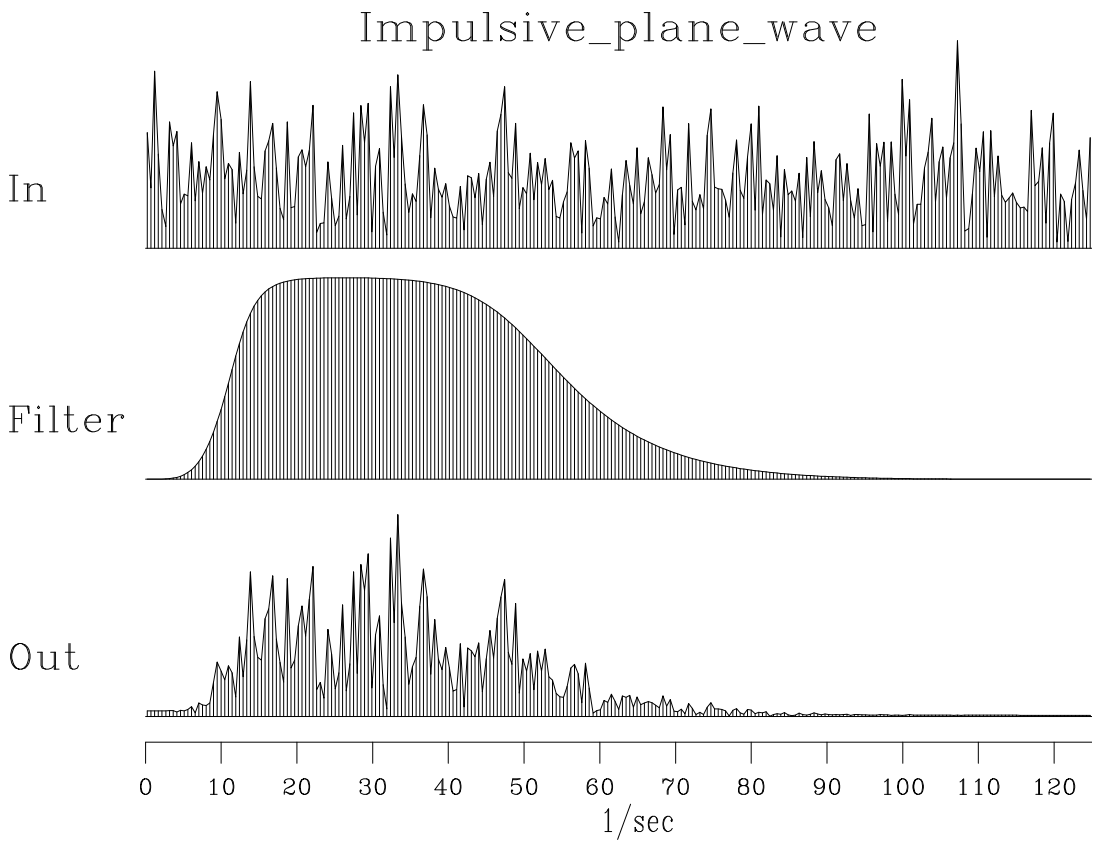


Figure 7.11: Spectra of random numbers, a filter, and the output of the filter. tsa-model [ER]

For **blind deconvolution** we simply apply the inverse filter. The simplest way to get such a filter is to inverse transform the smoothed amplitude spectrum of the data to a time function. This time-domain wavelet will be a symmetrical signal, but in real life the wavelet should be causal. Chapter 10 shows a Fourier method, called the "Kolmogoroff method," for finding a causal wavelet of a given spectrum. Chapter 11 shows that the length of the Kolmogoroff wavelet depends on the amount of spectral smoothing, which in this chapter is like the ratio of the data length to the filter length.

In **blind deconvolution**, Fourier methods determine the spectrum of the unknown wavelet. They seem unable to determine the wavelet's phase by measurements, however—only to assert it by theory. We will see that this is a limitation of the "**stationarity**" assumption, that signal strengths are uniform in time. Where signal strengths are nonuniform, better results can be found with weighting functions and time-domain methods. In Figure 7.14 we will see that the **all-pass filter** again becomes visible when we take the trouble to apply appropriate weights.

7.5. WEIGHTED ERROR FILTERS

What I have described above is "industrial standard" material. A great many companies devote much human and computer energy to it. Now we will see what new opportunities are promised by a formulation that includes weighting functions.

7.5.1. Automatic gain control

Echos get weaker with time, though the information content is unrelated to the signal strength. Echos also vary in strength as different materials are encountered by the outgoing wave. Programs for echo analysis typically divide the data by a scaling factor that is a smoothed average of the signal strength. This practice is nearly universal, although it is fraught with hazards. An example of **automatic gain control** (**AGC**) is to compute the divisor by forming the absolute value of the signal strength and then smoothing with the program `triangle()` /prog:triangle or the program `leaky()` /prog:leaky. Pitfalls are the strange amplitude behavior surrounding the water bottom, and the overall loss of information contained in amplitudes. Personally, I have found that the gain function t^2 nearly always eliminates the need for AGC on raw field data, but I have no doubt that AGC is occasionally needed. (A

theoretical explanation for t^2 is given in IEI.)

7.5.2. Gain before or after convolution

It is a common but questionable practice to apply **AGC** to echo soundings before filter analysis. A better practice is first to analyze according to the laws of physics and only at the last stage to apply gain functions for purposes of statistical estimation and final display. Here we will examine correct and approximate ways of setting up deconvolution problems with gain functions. Then we will use CG to solve the proper formulation.

Solving problems in the time domain offers an advantage over the frequency domain because in the time domain it is easy to control the interval where the solution should exist. Another advantage of the time domain arises when weighting functions are appropriate. I have noticed that people sometimes use Fourier solutions inappropriately, forcing themselves to use uniform weighting when another weighting would work better. Since we look at echos, it is unavoidable that we apply gain functions. Weighting is always justified on the process *outputs*, but it is an approximation of unknown validity on the data that is *input* to those processes. I

will clarify this approximation by an equation with two filter points and an output of four time points. In real-life applications, the output is typically 1000-2000 points and the filter 5-50 points. The valid formulation of a filtering problem is

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \approx \begin{bmatrix} w_1 & 0 & 0 & 0 \\ 0 & w_2 & 0 & 0 \\ 0 & 0 & w_3 & 0 \\ 0 & 0 & 0 & w_4 \end{bmatrix} \left(\begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{bmatrix} - \begin{bmatrix} x_1 & 0 \\ x_2 & x_1 \\ x_3 & x_2 \\ 0 & x_3 \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} \right) \tag{7.18}$$

The weights w_t are any positive numbers we choose. Typically the w_t are chosen so that the residual components are about equal in magnitude.

If, instead, the weighting function is applied to the *inputs*, we have an approximation that is somewhat different:

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \approx \begin{bmatrix} w_1 d_1 \\ w_2 d_2 \\ w_3 d_3 \\ w_4 d_4 \end{bmatrix} - \begin{bmatrix} w_1 x_1 & 0 \\ w_2 x_2 & w_1 x_1 \\ w_3 x_3 & w_2 x_2 \\ 0 & w_3 x_3 \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} \tag{7.19}$$

Comparing the weighted output-residual equation (7.18) to the weighted input-data

equation (7.19), we note that their right-hand columns do not match. The right-hand column in (7.18) is $(0, w_2x_1, w_3x_2, w_4x_3)'$ but in (7.19) is $(0, w_1x_1, w_2x_2, w_3x_3)'$. The matrix in (7.19) is a simple convolution, so some fast solution methods are applicable.

7.5.3. Meet the Toeplitz matrix

The solution to any least-squares problem proceeds explicitly or implicitly by finding the inverse to a **covariance matrix**. Recall the basic filtering equation (**??**),

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} x_1 & 0 & 0 \\ x_2 & x_1 & 0 \\ x_3 & x_2 & x_1 \\ x_4 & x_3 & x_2 \\ x_5 & x_4 & x_3 \\ 0 & x_5 & x_4 \\ 0 & 0 & x_5 \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix} \tag{7.20}$$

which we can abbreviate by $\mathbf{y} = \mathbf{Xf}$. To gain some understanding of your cultural heritage in time-series analysis, form the **covariance matrix** $\mathbf{X}'\mathbf{X}$,

$$\mathbf{X}'\mathbf{X} = \begin{bmatrix} s_0 & s_1 & s_2 \\ s_1 & s_0 & s_1 \\ s_2 & s_1 & s_0 \end{bmatrix} \tag{7.21}$$

where the elements s_t are lags of the **autocorrelation** of x_t. This **covariance matrix** is an example of a **Toeplitz** matrix. When an application is formulated in the frequency domain, you may encounter a spectrum as a divisor. When the same application is formulated in the time domain, you will see an autocorrelation matrix that needs inversion.

The Toeplitz matrix is highly structured. Whereas an $n \times n$ matrix could contain n^2 different elements, the Toeplitz matrix contains only n elements that are different from each other. When computers had limited memory, this memory savings was important. Also, there are techniques for solving least-squares problems with Toeplitz covariance matrices that are much faster than those for solving problems with arbitrary matrices. The effort for arbitrary matrices is proportional to n^3, whereas for Toeplitz matrices it is n^2. These advantages of Toeplitz

matrices were once overwhelming, although now they are rarely significant. But because old methods linger on, we need to decide if they are warranted. Recall that we wrote three convolution programs, `contran()` /prog:contran, `contrunc()` /prog:contrunc, and `convin()` /prog:convin. You can verify that a Toeplitz matrix arises only in the first of these. The other two represent different ways of handling boundaries. Let **W** be a diagonal matrix of weighting functions. You can also verify that the covariance matrix $\mathbf{B}'\mathbf{W}\mathbf{B}$ is not Toeplitz. Thus, Toeplitz matrices only arise with uniform weighting and transient boundary conditions. If the only tool you have is a hammer, then everything you see starts to look like a nail. In earlier days, and by inertia even today, convolution applications tend to be formulated as uniformly weighted with transient boundaries. This is a **pitfall**.

> Toeplitz matrices are associated with elegant mathematics and rapid numerical solutions. Applications that are solvable by standard methods have historically been cast in Toeplitz form by imposing simplifying assumptions. This is risky.

The approximation (7.19) becomes reasonable when the weights are slowly variable, i.e., when w_t is a slowly variable function of t. In practice, I think the

approximation is often justified for slow t^2 gain but questionable for automatic gains that are faster. Compared to Toeplitz methods of solving equation (7.19), the CG method of solving (7.18) is slower. Here we are going to see how to solve the problem correctly. If you want to solve the correct problem rapidly, perhaps you can do so by solving the approximate problem first by a quasi-analytic method and then doing a few steps of CG.

7.5.4. Setting up any weighted CG program

Equation (7.18) is of the form $\mathbf{0} \approx \mathbf{W}(\mathbf{d} - \mathbf{B}\mathbf{f})$. This can be converted to a new problem without weights by defining a new data vector $\mathbf{W}\mathbf{d}$ and a new operator $\mathbf{W}\mathbf{B}$ simply by carrying $\mathbf{W}$ through the parentheses to $\mathbf{0} \approx \mathbf{W}\mathbf{d} - (\mathbf{W}\mathbf{B})\mathbf{f}$. Convolution followed by weighting is implemented in subroutine `wcontrunc()` /prog:wcontrunc.

wcontrunc

```
# filter and weight.
#
subroutine wcontrunc( adj, add, ww,    lag, nx, xx, nf,ff,  nn,yy )
integer i,            adj, add,        lag, nx,     nf,     nn
real                           ww(nn),     xx(nx), ff(nf), yy(nn)
temporary real                 ss(nn)
call adjnull(         adj, add,                         ff,nf,  yy,nn)
if( adj == 0) {  call contrunc( 0,0, lag,  nx,xx,   nf,ff,  nn,ss)
                 do i= 1, nn
                        yy(i) =  yy(i) + ss(i) * ww(i)
                 }
else          {  do i= 1, nn
                        ss(i) =          yy(i) * ww(i)
                 call contrunc( 1,1, lag,  nx,xx,   nf,ff,  nn,ss)
                 }
return;  end
```

Back

7.6. CALCULATING ERROR FILTERS

The *error* in prediction (or interpolation) is often more interesting than the prediction itself. When the predicted component is removed, leaving the unpredictable, the residual is the prediction error. Let us see how the program `shaper()` can be used to find an interpolation-error filter like $(f_{-2}, f_{-1}, 1, f_1, f_2)$. The statement of wishes is

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \approx \begin{bmatrix} . \\ . \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ . \\ . \end{bmatrix} + \begin{bmatrix} x_1 & . & . & . \\ x_2 & x_1 & . & . \\ x_3 & x_2 & . & . \\ x_4 & x_3 & x_1 & . \\ x_5 & x_4 & x_2 & x_1 \\ x_6 & x_5 & x_3 & x_2 \\ . & x_6 & x_4 & x_3 \\ . & . & x_5 & x_4 \\ . & . & x_6 & x_5 \\ . & . & . & x_6 \end{bmatrix} \begin{bmatrix} f_{-2} \\ f_{-1} \\ f_1 \\ f_2 \end{bmatrix} \tag{7.22}$$

Taking the column vector of x_t to the other side of the equation gives the form required by previous CG programs. After solving this system for (f_{-2}, f_{-1}, f_1, f_2), we insert the "1" to make the **IE filter** (f_{-2}, f_{-1}, 1, f_1, f_2), which, applied to the data x_t, gives the desired IE output.

Notice that the matrix in (7.22) is *almost* **convolution**. It would be convolution if the central column were not absent. I propose that you not actually solve the system (7.22). Instead I will show you a more general solution that uses the convolution operator itself. That way you will not need to write programs for the many "almost" convolution operators arising from the many PE and IE filters with their various **gap**s and **lag**s.

The conjugate-gradient program here is a combination of earlier CG programs and the weighting methods we must introduce now:

- We need to constrain a filter coefficient to be unity, which we can do by initializing it to unity and then allowing no changes to it.
- We may wish to constrain some other filter coefficients to be zero (gapping) by initializing them to zero and allowing no changes to them.
- We may want the output to occur someplace other than off-end prediction.

Thus we will specify a time lag that denotes the predicted or interpolated time point. The program `contrunc()` /prog:contrunc is designed for this.

Incorporating all these features into `shaper()`, we get `iner()`. iner For a filter of the form $(1, f_1, f_2, \cdots, f_{n-1})$, we would specify `lag=1, gap1=1, gapn=1`. For a filter of the form $(1, 0, f_2, \cdots, f_{n-1})$, we would specify `lag=1, gap1=1, gapn=2`. For a filter of the form $(f_{-2}, f_{-1}, 1, f_1, f_2)$, we would specify `nf=5, lag=3, gap1=3, gapn=3`.

This program uses the convolution program `contrunc()`, which is handy in practice because its output has the same length as its input. This convenience is partly offset by the small danger that significant output energy in the "start up" and "off end" zones could be truncated. Specifically, that energy would be in the top two and bottom two rows of equation (7.22).

7.6.1. Stabilizing technique

Theory for **stabilizing least squares**, using equations (**??**) and (**??**), was described earlier in this book. I installed this stabilization, along with the filter determinations discussed in this chapter, but as I expected, stabilization in this highly overde-

```
# weighted interpolation-error filter
#
subroutine iner( nf,f, nr,yy,rr, ww, niter, lag, gap1, gapn)
integer i, iter, nf,   nr,           niter, lag, gap1, gapn
real     f(nf), yy(nr), rr(nr), ww(nr)
temporary real  df(nf), sf(nf), dr(nr), wr(nr), sr(nr)
if( lag < gap1 || lag > gapn )  call erexit('input fails gap1<=lag<=gapn')
do i= 1, nf
        f(i) = 0.
f(lag) = 1.                                                  # set output lag
call         wcontrunc( 0,0, ww, lag, nr,yy, nf, f, nr,wr)
call scaleit( -1.,                                     nr,wr)   # negative
do iter= 0, niter {
        call wcontrunc( 1,0, ww, lag, nr,yy, nf,df, nr,wr)   # df=yy*wr
        do i= gap1, gapn
                df(i) = 0.                                   # constrained lags
        call wcontrunc( 0,0, ww, lag, nr,yy, nf,df, nr,dr)   # dr=yy*df
        call cgstep( iter,  nf, f,df,sf, _
                            nr,wr,dr,sr )                    # f=f+df
        }
call          contrunc( 0,0,     lag, nr,yy, nf,f,  nr,rr)   # unweighted res
return; end
```

Back

termined application showed no advantages. Nevertheless, it is worth seeing how stabilization is implemented, particularly since the changes to the program calling `iner()` make for more informative plots.

The input data is modified by appending a zero-padded impulse at the data's end. The output will contain the filter impulse response in that region. The spike size is chosen to be compatible with the data size, for the convenience of the plotting programs. The **weighting function** in the appended region is scaled according to how much stabilization is desired. Figure 7.12 shows the complete input and residual. It also illustrates the problem that output data flows beyond the length of the input data because of the nonzero length of the filter. This extra output is undoubtedly affected by the truncation of the data, and its energy should not be part of the energy minimization. Therefore it is weighted by zero.

EXERCISES:

1 Given a sinusoidal function $x_t = \cos(\omega t + \phi)$, a three-term recurrence relationship predicts x_t from the previous two points, namely, $x_t = a_1 x_{t-1} + a_2 x_{t-2}$. Find a_1 and a_2 in terms of $\omega \Delta t$. HINT: See chapter 3. (Notice that the coefficients depend on ω but not ϕ.)

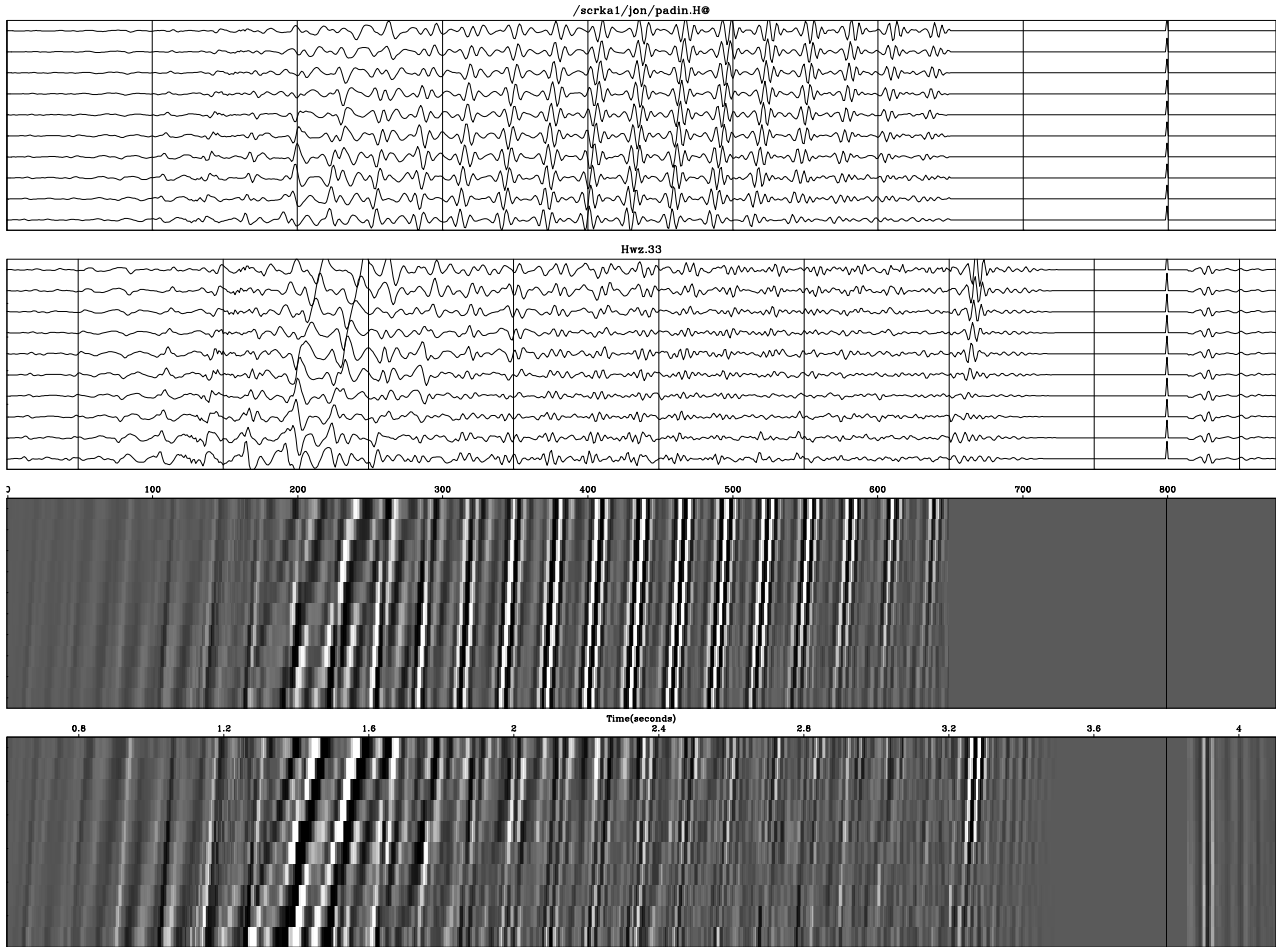


Figure 7.12: Data from the North Sea (extracted from Yilmaz and Cumro dataset 33) processed by prediction error. Rightmost box is weighted according to the desired stabilization. The **truncation** event is weighted by zero. tsa-wz33 [NR]

2 Figure 7.9 has a separate filter for each trace. Consider the problem of finding a single filter for all the traces. What is the basic operator and its adjoint? Assemble these operators using subroutine `contrunc()` /prog:contrunc.

3 Examine the filters on Figure 7.12. Notice that, besides the pulse at the water depth, another weak pulse occurs at double that depth. Suggest a physical mechanism. Suggest a mechanism relating to computational approximations.

7.7. INTERPOLATION ERROR

Interpolation-error filters have the form $(a_{-m}, \cdots, a_{-2}, a_{-1}, 1, a_1, a_2, a_3, \cdots, a_n)$, where the a_t coefficients are adjusted to minimize the power in the filter output. IE filters have the strange characteristic that if the input spectrum is $S(\omega)$, then the output spectrum will tend to $S(\omega)^{-1}$. Thus these filters tend to turn poles into zeros and vice versa. To see why IE filters invert the spectrum of the input, we only need recall the basic premise of least-squares methods, that the residual (the output) is orthogonal to the fitting function (the input at all lags except the zero lag). Thus, the crosscorrelation of the input and the output is an impulse. This

can only happen if their spectra are inverses, which is a disaster for the overall appearance of a seismogram. Such drastic spectral change can be controlled in a variety of ways, as is true with PE filters, but with IE filters there seems to be little experience to build on besides my own. Figure 7.13 illustrates an interpolation-error result where gapping has been used to limit the color changes. I also chose the **gap** to condense the wavelet. You judge whether the result is successful. Notice also a high-frequency arrival after the diagonal lines: this shows that the IE filters are boosting very high frequencies despite the gapping.

7.7.1. Blind all-pass deconvolution

A well-established theoretical concept that leads to unwarranted pessimism is the idea that **blind deconvolution** cannot find an **all-pass filter**. If we carefully examine the analysis leading to that conclusion, we will find lurking the assumption that the weighting function used in the least-squares estimation is uniform. And when this assumption is wrong, so is our conclusion, as Figure 7.14 shows. Recall that the inverse to an all-pass filter is its time reverse. The reversed shape of the filter is seen on the inputs where there happen to be isolated spikes.

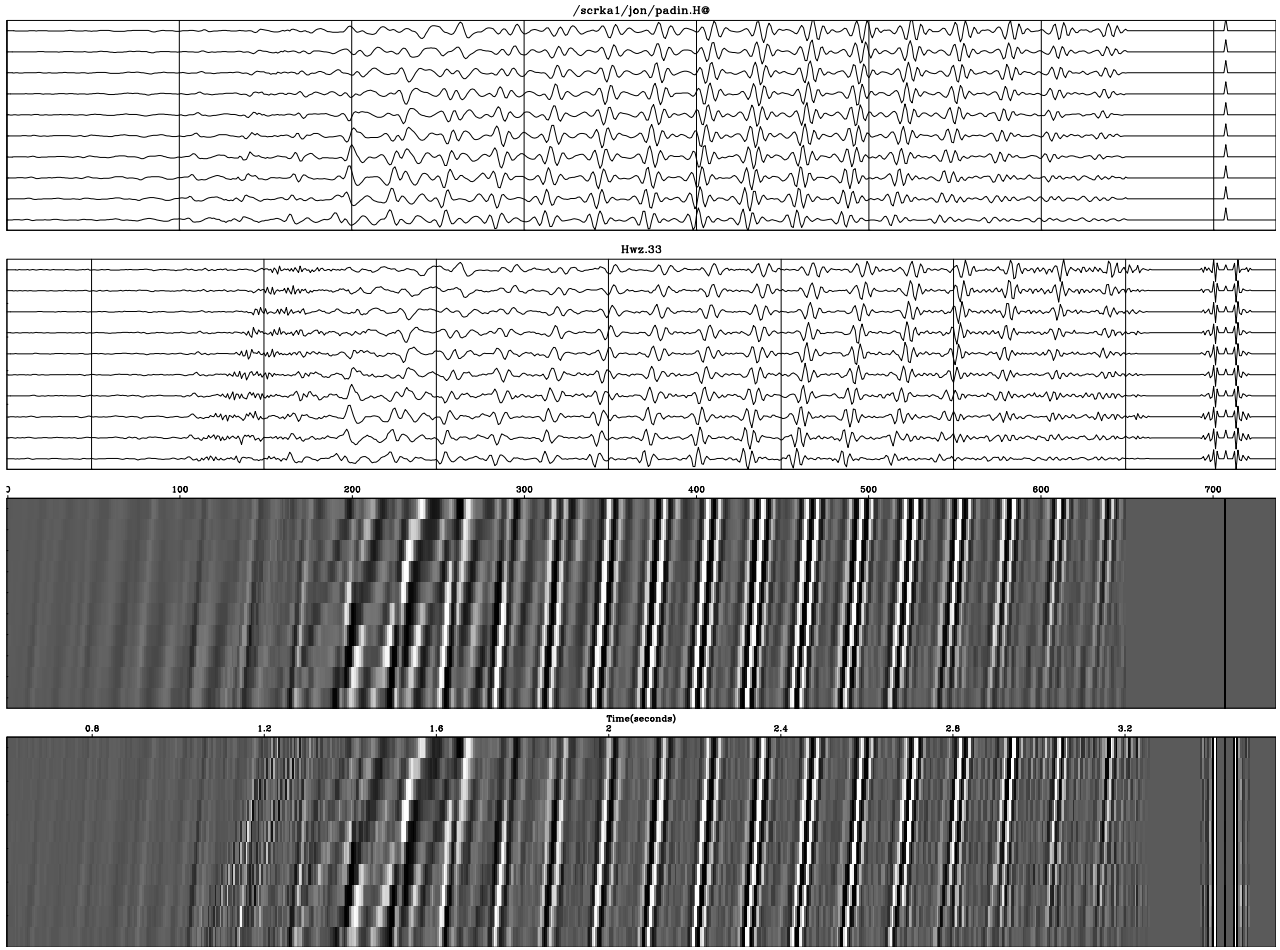


Figure 7.13: Data from the North Sea (extracted from Yilmaz and Cumro dataset 33) processed by interpolation error. Inputs above outputs. Filters displayed on the right. tsa-wz33ie [NR]

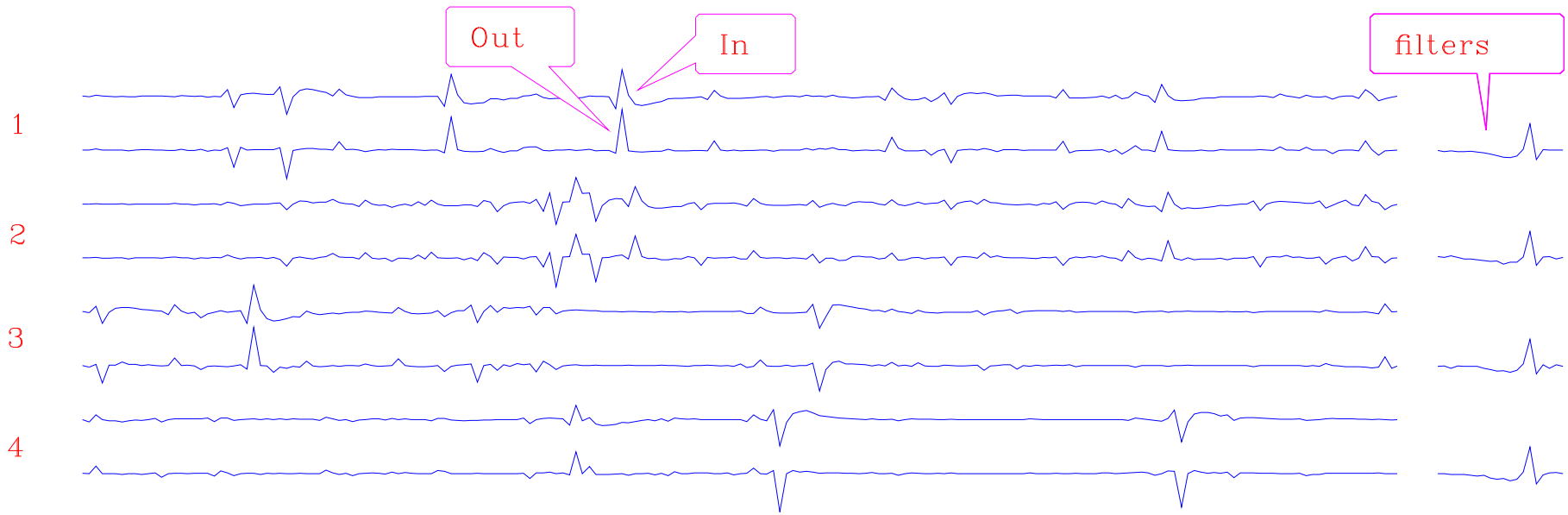


Figure 7.14: Four independent trials of deconvolution of sparse noise into an all-pass filter. Alternate lines are input and output. tsa-dallpass [NR]

Let us see what theory predicts cannot be done, and then I will tell you how I did it. If you examine the unweighted least-squares error-filter programs, you will notice that the first calculation is the convolution operator and then its transpose. This takes the autocorrelation of the input and uses it as a gradient search direction. Take a white input and pass it through a phase-shift filter; the output autocorrelation is an impulse function. This function vanishes everywhere except for the impulse itself, which is constrained against rescaling. Thus the effective gradient is zero. The solution, an impulse filter, is already at hand, so a phase-shift filter seems unfindable.

On the other hand, if the signal strength of the input varies, we should be balancing its expectation by weighting functions. This is what I did in Figure 7.14. I chose a weighting function equal to the inverse of the absolute value of the output of the filter plus an ϵ. Since the weighting function depends on the output, the process is iterative. The value of ϵ chosen was 20% of the maximum signal value.

Since the iteration is a **nonlinear** procedure, it might not always work. A well-established body of theory says it will not work with **Gaussian** signals, and Figure 7.15 is consistent with that theory.

In Figure 7.13, I used weighting functions roughly inverse to the envelope of the

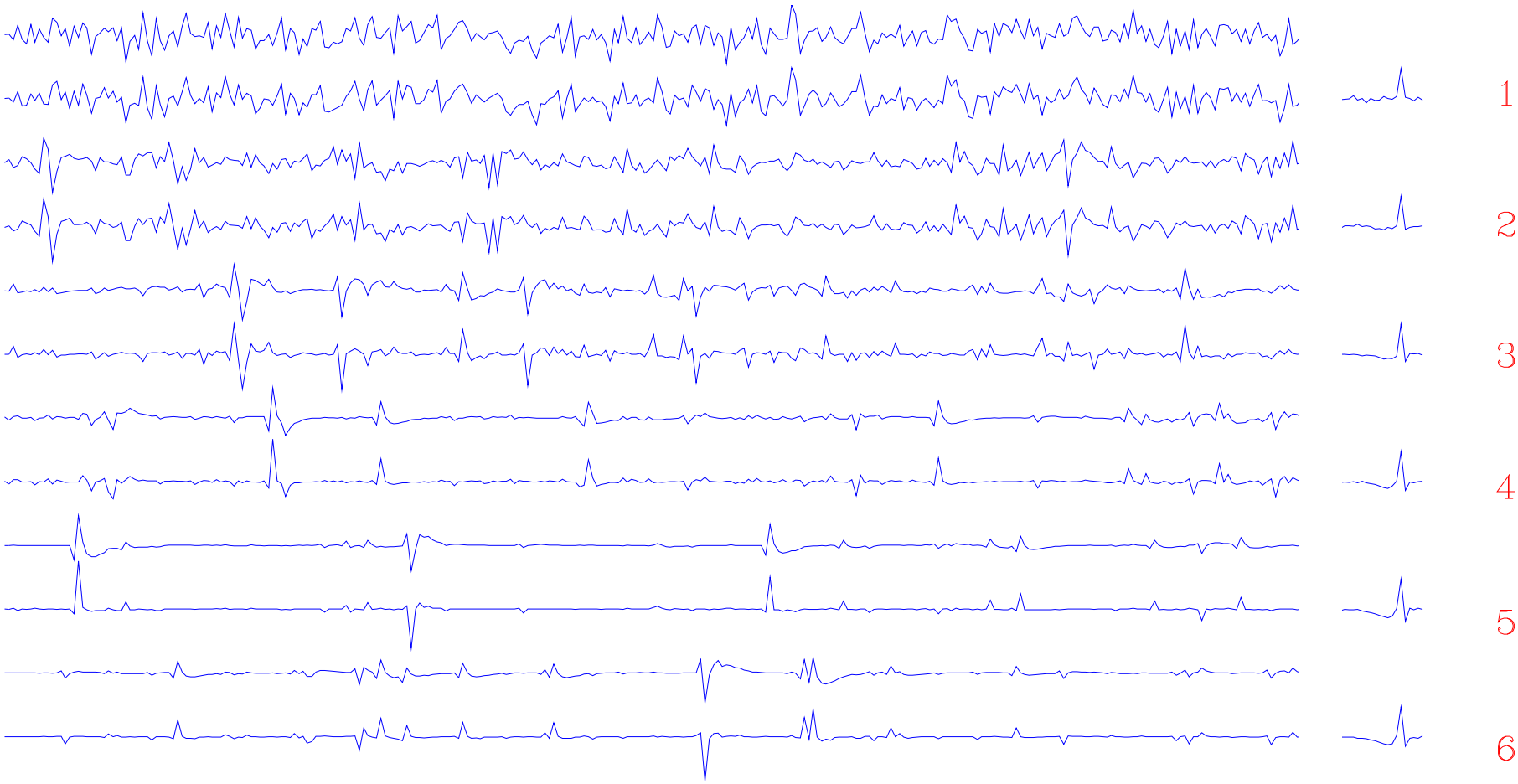


Figure 7.15: Failure of blind all-pass deconvolution for Gaussian signals. The top signal is based on Gaussian random numbers. Lower signals are based on successive integer powers of Gaussian signals. Filters (on the right) fail for the Gaussian case, but improve as signals become sparser. tsa-dgauss [NR]

signal, taking a floor for the envelope at 20% of the signal maximum. Since weighting functions were used, the filters need not have turned out to be symmetrical about their centers, but the resulting asymmetry seems to be small.

Chapter 8

Missing-data restoration

A brief summary of chapters 5 and 6 is that “the answer” is the solution to an inversion problem—a series of steps with many **pitfall**s. Practitioners often stop after the first step, while academics quibble about the convergence, i.e., the last

steps. Practitioners might stop after one step to save effort, to save risk, or because the next step is not obvious. Here we study a possible second step—replacing the zero-valued data presumed by any adjoint operator with more reasonable values.

A great many processes are limited by the requirement to avoid **spatial alias**ing—that no wavelength should be shorter than twice the sampling interval on the data wave field. This condition forces costly expenditures in 3-D reflection data acquisition and yields a mathematical dichotomy between data processing in exploration seismology and data processing in earthquake seismology.

The simple statement of the spatial Nyquist requirement oversimplifies real life. Recently, S. **Spitz** (1991) showed astonishing results that seem to violate the Nyquist requirement. In fact they force us to a deeper understanding of it. In this chapter we will discuss many new opportunities that promise much lower data-acquisition costs and should also reduce the conceptual gap between exploration and earthquake seismology.

8.1. INTRODUCTION TO ALIASING

In its simplest form, the Nyquist condition says that we can have no frequencies higher than two points per wavelength. In migration, this is a strong constraint on data collection. It seems there is no escape. Yet, in applications dealing with a CMP gather (such as in Figure 5.5 or 5.6), we see data with spatial frequencies that exceed Nyquist and we are not bothered, because after NMO, these frequencies are OK. Nevertheless, such data is troubling because it breaks many of our conventional programs, such as downward continuation with finite differences or with Fourier transforms. (No one uses focusing for **stacking**.) Since NMO defies the limitation imposed by the simple statement of the Nyquist condition, we revise the condition to say that the real limitation is on the spectral bandwidth, not on the maximum frequency. Mr. Nyquist does not tell us where that bandwidth must be located. Further, it seems that precious bandwidth *need not be contiguous.* The signal's spectral band can be split into pieces and those pieces positioned in different places. Fundamentally, the issue is whether the total bandwidth exceeds Nyquist. Noncontiguous Nyquist bands are depicted in Figure 8.1.

Noncontiguous bandwidth arises naturally with two-dimensional data where there are several plane waves present. There the familiar spatial Nyquist limitation

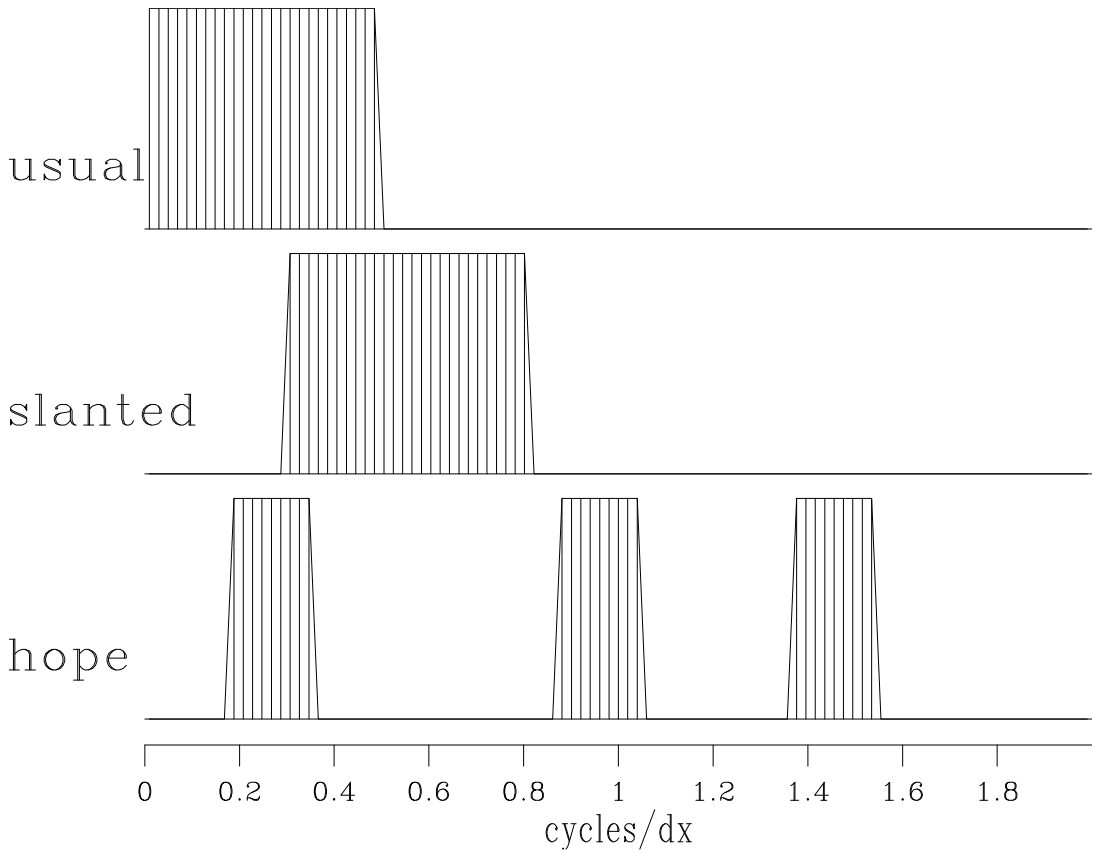


Figure 8.1: Hypothetical spatial frequency bands. Top is typical. Middle for data skewed with $\tau = t - px$. Bottom depicts data with wave arrivals from three directions. mis-nytutor [ER]

oversimplifies real life because the plane waves link time and space.

> The spatial Nyquist frequency need not limit the analysis of seismic data because the plane-wave model links space with time.

8.1.1. Relation of missing data to inversion

We take *data space* to be a uniform mesh on which some values are given and some are missing. We rarely have missing values on a time axis, but commonly have missing values on a space axis, i.e., missing signals. Missing signals (traces) happen occasionally for miscellaneous reasons, and they happen systematically because of **alias**ing and **truncation**. The aliasing arises for economic reasons—saving instrumentation by running receivers far apart. Truncation arises at the ends of any survey, which, like any human activity, must be finite. Beyond the survey lies more hypothetical data. The traces we will find for the **missing data** are not as good as real observations, but they are closer to reality than supposing unmeasured data is zero valued. Making an image with a single application of an adjoint modeling operator amounts to assuming that data vanishes beyond its given locations. **Migration**

is an example of an economically important process that makes this assumption. Dealing with missing data is a step beyond this. In **inversion**, restoring **missing data** reduces the need for arbitrary model filtering.

8.1.2. My model of the world

In your ears now are sounds from various directions. From moment to moment the directions change. Momentarily, a single direction (or two) dominates. Your ears sample only two points in x-space. Earthquake data is a little better. Exploration data is much better and sometimes seems to satisfy the Nyquist requirement, especially when we forget that the world is 3-D.

We often characterize data from any region of (t, x)-space as "good" or "noisy" when we really mean it contains "few" or "many" plane-wave events in that region. For noisy regions there is no escaping the simple form of the Nyquist limitation. For good regions we may escape it. Real data typically contains both kinds of regions. Undersampled data with a broad distribution of plane waves is nearly hopeless. Undersampled data with a sparse distribution of plane waves is prospective. Consider data containing a spherical wave. The angular bandwidth in a plane-wave

decomposition appears huge *until we restrict attention to a small region* of the data. (Actually a spherical wave contains very little information compared to an arbitrary wave field.) It can be very helpful in reducing the local angular bandwidth if we can deal effectively with tiny pieces of data as we did in chapter 4. If we can deal with tiny pieces of data, then we can adapt to rapid spatial and temporal variations. This chapter will show such tiny windows of data. We will begin with missing-data problems in one dimension. Because these are somewhat artificial, we will move on to two dimensions, where the problems are genuine.

8.2. MISSING DATA IN ONE DIMENSION

A method for restoring **missing data** is to ensure that the restored data, after specified filtering, has minimum energy. Specifying the filter chooses the interpolation philosophy. Generally the filter is a “roughening" filter. When a roughening filter goes off the end of smooth data, it typically produces a big end transient. Minimizing energy implies a choice for unknown data values at the end, to minimize the transient. We will examine five cases and then make some generalizations.

A method for restoring missing data is to ensure that the restored data, after specified filtering, has minimum energy.

Let m denote a missing value. The dataset on which the examples are based is $(\cdots, m, m, 1, m, 2, 1, 2, m, m, \cdots)$. Using subroutine `miss1()` /prog:miss1, values were found to replace the missing m values so that the power in the filtered data is minimized. Figure 8.2 shows interpolation of the dataset with $1 - Z$ as a roughening filter. The interpolated data matches the given data where they overlap.

Figures 8.2–8.6 illustrate that the rougher the filter, the smoother the interpolated data, and vice versa. Let us switch our attention from the residual spectrum to the residual itself. The residual for Figure 8.2 is the *slope* of the signal (because the filter $1 - Z$ is a *first derivative*), and the slope is constant (uniformly distributed) along the straight lines where the least-squares procedure is choosing signal values. So these examples confirm the idea that the **least-squares method** abhors large values (because they are squared). Thus, least squares tend to distribute uniformly residuals in both time and frequency to the extent the **constraint**s allow.

This idea helps us answer the question, what is the best filter to use? It suggests choosing the filter to have an amplitude spectrum that is inverse to the spectrum we

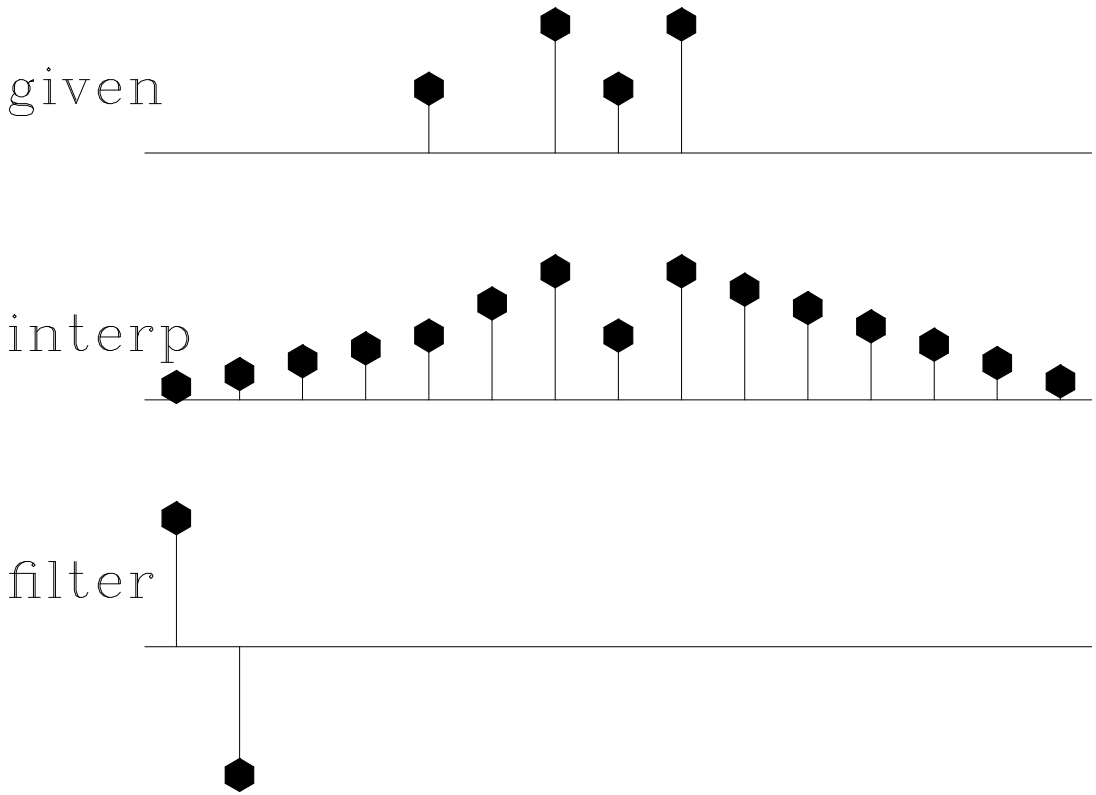


Figure 8.2: Top is given data. Middle is given data with interpolated values. Missing values seem to be interpolated by straight lines. Bottom shows the filter $(1, -1)$, whose output has minimum power. mis-mlines [ER]

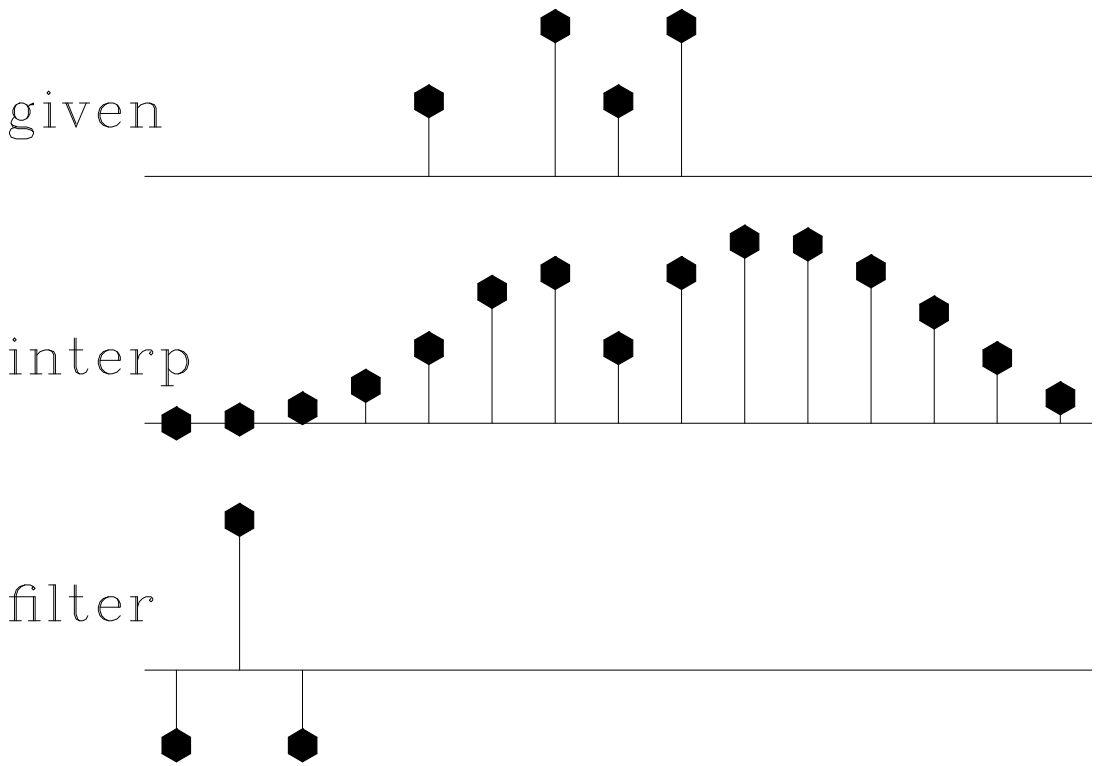


Figure 8.3: Top is the same input data as in Figure 8.2. Middle is interpolated. Bottom shows the filter $(-1, 2, -1)$. The missing data seems to be interpolated by parabolas. mis-mparab [ER]

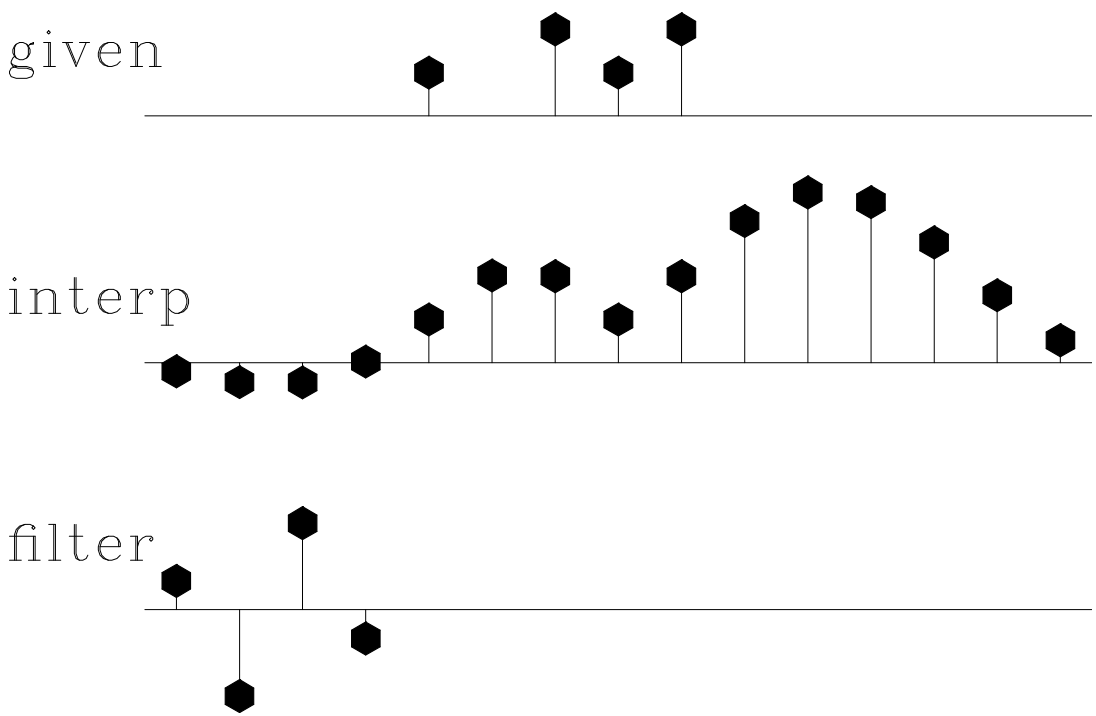


Figure 8.4: Top is the same input. Middle is interpolated. Bottom shows the filter $(1, -3, 3, -1)$. The missing data is very smooth. It shoots upward high off the right end of the observations, apparently to match the data slope there. mis-mseis [ER]

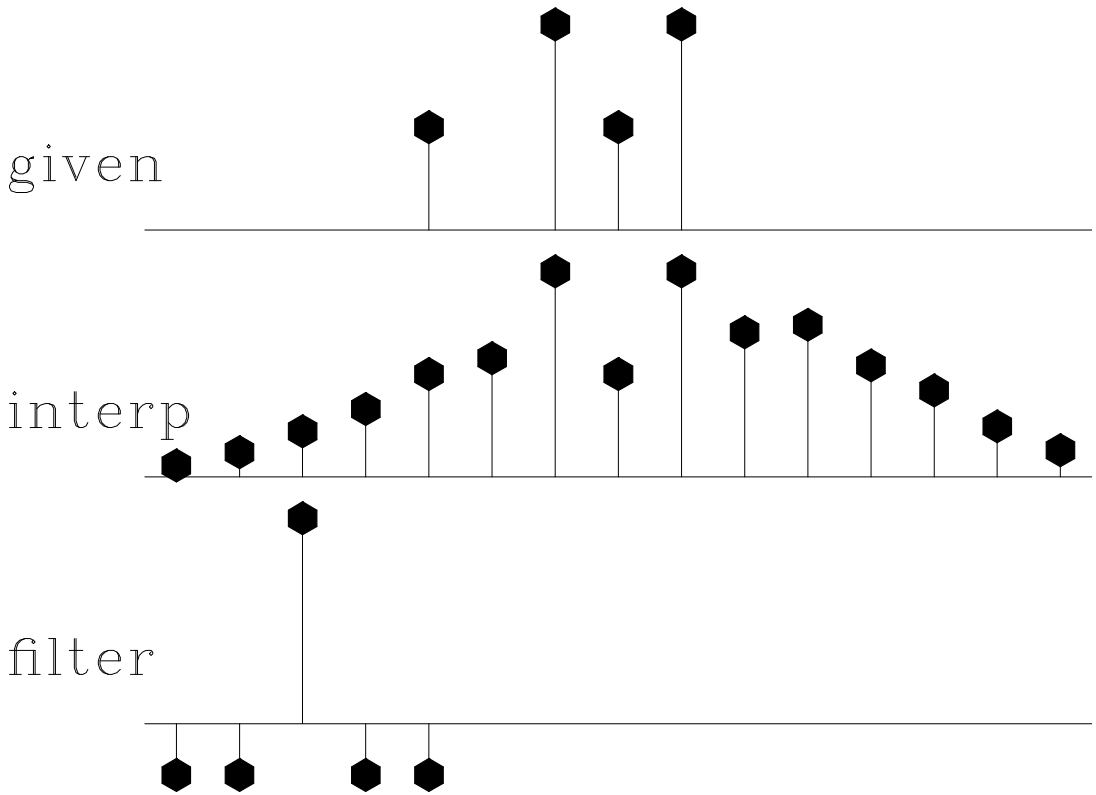


Figure 8.5: The filter $(-1,-1,4,-1,-1)$ gives interpolations with stiff lines. They resemble the straight lines of Figure 8.2, but they project through a cluster of given values instead of projecting to the nearest given value. Thus, this interpolation tolerates noise in the given data better than the interpolation shown in Figure 8.4. mis-msmo [ER]

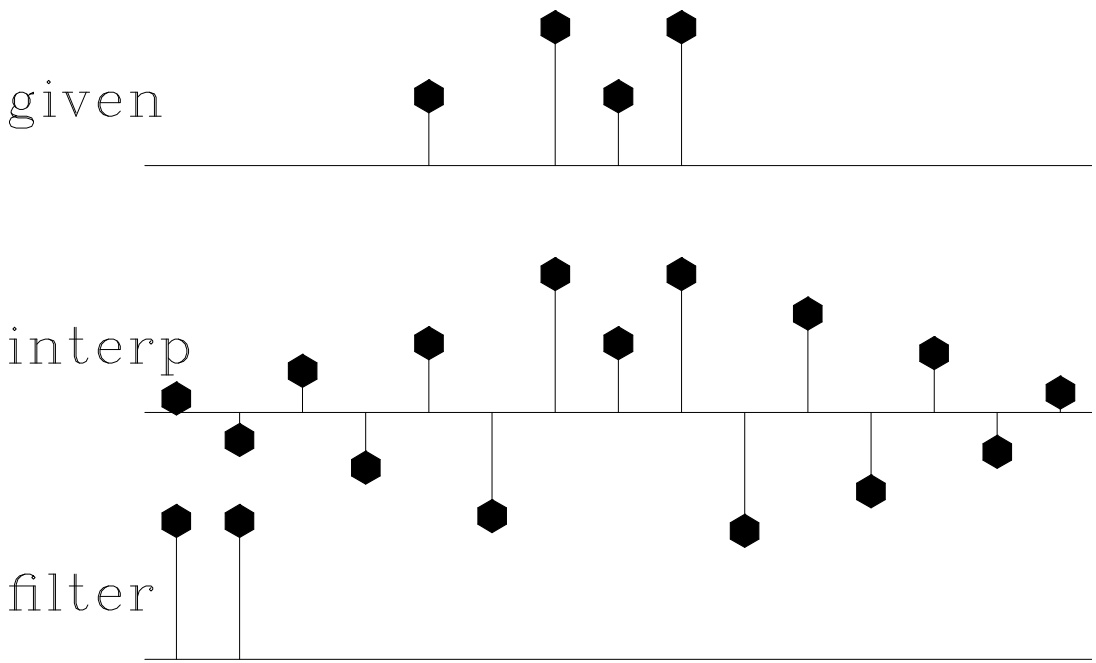


Figure 8.6: Bottom shows the filter $(1, 1)$. The interpolation is rough. Like the given data itself, the interpolation has much energy at the Nyquist frequency. But unlike the given data, it has little zero-frequency energy. mis-moscil [ER]

want for the interpolated data. A systematic approach is given in the next section, but I will offer a simple subjective analysis here. Looking at the data, I see that all points are positive. It seems, therefore, that the data is rich in low frequencies; thus the filter should contain something like $(1-Z)$, which vanishes at zero frequency. Likewise, the data seems to contain Nyquist frequency, so the filter should contain $(1+Z)$. The result of using the filter $(1-Z)(1+Z) = 1-Z^2$ is shown in Figure 8.7. This is my best subjective interpolation based on the idea that the missing data should look like the given data. The **interpolation** and **extrapolation**s are so good that you can hardly guess which data values are given and which are interpolated.

8.2.1. Missing-data program

There are two ways to code the missing-data estimation, one conceptually simple and the other leading to a concise program. Begin with a given filter **f** and create a shifted-column matrix **F**, as in equation **??**. The problem is that $0 \approx \mathbf{Fd}$ where **d** is the data. The columns of **F** are of two types, those that multiply *missing* data values and those that multiply *known* data values. Suppose we reorganize **F** into two collections of columns: $\mathbf{F}_m$ for the missing data values, and $\mathbf{F}_k$ for the known

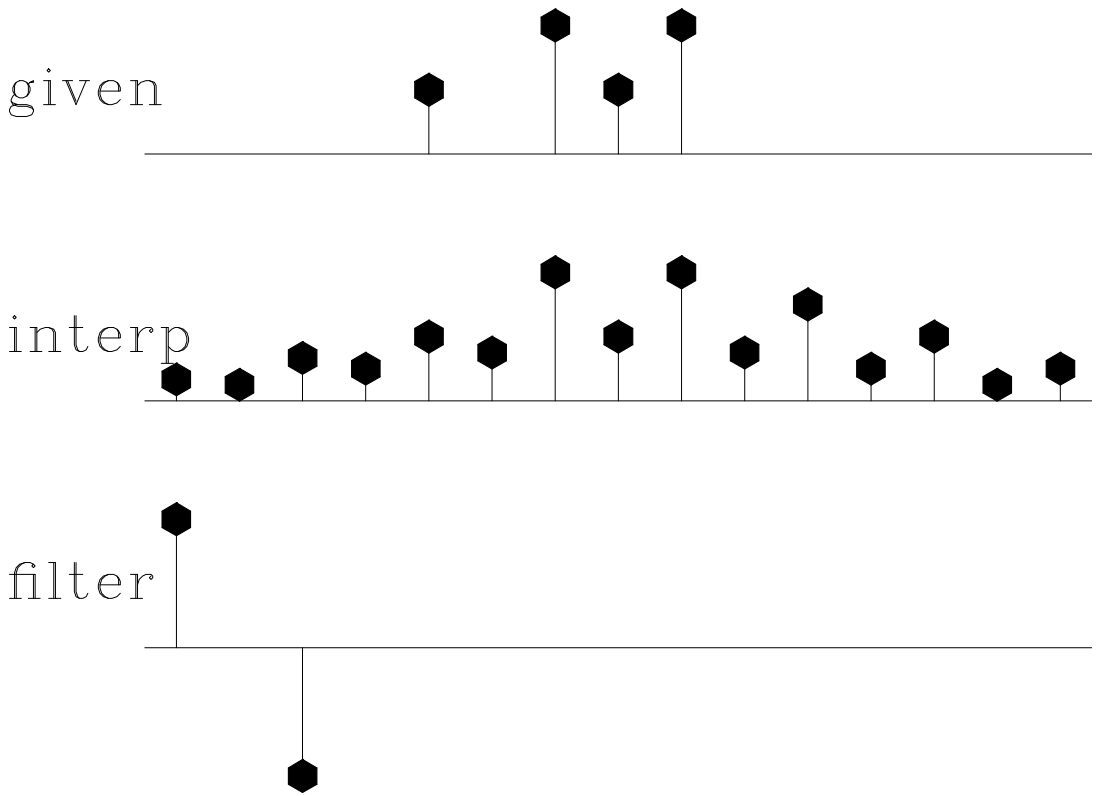


Figure 8.7: Top is the same as in Figures 8.2 to 8.6. Middle is interpolated. Bottom shows the filter $(1, 0, -1)$, which comes from the coefficients of $(1 - Z)(1 + Z)$. Both the given data and the interpolated data have significant energy at both zero and Nyquist frequencies. mis-mbest [ER]

data values. Now, instead of $0 \approx \mathbf{F}\mathbf{d}$, we have $0 \approx \mathbf{F}_m\mathbf{d}_m + \mathbf{F}_k\mathbf{d}_k$ or $-\mathbf{F}_k\mathbf{d}_k \approx \mathbf{F}_m\mathbf{d}_m$. Taking $-\mathbf{F}_k\mathbf{d}_k = \mathbf{y}$, we have simply an overdetermined set of simultaneous equations like $\mathbf{y} \approx \mathbf{A}\mathbf{x}$, which we solved with `cgmeth()` /prog:cgmeth.

The trouble with this approach is that it is awkward to program the partitioning of the operator into the known and missing parts, particularly if the application of the operator uses arcane techniques, such as those used by the fast Fourier transform operator or various numerical approximations to differential or partial differential operators that depend on regular data sampling. Even for the modest convolution operator, we already have a library of convolution programs that handle a variety of end effects, and it would be much nicer to use the library as it is rather than recode it for all possible geometrical arrangements of missing data values. Here I take the main goal to be the clarity of the code, not the efficiency or accuracy of the solution. (So, if your problem consumes too many resources, and if you have many more known points than missing ones, maybe you should solve $\mathbf{y} \approx \mathbf{F}_m\mathbf{x}$ and ignore what I suggest below.)

How then can we mimic the erratically structured $\mathbf{F}_m$ operator using the $\mathbf{F}$ operator? When we multiply any vector into $\mathbf{F}$, we must be sure that the vector has zero-valued components to hit the columns of $\mathbf{F}$ that correspond to missing data.

When we look at the result of multiplying the adjoint $\mathbf{F}'$ into any vector, we must be sure to ignore the output at the rows corresponding to the missing data. As we will see, both of these criteria can be met using a single loop.

The missing-data program begins by loading the negative-filtered known data into a **residual**. Missing data should try to reduce this residual. The iterations proceed as in `cgmeth()` /prog:cgmeth, `invstack()` /prog:invstack, `deghost()` /prog:deghost, `shaper()` /prog:shaper, and `iner()` /prog:iner. The new ingredient in the missing-data subroutine `miss1()` /prog:miss1 is the simple **constraint** that the known data cannot be changed. Thus, after the gradient is computed, the components that correspond to known data values are set to zero. miss1 That prevents changes to the known data by motion any distance along the **gradient**. Likewise, motion along previous steps cannot perturb the known data values. Hence, the CG method (finding the minimum power in the plane spanned by the gradient and the previous step) leads to minimum power while respecting the constraints.

EXERCISES:

1 Figure 8.2–8.6 seem to extrapolate to vanishing signals at the side boundaries. Why is that so, and what could be done to leave the sides unconstrained in that

```
# fill in missing data on 1-axis by minimizing power out of a given filter.
#
subroutine miss1(     na, a, np, p, copy, niter)
integer iter, ip, nr, na,    np,          niter
real    p(np)           # in: known data with zeros for missing values.
                        # out: known plus missing data.
real    copy(np)        # in: copy(ip) vanishes where p(ip) is a missing value.
real    a(na)           # in: roughening filter
temporary real  dp(np),sp(np),  r(np+na-1),dr(np+na-1),sr(np+na-1)
nr = np+na-1
        call contran( 0, 0, na,a, np, p, r)     # r = a*p   convolution
        call scaleit  (        -1., nr,    r)     # r = -r
do iter= 0, niter {                             # niter= number missing or less
        call contran( 1, 0, na,a, np,dp, r)     # dp(a,r)   correlation
        do ip= 1, np
                if( copy(ip) != 0.)             # missing data where copy(ip)==0
                        dp(ip) = 0.             # can't change known data
        call contran( 0, 0, na,a, np,dp, dr)    # dr=a*dp   convolution
        call cgstep( iter, np,p,dp,sp, nr,r,dr,sr)      # p=p+dp; r=r-dr
        }
return; end
```

Back

way?

2 Compare Figure 8.7 to the interpolation values you expect for the filter $(1, 0, -.5)$.

3 Indicate changes to `miss1()` /prog:miss1 for missing data in two dimensions.

4 Suppose the call in `miss1()` /prog:miss1 was changed from `contran()` /prog:contr to `convin()` /prog:convin. Predict the changed appearance of Figure 8.2.

5 Suppose the call in `miss1()` was changed from `contran()` /prog:contran to `convin()` /prog:convin. What other changes need to be made?

6 Show that the interpolation curve in Figure 8.3 is not parabolic as it appears, but cubic. (HINT: Show that $(\nabla^2)'\nabla^2 u = 0$.)

7 Verify by a program example that the number of iterations required with simple constraints is the number of free parameters.

8.3. MISSING DATA AND UNKNOWN FILTER

Recall the missing-data figures beginning with Figure 8.2. There the filters were taken as known, and the only unknowns were the missing data. Now, instead of having a predetermined filter, we will solve for the filter along with the missing data. The principle we will use is that the output power is minimized while the filter is constrained to have one nonzero coefficient (else all the coefficients would go to zero). We will look first at some results and then see how they were found.

In Figure 8.8 the filter is constrained to be of the form $(1, a_1, a_2)$. The result is pleasing in that the interpolated traces have the same general character as the given values. The filter came out slightly different from the $(1, 0, -1)$ that I suggested for Figure 8.7 based on a subjective analysis. Curiously, constraining the filter to be of the form $(a_{-2}, a_{-1}, 1)$ in Figure 8.9 yields the same interpolated missing data as in Figure 8.8. I understand that the sum squared of the coefficients of $A(Z)P(Z)$ is the same as that of $A(1/Z)P(Z)$, but I do not see why that would imply the same interpolated data.

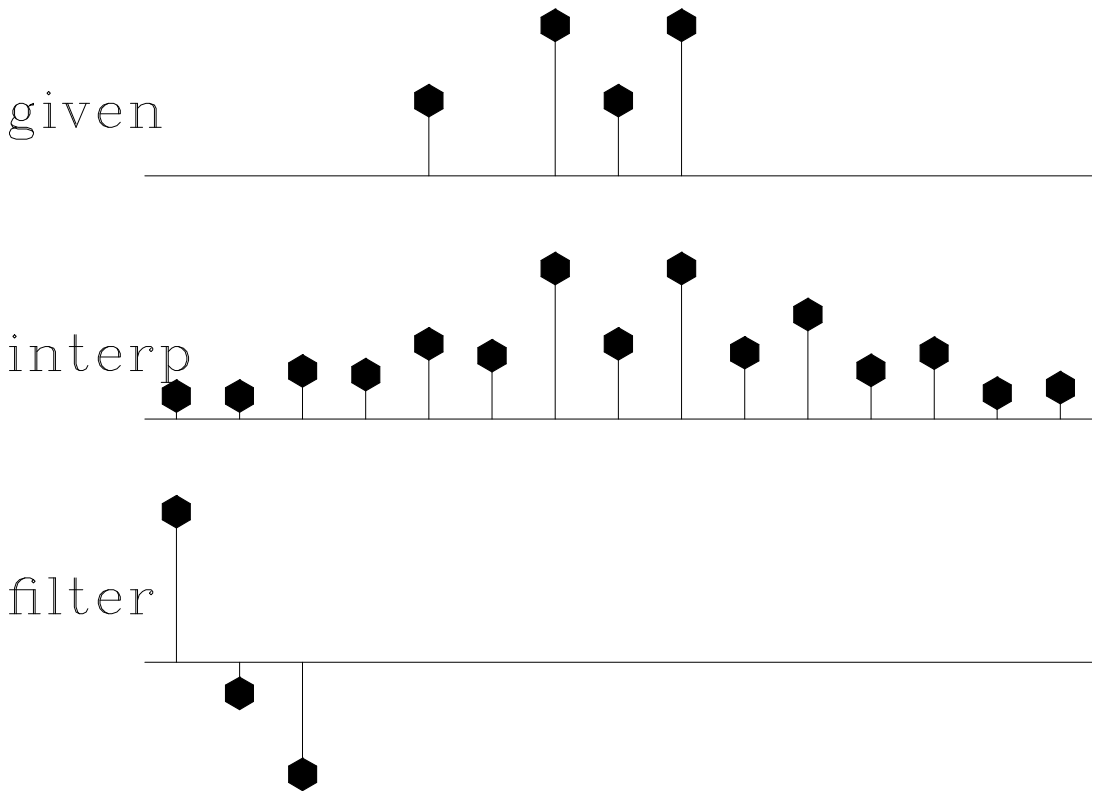


Figure 8.8: Top is known data. Middle includes the interpolated values. Bottom is the filter with the leftmost point constrained to be unity and other points chosen to minimize output power. mis-missif [ER]

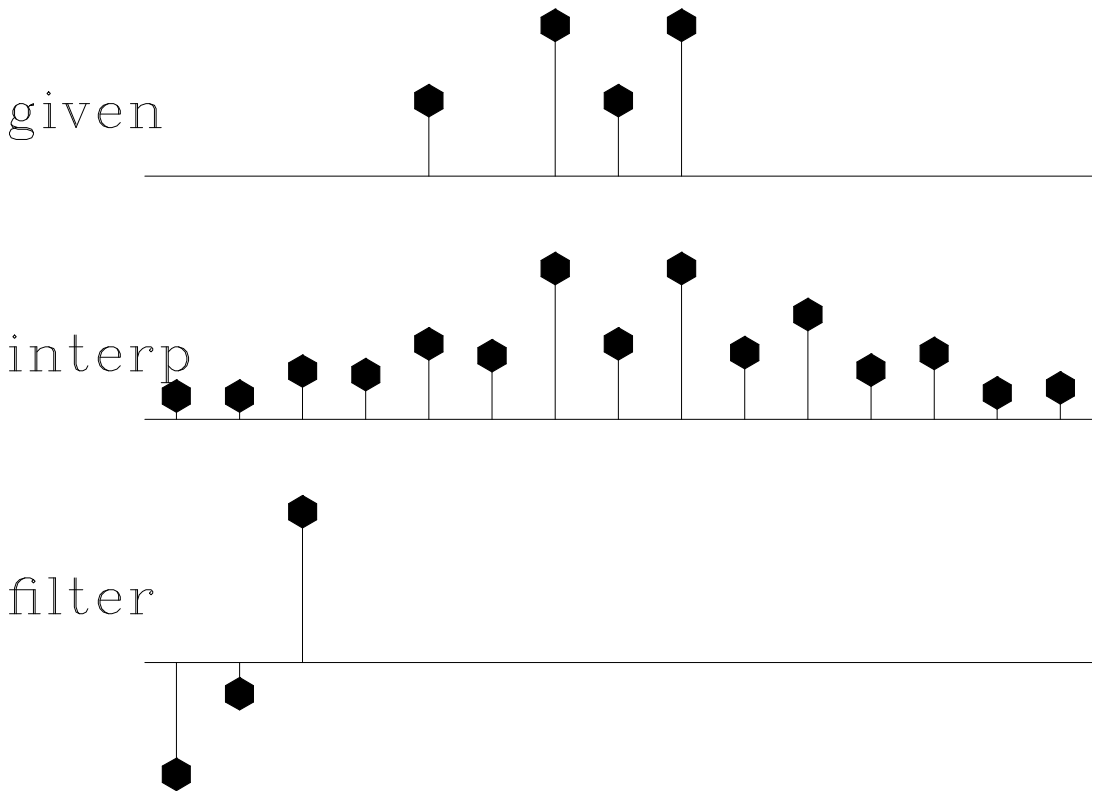


Figure 8.9: The filter here had its rightmost point constrained to be unity—i.e., this filtering amounts to backward prediction. The interpolated data seems to be identical, as with forward prediction. mis-backwards [ER]

8.3.1. Objections to interpolation error

In any data interpolation or extrapolation, we want the extended data to behave like the original data. And, in regions where there is no observed data, the extrapolated data should drop away in a fashion consistent with its **spectrum** determined from the known region. We will see that a filter like $(a_{-2}, a_{-1}, 1, a_1, a_2)$ fails to do the job. We need to keep an *end* value constrained to "1," not the middle value.

In chapter 7 we learned about the **interpolation-error filter** (IE filter), a filter constrained to be "+1" near the middle and consisting of other coefficients chosen to minimize the power out. The basic fact about the IE filter is that the spectrum out tends to the inverse of the spectrum in, so the spectrum of the IE filter tends to the inverse *squared* of the spectrum in. The IE filter is thus not a good weighting function for a minimization, compared to the prediction-error (PE) filter, whose spectrum is inverse to the input. To confirm these concepts, I prepared synthetic data consisting of a fragment of a damped exponential, and off to one side of it an impulse function. Most of the energy is in the damped exponential. Figure 8.10 shows that the spectrum and the extended data are about what we would expect. From the extrapolated data, it is impossible to see where the given data ends. For comparison, I prepared Figure 8.11. It is the same as Figure 8.10, except that the

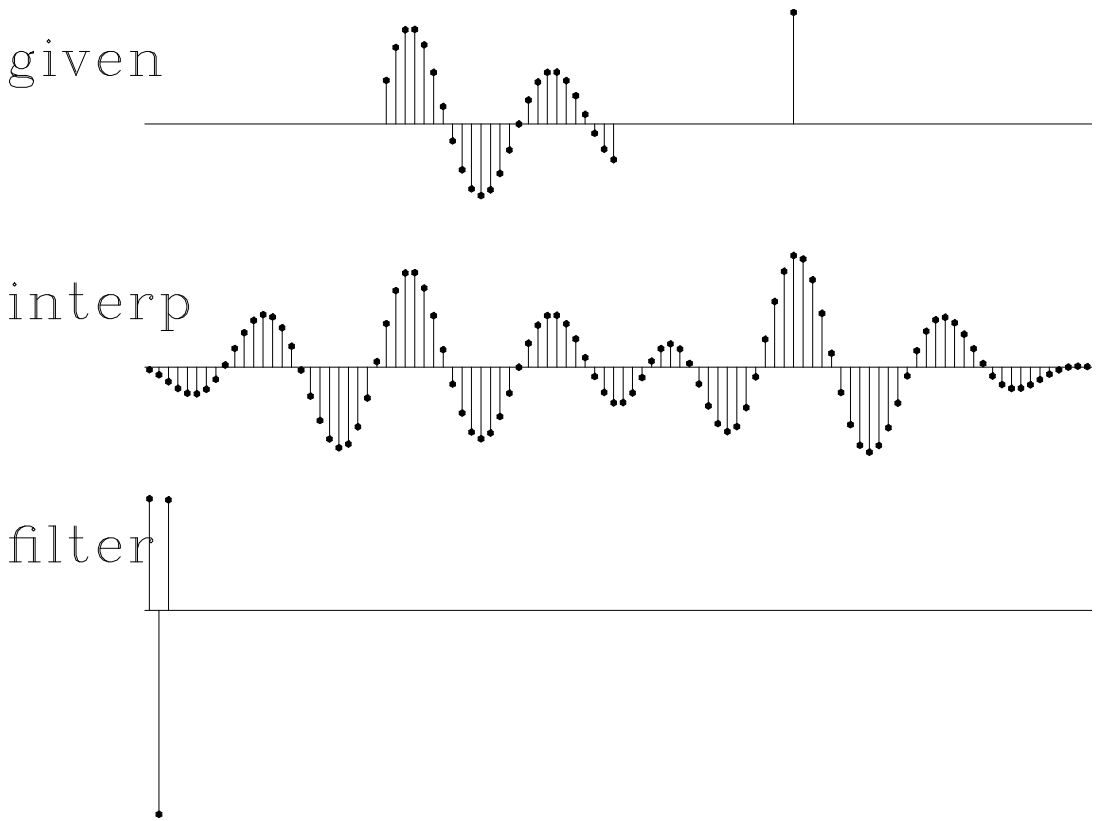


Figure 8.10: Top is synthetic data with missing data represented by zeros. Middle includes the interpolated values. Bottom is the filter, a prediction-error filter which may look symmetric but is not quite. mis-exp [ER]

filter is constrained in the middle. Notice that the extended data does *not* have the spectrum of the given data—the wavelength is much shorter. The boundary between real data and extended data is not nearly as well hidden as in Figure 8.10.

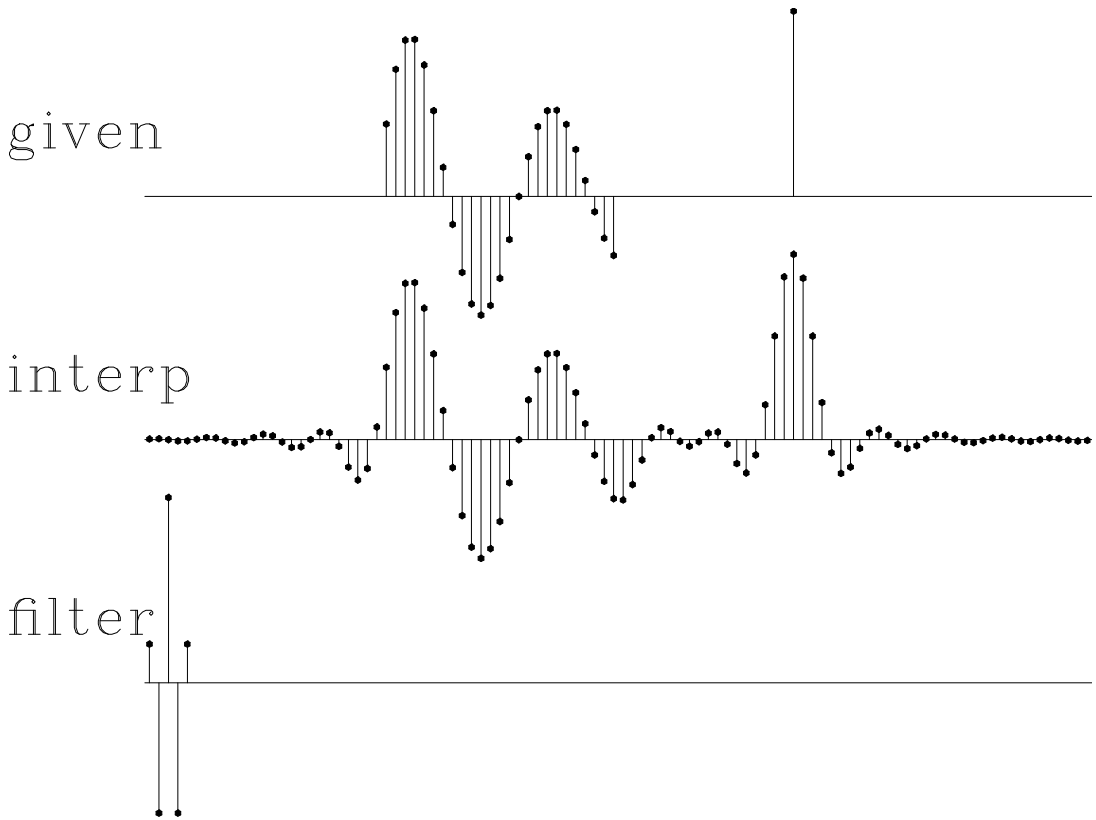


Figure 8.11: Top is synthetic data with missing data represented by zeros. Middle includes the interpolated values. Bottom is the filter, an interpolation-error filter. mis-center [ER]

Next I will pursue some esoteric aspects of one-dimensional missing-data prob-

lems. You might prefer to jump forward to section 8.4, where we tackle two-dimensional analysis.

8.3.2. Packing both missing data and filter into a CG vector

Now let us examine the theory and coding behind the above examples. Define a roughening filter $A(Z)$ and a data signal $P(Z)$ at some stage of interpolation. The regression is $0 \approx A(Z)P(Z)$ where the filter $A(Z)$ has at least one coefficient constrained to be nonzero and the data contains both known and missing values. Think of perturbations ΔA and ΔP. We neglect the nonlinear term $\Delta A \Delta P$ as follows:

$$0 \approx (A + \Delta A)(P + \Delta P) \tag{8.1}$$

$$0 \approx AP + P\,\Delta A + A\,\Delta P + \Delta A\,\Delta P \tag{8.2}$$

$$-AP \approx P\,\Delta A + A\,\Delta P \tag{8.3}$$

To make a program such as `miss1()` /prog:miss1, we need to **pack** both unknowns into a single vector `x()` $= (\Delta P, \Delta A)$ before calling the conjugate-gradient program. Likewise, the resulting filter and data coming out must be unpacked. Also, the

```
# MISSIF -- find MISSing Input data and Filter on 1-axis by min power out.
#
subroutine missif( na, lag, aa, np, pp, known, niter)
integer iter,      na, lag,     np,            niter, nx, ax, px, ip, nr
real    pp(np)           # input: known data with zeros for missing values.
                         # output: known plus missing data.
real    known(np)        # known(ip) vanishes where p(ip) is a missing value.
real    aa(na)           # input and output: roughening filter
temporary real  x(np+na), g(np+na), s(np+na)
temporary real  rr(np+na-1), gg(np+na-1), ss(np+na-1)
nr= np+na-1;    nx= np+na;      px=1;   ax=1+np;
call copy( np,  pp, x(px))
call copy( na,  aa, x(ax))
if( aa(lag) == 0. )     call erexit('missif: a(lag)== 0.')
do iter= 0, niter {
        call contran( 0, 0, na,aa, np, pp,    rr)
        call scaleit  (          -1., nr,       rr)
        call contran( 1, 0, na,aa, np, g(px), rr)
        call contran( 1, 0, np,pp, na, g(ax), rr)
        do ip= 1, np
                if( known(ip) != 0)
                        g( ip) = 0.
        g( lag+np) = 0.
        call contran( 0, 0, na,aa, np, g(px), gg)
        call contran( 0, 1, np,pp, na, g(ax), gg)
        call cgstep( iter, nx, x, g, s, nr, rr, gg, ss)
        call copy( np, x(px), pp)
        call copy( na, x(ax), aa)
        }
return; end
```

Back

gradient now has two contributions, one from $A\,\Delta P$ and one from $P\,\Delta A$, and these must be combined. The program `missif()`, which makes Figures 8.8 through 8.11, effectively combines `miss1()` /prog:miss1 and `iner()` /prog:iner. A new aspect is that, to avoid accumulation of errors from the neglect of the **nonlinear** product $\Delta A\,\Delta P$, the residual is recalculated inside the iteration loop instead of only once at the beginning. missif

There is a danger that `missif()` might converge very slowly or fail if `aa()` and `pp()` are much out of scale with each other, so be sure you input them with about the same scale. I really should revise the code, perhaps to scale the "1" in the filter to the data, perhaps to equal the square root of the sum of the data values.

8.3.3. Spectral preference and training data

I tried using the `missif()` program to **interlace** data—i.e., to put new data values between each given value. This did not succeed. The interlaced missing values began equaling zero and remained zero. Something is missing from the problem formulation.

This paragraph describes only the false starts I made toward the solution. It

seems that the filter should be something like $(1, -2, 1)$, because that filter interpolates on straight lines and is not far from the feedback coefficients of a damped sinusoid. (See equation (**??**).) So I thought about different ways to force the solution to move in that direction. Traditional **linear inverse theory** offers several suggestions; I puzzled over these before I found the right one. First, I added the obvious stabilizations $\lambda_1^2||\mathbf{p}||$ and $\lambda_2^2||\mathbf{a}||$, but they simply made the filter and the interpolated data smaller. I thought about changing the identity matrix in $\lambda\mathbf{I}$ to a diagonal matrix $||\Lambda_3\mathbf{p}||$ or $||\Lambda_4\mathbf{a}||$. Using Λ_4, I could penalize the filter at even-valued lags, hoping that it would become nonzero at odd lags, but that did not work. Then I thought of using $\lambda_5^2||\mathbf{p}-\bar{\mathbf{p}}||$, $\lambda_6^2||\mathbf{a}-\bar{\mathbf{a}}||$, $\Lambda_7^2||\mathbf{p}-\bar{\mathbf{p}}||$, and $\Lambda_8^2||\mathbf{a}-\bar{\mathbf{a}}||$, which would allow freedom of choice of the **mean** and **variance** of the unknowns. In that case, I must supply the mean and variance, however, and doing that seems as hard as solving the problem itself. Suddenly, I realized the answer. It is simpler than anything above, yet formally it seems more complicated, because a full inverse **covariance matrix** of the unknown filter is implicitly supplied.

I found a promising new approach in the **stabilize**d minimization

$$\min_{P,A} \; (||PA|| \; + \; \lambda_9||P_0A|| \; + \; \lambda_{10}||PA_0|| \;) \tag{8.4}$$

where P_0 and A_0 are like given *priors*. But they are not prior estimates of P and A because the phases of P_0 and A_0 are irrelevant, washing out in the squaring. If we specify large values for λ, the overall problem becomes more linear, so P_0 and A_0 give a way to impose **uniqueness** in a **nonlinear** case where uniqueness is otherwise unknown. Then, of course, the λ values can be reduced to see where the nonlinear part $||PA||$ is leading.

The next question is, what are the appropriate definitions for P_0 and A_0? Do we need both P_0 and A_0, or is one enough? We will come to understand P_0 and A_0 better as we study more examples. Simple theory offers some indications, however. It seems natural that P_0 should have the spectrum that we believe to be appropriate for P. We have little idea about what to expect for A, except that its spectrum should be roughly inverse to P.

To begin with, I think of P_0 as a low-pass filter, indicating that data is normally oversampled. Likewise, A_0 should resemble a high-pass filter. When we turn to two-dimensional problems, I will guess first that P_0 is a low-pass *dip* filter, and A_0 a high-pass dip filter.

Returning to the one-dimensional signal-interlacing problem, I take $A_0 = 0$ and choose P_0 to be a *different* dataset, which I will call the "**training data**." It is

a small, additional, theoretical dataset that has no missing values. Alternately, the training data could come from a large collection of observed data that is without missing parts. Here I simply chose the short signal $(1, 1)$ that is *not* interlaced by zeros. This gives the fine solution we see in Figure 8.12.

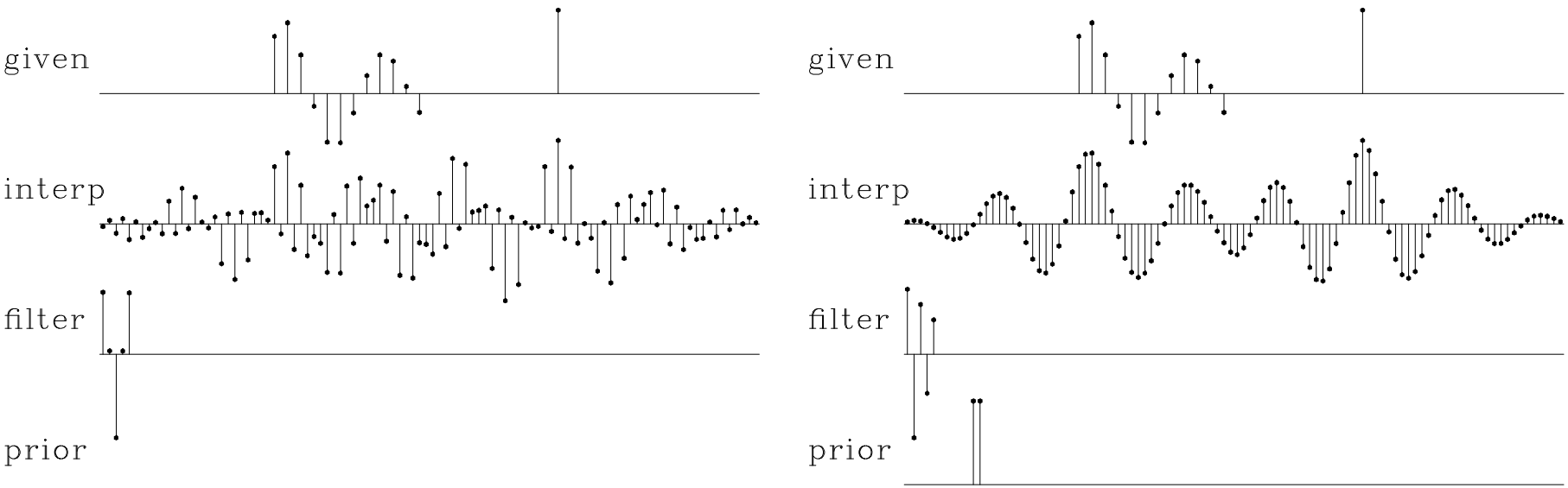


Figure 8.12: Left shows that data will not interlace without training data. Right shows data being interlaced because of training data. mis-priordata [ER]

To understand the coding implied by the optimization (8.4), it is helpful to write the linearized regression. The training signal P_0 enters as a matrix of shifted columns of the training signal, say $\mathbf{T}$; and the high-pass filter A_0 also appears as shifted columns in a matrix, say $\mathbf{H}$. The unknowns A and P appear both in the matrices $\mathbf{A}$ and $\mathbf{P}$ and in vectors $\mathbf{a}$ and $\mathbf{p}$. The **linearized regression** is

$$\begin{bmatrix} -\mathbf{Pa} \\ -\mathbf{Hp} \\ -\mathbf{Ta} \end{bmatrix} \approx \begin{bmatrix} \mathbf{A} & \mathbf{P} \\ \mathbf{H} & \mathbf{0} \\ \mathbf{0} & \mathbf{T} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{p} \\ \Delta\mathbf{a} \end{bmatrix} \tag{8.5}$$

The top row restates equation (8.3). The middle row says that $\mathbf{0} = \mathbf{H}(\mathbf{p} + \Delta\mathbf{p})$, and the bottom row says that $\mathbf{0} = \mathbf{T}(\mathbf{a} + \Delta\mathbf{a})$. A program that does the job is `misfip()` /prog:misfip. It closely resembles `missif()` /prog:missif. misfip The new computations are the lines containing the training data `tt`. (I omitted the extra clutter of the high-pass filter `hh` because I did not get an interesting example with it.) Compared to `missif()` /prog:missif, additional clutter arises from pointers needed to partition the residual and the gradient abstract vectors into three parts, the usual one for $||P\,A||$ and the new one for $||P_0\,A||$ (and potentially $||P\,A_0||$).

You might wonder why we need another program when we could use the old

```
# MISFIP --- find MISsing peF and Input data on 1-axis using Prior data.
#
subroutine misfip( nt,tt, na,aa, np,pp,known, niter)
integer nt, na,  ip,np,  npa, nta, nx,nr, iter,niter,  ax, px, qr, tr
real    pp(np), known(np), aa(na)       # same as in missif()
real    tt(nt)                          # input: prior training data set.
temporary real   x(np+na),             g(np+na),             s(np+na)
temporary real  rr(np+na-1 +na+nt-1), gg(np+na-1 +na+nt-1), ss(np+na-1 +na+nt-1)
npa= np+na-1;   nta= nt+na-1            # lengths of outputs of filtering
nx = np+na;     nr= npa+nta             # length of unknowns and residuals
px=1;           qr=1;   ax=1+np;        tr=1+npa        # pointers
call zero( na, aa);     aa(1) = 1.
call copy( np,  pp, x(px))
call copy( na,  aa, x(ax))
do iter= 0, niter {
        call contran( 0, 0, na,aa, np, pp,    rr(qr))
        call contran( 0, 0, na,aa, nt, tt,    rr(tr))  # extend rr with train
        call scaleit( -1.,            nr,       rr    )
        call contran( 1, 0, na,aa, np, g(px), rr(qr))
        call contran( 1, 0, np,pp, na, g(ax), rr(qr))
        call contran( 1, 1, nt,tt, na, g(ax), rr(tr))
        do ip= 1, np {  if( known(ip) != 0) { g( ip+(px-1)) = 0. } }
                                            g( 1 +(ax-1)) = 0.
        call contran( 0, 0, na,aa, np, g(px), gg(qr))
        call contran( 0, 1, np,pp, na, g(ax), gg(qr))
        call contran( 0, 0, nt,tt, na, g(ax), gg(tr))
        call cgstep( iter,  nx, x, g, s,  nr, rr, gg, ss)
        call copy( np, x(px), pp)
        call copy( na, x(ax), aa)
        }
return; end
```

Back

program and simply append the training data to the observed data. We will encounter some applications where the old program will not be adequate. These involve the boundaries of the data. (Recall that, in chapter 4, when seismic events changed their dip, we used a two-dimensional wave-killing operator and were careful not to convolve the operator over the edges.) Imagine a dataset that changes with time (or space). Then P_0 might not be training data, but data from a large interval, while P is data in a tiny window that is moved around on the big interval. These ideas will take definite form in two dimensions.

8.3.4. Summary of 1-D missing-data restoration

Now I will summarize our approach to 1-D missing-data restoration in words that will carry us towards 2-D missing data. First we noticed that, given a filter, minimizing the output power will find missing input data regardless of the volume missing or its geometrical complexity. Second, we experimented with various filters and saw that the **prediction-error filter** is an appropriate choice, because data extensions into regions without data tend to have the spectrum inverse to the PE filter, which (from chapter 7) is inverse to the known data. Thus, the overall problem is

perceived as a **nonlinear** one, involving the product of unknown filter coefficients and unknown data. It is well known that nonlinear problems are susceptible to multiple solutions; hence the importance of the stabilization method described, which enables us to ensure a reasonable solution.

8.3.5. 2-D interpolation before aliasing

A traditional method of data interpolation on a regular mesh is a four-step procedure: (1) set zero values at the points to be interpolated; (2) Fourier transform; (3) set to zero the high frequencies; and (4) inverse transform. This is a fine method and is suitable for many applications in both one dimension and higher dimensions. Where the method falls down is where more is needed than simple interlacing—for example, when signal values are required beyond the ends of the data sample. The simple Fourier method of interlacing also loses its applicability when known data is irregularly distributed. An example of an application in two dimensions of the methodology of this section is given in the section on tomography beginning on page 504.

8.4. 2-D INTERPOLATION BEYOND ALIASING

I have long marveled at the ability of humans to interpolate seismic data containing mixtures of dips where spatial frequencies exceed the Nyquist limits. These limits are hard limits on migration programs. Costly field-data-acquisition activities are designed with these limits in mind. I feared this human skill of going beyond the limit was deeply nonlinear and beyond reliable programming. Now, however, I have obtained results comparable in quality to those of S. **Spitz**, and I am doing so in a way that seems reliable—using two-stage, linear least squares. First we will look at some results and then examine the procedure. Before this program can be applied to field data for migration, remember that the data must be broken into many overlapping tiles of about the size shown here and the results from each tile pieced together.

Figure 8.13 shows three plane waves recorded on five channels and the interpolated data. Both the original data and the interpolated data can be described as "beyond **alias**ing" because on the input data the signal shifts exceed the signal duration. The calculation requires only a few seconds of a "two-stage least-squares" method, where the first stage estimates an inverse **covariance matrix** of the known data, and the second uses it to estimate the missing traces. Actually, a **2-D prediction-error**

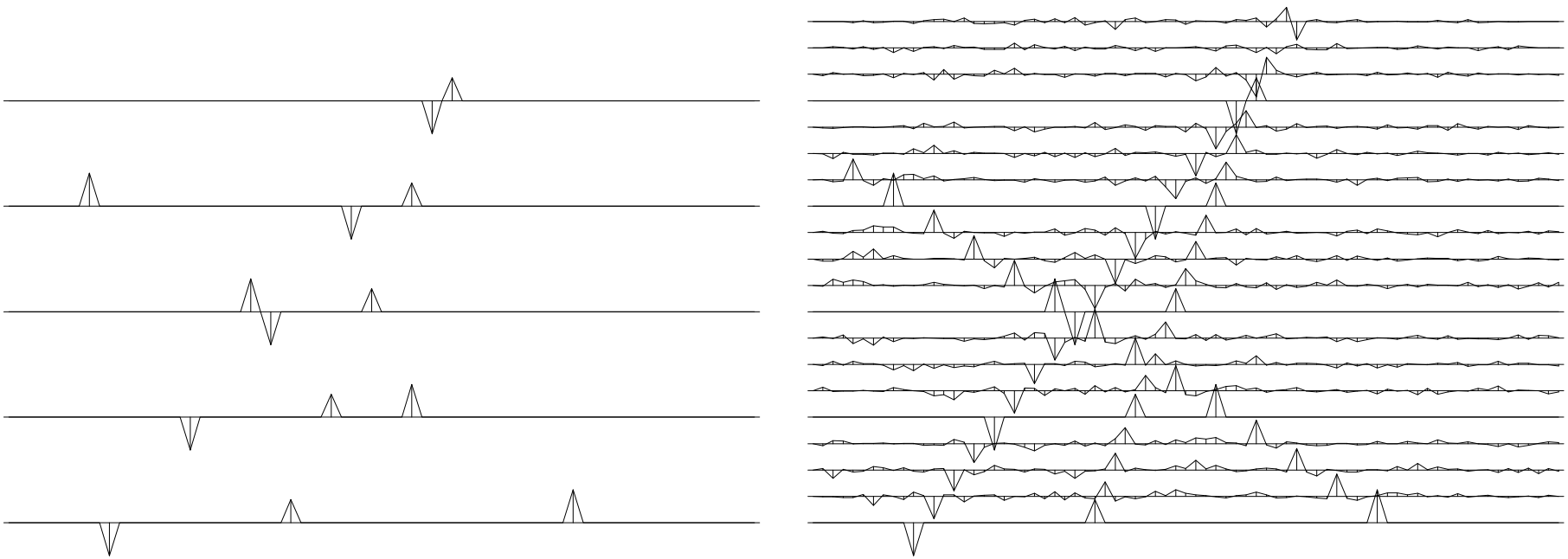


Figure 8.13: Left is five signals, each showing three arrivals. Using the data shown on the left (and no more), the signals have been interpolated. Three new traces appear between each given trace as shown on the right. mis-lace3 [ER]

filter is estimated, and the inverse covariance matrix, which amounts to the PE filter times its adjoint, is not needed explicitly.

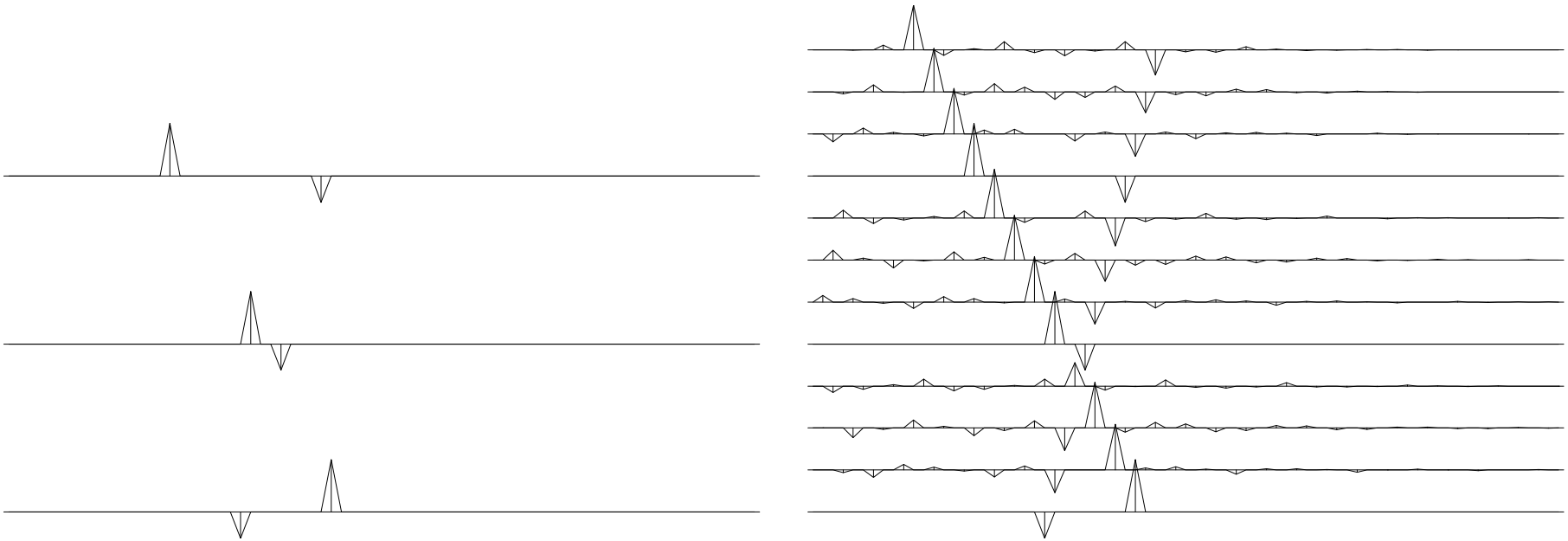


Figure 8.14: Two plane waves and their interpolation. mis-lace2 [ER]

Let us now examine a case with minimal complexity. Figure 8.14 shows two **plane wave**s recorded on three channels. That is the minimum number of channels

required to distinguish two superposing plane waves. Notice on the interpolated data that the original traces are noise-free, but the new traces have acquired a low level of noise. This will be dealt with later.

Figure 8.15 shows the same calculation in the presence of noise on the original data. We see that the noisy data is interpolatable just as was the noise-free data, but now we can notice the organization of the noise. It has the same slopes as the plane waves. This was also true on the earlier figures (Figure 8.13 and 8.14), as is more apparent if you look at the page from various grazing angles. To display the slopes more clearly, Figure 8.15 is redisplayed in a raster mode in Figure 8.16.

8.4.1. Interpolation with spatial predictors

A **two-dimensional filter** is a small plane of numbers that is convolved over a big data plane of numbers. One-dimensional convolution can use the mathematics of **polynomial multiplication**, such as $Y(Z) = X(Z)F(Z)$, whereas two-dimensional convolution can use something like $Y(Z_1, Z_2) = X(Z_1, Z_2)F(Z_1, Z_2)$. Polynomial mathematics is appealing, but unfortunately it implies transient **edge** conditions, whereas here we need different edge conditions, such as those of the dip-rejection

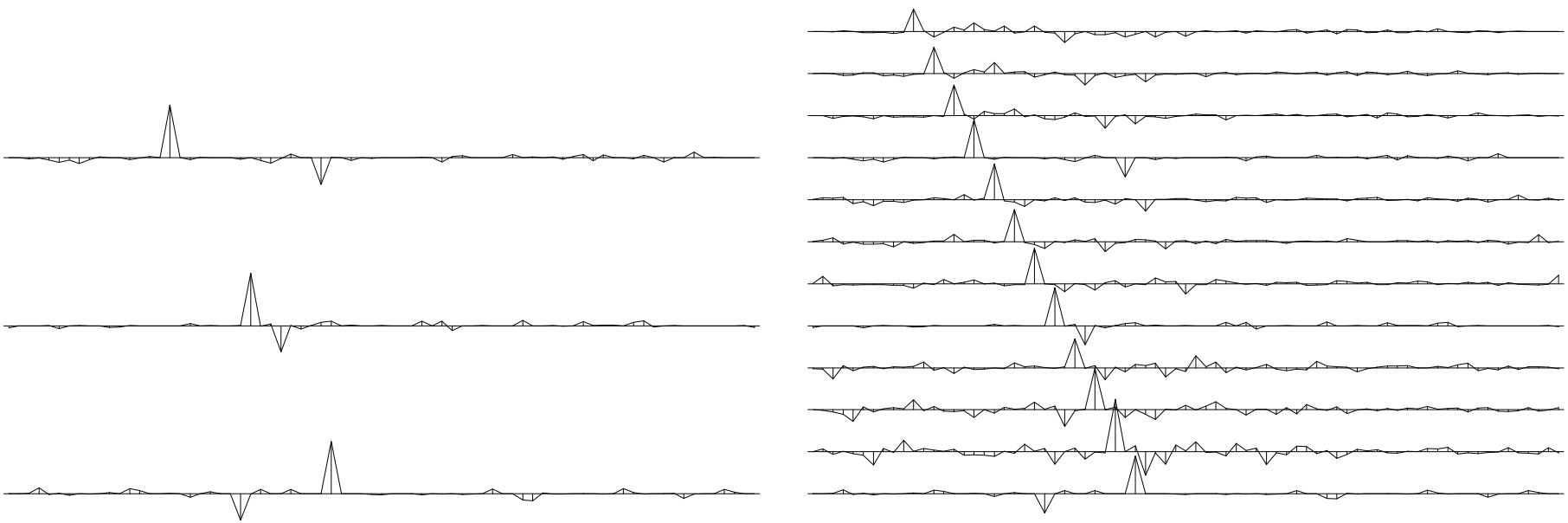


Figure 8.15: Interpolating noisy plane waves. mis-lacenoise [ER]

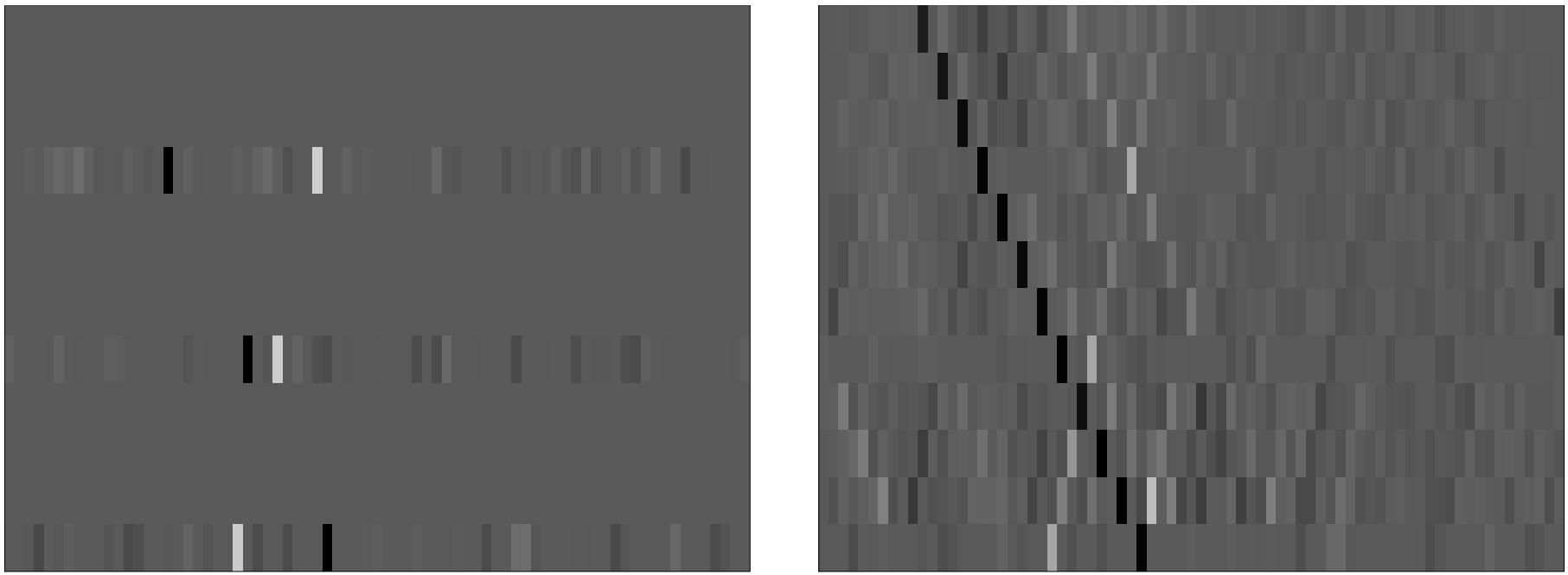


Figure 8.16: Interpolating noisy plane waves. mis-laceras [ER]

filters discussed in Chapter 4, which were based on simple partial differential equations. Here we will examine **spatial prediction-error filters** (2-D PE filters) and see that they too can behave like dip filters.

The typesetting software I am using has no special provisions for two-dimensional filters, so I will set them in a little table. Letting "·" denote a zero, we denote a **two-dimensional filter** that can be a dip-rejection filter as

$$\begin{array}{ccccc} a & b & c & d & e \\ \cdot & \cdot & 1 & \cdot & \cdot \end{array} \tag{8.6}$$

where the coefficients (a,b,c,d,e) are to be estimated by least squares in order to minimize the power out of the filter. (In the table, the time axis runs horizontally, as on data displays.)

Fitting the filter to two neighboring traces that are identical but for a time shift, we see that the filter (a,b,c,d,e) should turn out to be something like $(-1,0,0,0,0)$ or $(0,0,-.5,-.5,0)$, depending on the dip (stepout) of the data. But if the two channels are not fully coherent, we expect to see something like $(-.9,0,0,0,0)$ or $(0,0,-.4,-.4,0)$. For now we will presume that the channels are fully coherent.

8.4.2. Refining both t and x with a spatial predictor

Having determined a 2-D filter, say on the original data **mesh**, we can now **interlace** both t and x and expect to use the identical filter. This is because slopes are preserved if we replace $(\Delta t, \Delta x)$ by $(\Delta t/2, \Delta x/2)$. Everyone knows how to interpolate data on the time mesh, so that leaves the job of interpolation on the space mesh: in (8.6) the known (a,b,c,d,e) can multiply a known trace, and then the "1" can multiply the interlaced and unknown trace. It is then easy to minimize the power out by *defining* the missing trace to be the negative of that predicted by the filter (a,b,c,d,e) on the known trace. (The spatial interpolation problem seems to be solved regardless of the amount of the signal shift. A "spatial aliasing" issue does not seem to arise.) It is nice to think of the unknowns being under the "1" and the knowns being under the (a,b,c,d,e), but the CG method has no trouble working backwards too.

After I became accustomed to using the CG method, I stopped thinking that the unknown data is that which is predicted, and instead began to think that the unknown data is that which minimizes the power out of the prediction filter. I ignored the question of which data values are known and which are unknown. This thinking enables a reformulation of the problem, so that interpolation on the time axis is an

unnecessary step. This is the way all my programs work. Think of the filter that follows as applied on the original coarse-mesh data:

$$\begin{array}{ccccccccc} a & \cdot & b & \cdot & c & \cdot & d & \cdot & e \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \end{array} \tag{8.7}$$

The first stage is to use CG to find (a,b,c,d,e) in (8.7). For the second stage, we assert that the same values (a,b,c,d,e) found from (8.7) can be used in (8.6), and we use CG a second time to find the missing data values. A wave field interpolated this way is shown in Figure 8.17. Figures 8.13 to 8.16 were made with filters that had more rows than (8.7), for reasons we will discuss next.

8.4.3. The prediction form of a two-dip filter

Now we handle two **dip**s simultaneously. The following filter destroys a wave that is sloping down to the right:

$$\begin{array}{ccc} -1 & \cdot & \cdot \\ \cdot & \cdot & 1 \end{array} \tag{8.8}$$

Figure 8.17: Two signals with one dip. mis-lace1 [ER]

The next filter destroys a wave that is sloping less steeply down to the left:

$$\begin{matrix} \cdot & -1 \\ 1 & \cdot \end{matrix} \tag{8.9}$$

Convolving the above two filters together, we get

$$\begin{matrix} \cdot & 1 & \cdot & \cdot \\ -1 & \cdot & \cdot & -1 \\ \cdot & \cdot & 1 & \cdot \end{matrix} \tag{8.10}$$

The 2-D filter (8.10) destroys waves of both slopes. Given appropriate interlacing, the filter (8.10) destroys the data in Figure 8.14 both before *and* after interpolation. To find filters such as (8.10), I adjust coefficients to minimize the power out of filters like

$$\begin{matrix} v & w & x & y & z \\ a & b & c & d & e \\ \cdot & \cdot & 1 & \cdot & \cdot \end{matrix} \tag{8.11}$$

A filter of this shape is suitable for figures like 8.14 and 8.15.

Let us examine the Fourier domain for this filter. The filter (8.10) was transformed to the Fourier domain; it was multiplied by its conjugate; the square root

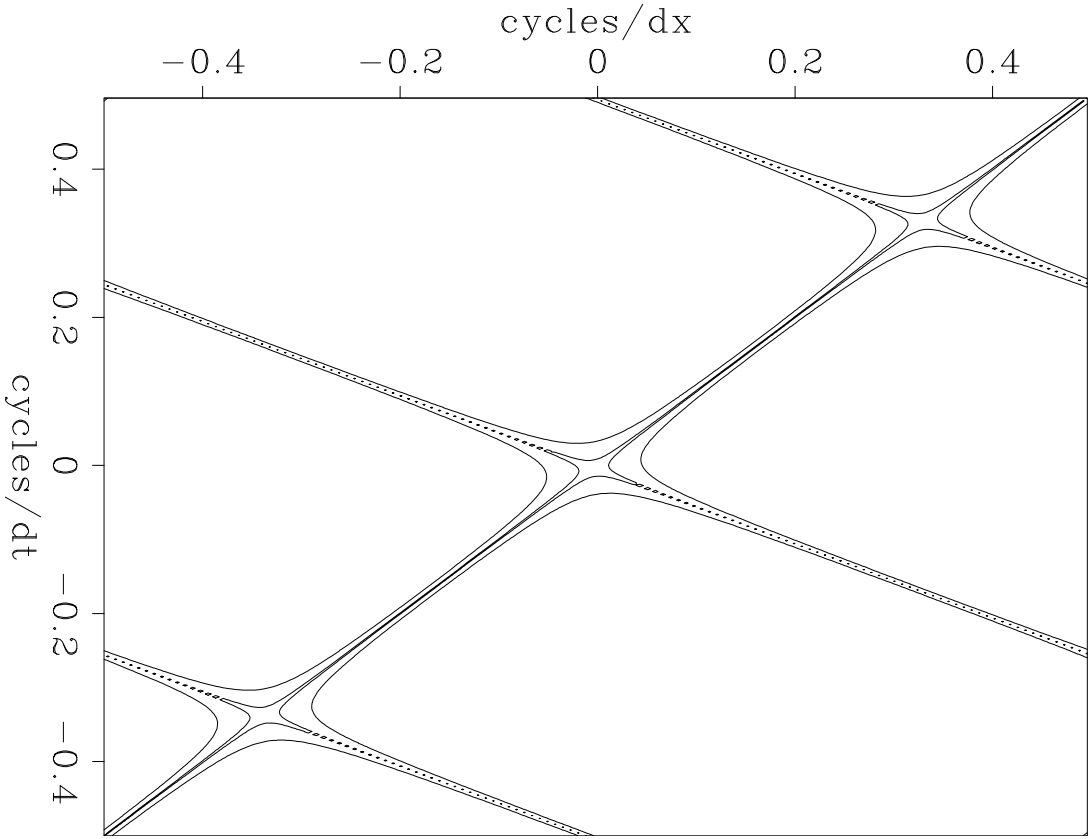


Figure 8.18: Magnitude of two-dimensional Fourier transform of the 2-D filter contoured at .01 and at .1. mis-fk2dip [ER]

```
# CINJOF --- Convolution INternal with Jumps. Output and FILTER are adjoint.
#
subroutine cinjof( adj, add, jump,   n1,n2,xx, nb1,nb2,bb, yy)
integer            adj, add, jump,   n1,n2,    nb1,nb2    # jump subsamples data
real                                  xx( n1,n2),  bb( nb1,nb2),  yy( n1,n2)
integer y1,y2, x1,x2, b1, b2, ny1, ny2
call adjnull( adj, add,                                  bb, nb1*nb2,   yy, n1*n2)
ny1 = n1 - (nb1-1) * jump;              if( ny1<1 ) call erexit('cinjof: ny1<1')
ny2 = n2 - (nb2-1);                     if( ny2<1 ) call erexit('cinjof: ny2<1')
if( adj == 0 )
        do b2=1,nb2 { do y2=1,ny2 {     x2 =  y2 - (b2-nb2)
        do b1=1,nb1 { do y1=1,ny1 {     x1 =  y1 - (b1-nb1) * jump
                                yy(y1,y2) = yy(y1,y2) + bb(b1,b2) * xx(x1,x2)
                        }} }}
else
        do b2=1,nb2 { do y2=1,ny2 {     x2 =  y2 - (b2-nb2)
        do b1=1,nb1 { do y1=1,ny1 {     x1 =  y1 - (b1-nb1) * jump
                                bb(b1,b2) = bb(b1,b2) + yy(y1,y2) * xx(x1,x2)
                        }} }}
return; end
```

Back

was taken; and contours are plotted at near-zero magnitudes in Figure 8.18. The slanting straight lines have slopes at the two dips that are destroyed by the filters. Noticing the broad lows where the null lines cross, we might expect to see energy at this temporal and spatial frequency, but I have not noticed any. cinjof

In practice, wavefronts have **curvature**, so we will estimate the 2-D filters in many small windows on a wall of data. Therefore, to eliminate edge effects, I designed the 2-D filter programs starting from the 1-D internal convolution program `convin()` /prog:convin. The subroutine for two-dimensional filtering is `cinjof()` /prog:cinjof. The adjoint operation included in this subroutine is exactly what we need for estimating the filter.

A companion program, `cinloi()`, is essentially the same as `cinjof()`, except that in `cinloi()` the other adjoint is used (for unknown input instead of unknown filter), and there is no need to interlace the time axis. A new feature of `cinloi()` is that it arranges for the output residuals to come out directly on top of their appropriate location on the original data. In other words, the output of the filter is at the "1." Although the edge conditions in this routine are confusing, it should be obvious that `xx(,)` is aligned with `yy(,)` at `bb(lag1,lag2)`. cinloi

```
# CINLOI --- Convolution INternal with Lags.  Output is adjoint to INPUT.
#
subroutine cinloi( adj, add, lag1,lag2,nb1,nb2,bb,  n1,n2, xx, yy)
integer            adj, add, lag1,lag2,nb1,nb2,     n1,n2        # lag=1 causal
real                                        bb(nb1,nb2), xx(n1,n2), yy(n1,n2)
integer y1,y2, x1,x2, b1,b2
call adjnull(      adj, add,                             xx,n1*n2,  yy,n1*n2 )
if( adj == 0 )
        do b2=1,nb2 { do y2= 1+nb2-lag2, n2-lag2+1 {  x2= y2 - b2 + lag2
        do b1=1,nb1 { do y1= 1+nb1-lag1, n1-lag1+1 {  x1= y1 - b1 + lag1
                                   yy(y1,y2) = yy(y1,y2) + bb(b1,b2) * xx(x1,x2)
                                   }} }}
else
        do b2=1,nb2 { do y2= 1+nb2-lag2, n2-lag2+1 {  x2= y2 - b2 + lag2
        do b1=1,nb1 { do y1= 1+nb1-lag1, n1-lag1+1 {  x1= y1 - b1 + lag1
                                   xx(x1,x2) = xx(x1,x2) + bb(b1,b2) * yy(y1,y2)
                                   }} }}
return; end
```

Back

```
# Find spatial prediction-error filter.
#
subroutine pe2( eps, a1,a2,aa, n1,n2 ,pp, rr, niter, jump)
integer              a1,a2,    n1,n2,         niter, jump
integer i1, iter, midpt, r12, a12
real    aa( a1 , a2),  pp( n1 , n2),  rr( n1 , n2 * 2),  eps
temporary real da( a1, a2),  dr( n1, n2 * 2)
temporary real sa( a1, a2),  sr( n1, n2 * 2)
r12 = n1 * n2
a12 = a1 * a2
call null( aa, a12);    call null( rr, 2 * r12)
call null( da, a12);    call null( dr, 2 * r12)
midpt = (a1+1) / 2
aa( midpt, 1 ) = 1.
        call cinjof( 0, 0, jump, n1,n2,pp, a1,a2,aa, rr          )
        call ident ( 0, 0, eps,             a12,  aa, rr(1,n2+1) )
        call scaleit (       -1.,           2*r12,      rr          )
do iter= 0, niter {
        call cinjof( 1, 0, jump, n1,n2,pp, a1,a2,da, rr          )
        call ident ( 1, 1,  eps,            a12,  da, rr(1,n2+1) )
        do i1= 1, a1 {                                 da(i1, 1) = 0. }
        call cinjof( 0, 0, jump, n1,n2,pp, a1,a2,da, dr          )
        call ident ( 0, 0,  eps,            a12,  da, dr(1,n2+1) )
        call cgstep( iter,   a12, aa,da,sa, _
                           2*r12, rr,dr,sr )
        }
return; end
```

Back

8.4.4. The regression codes

The programs for the two-dimensional prediction-error filter and missing data resemble those for one dimension. I simplified the code by not trying to **pack** the unknowns and residuals tightly in the **abstract vector**s. Because of this, it is necessary to be sure those abstract vectors are initialized to zero. (Otherwise, the parts of the abstract vector that are not initialized could contribute to the result when `cgstep()` /prog:cgstep evaluates dot products on abstract vectors.) The routine `pe2()` /prog:pe2 finds the 2-D PE filter. pe2 This routine is the two-dimensional equivalent of finding the filter $A(Z)$ so that $0 \approx R(Z) = P(Z)A(Z)$. We coded the 1-D problem in `iner()` /prog:iner. In `pe2()`, however, I did not bother with the weighting functions. A further new feature of `pe2()` is that I added $\lambda\mathbf{I}$ capability (where λ is `eps`) by including the call to `ident()` /prog:ident, so that I could experiment with various forms of filter stabilization. (This addition did not seem to be helpful.)

Given the 2-D PE filter, the missing data is found with `miss2()` /prog:miss2, which is the 2-D equivalent of `miss1()` /prog:miss1. miss2 We will soon see that stabilization is more critical in `miss2()` than in `pe2()`. Furthermore, `miss2()` must be **stabilize**d with a **weighting function**, here `ww(,)`, which is why I used the di-

```
# fill in missing data in 2-D by minimizing power out of a given filter.
#
subroutine miss2(   lag1,lag2, a1,a2, aa, n1,n2, ww, pp, known, rr, niter)
integer i1,i2,iter, lag1,lag2, a1,a2,     n1,n2,                  niter, n12
real    pp(    n1, n2)          # in: known data with zeros for missing values
                                # out: known plus missing data.
real    known( n1, n2)          # in: known(ip) vanishes where pp(ip) is missing
real    ww(    n1, n2)          # in: weighting function on data pp
real    aa(    a1, a2)          # in: roughening filter
real    rr(    n1, n2*2)        # out: residual
temporary real  dp( n1, n2), dr( n1, n2*2)
temporary real  sp( n1, n2), sr( n1, n2*2)
n12 = n1 * n2;          call null( rr, n12*2)
call null( dp, n12);    call null( dr, n12*2)
        call cinloi( 0, 0, lag1,lag2,a1,a2,aa, n1,n2, pp, rr        )
        call diag  ( 0, 0,                 ww,  n12,  pp, rr(1,n2+1))
        call scaleit (-1.,                      2*n12,      rr       )
do iter= 0, niter {
        call cinloi( 1, 0, lag1,lag2,a1,a2,aa, n1,n2, dp, rr        )
        call diag  ( 1, 1,                 ww,  n12,  dp, rr(1,n2+1))
        do i1= 1, n1 {
        do i2= 1, n2 {  if( known(i1,i2) != 0.)       dp(i1,i2) = 0.
                        }}
        call cinloi( 0, 0, lag1,lag2,a1,a2,aa, n1,n2, dp, dr        )
        call diag  ( 0, 0,                 ww,  n12,  dp, dr(1,n2+1))
        call cgstep( iter,   n12, pp,dp,sp, _
                           2*n12, rr,dr,sr )
        }
return; end
```

Back

```
subroutine diag( adj, add, lambda,n,  pp,    qq)
integer i,       adj, add,        n                    # equivalence (pp,qq) OK
real                      lambda(n), pp(n), qq(n)
if( adj == 0 ) {
        if( add == 0 ) { do i=1,n {  qq(i) =         lambda(i) * pp(i) } }
        else           { do i=1,n {  qq(i) = qq(i) + lambda(i) * pp(i) } }
        }
else {  if( add == 0 ) { do i=1,n {  pp(i) =         lambda(i) * qq(i) } }
        else           { do i=1,n {  pp(i) = pp(i) + lambda(i) * qq(i) } }
        }
return; end
```

Back

agonal matrix multiplier `diag()` rather than the identity matrix I used in `deghost()` /prog:deghost and `pe2()` /prog:pe2. Subroutine `diag()` is used so frequently that I coded it in a special way to allow the input and output to overlie one another. diag

8.4.5. Zapping the null space with envelope scaling

Here we will see how to remove the small noise we are seeing in the interpolated outputs. The filter (8.10) obviously destroys the *input* in Figure 8.14. On the *output* interpolated data, the filter-output residuals (not shown) were all zeros despite the small noises. The filter totally extinguishes the small noise on the outputs because the noise has the same stepout (slope) as the signals. The noise is absent from the original traces, which are interlaced. How can dipping noises exist on the interpolated traces but be absent from the interlaced data? The reason is that one dip can interfere with another to cancel on the known, noise-free traces. The filter (8.10) destroys perfect output data as well as the noisy data in Figure 8.14. Thus, there is more than one solution to the problem. This is the case in linear equation solving whenever there is a null space. Since we manufactured many more data points than

we originally had, we should not be surprised by the appearance of a **null space**. When only a single **dip** is present, the null space should vanish because the dip vanishes on the known traces, having no other dips to interfere with it there. Confirm this by looking back at Figure 8.17, which contains no null-space noise. This is good news, because in real life, in any small window of seismic data, a single-dip model is often a good model.

If we are to eliminate the null-space noises, we will need some criterion in addition to stepout. One such criterion is *amplitude*: the noise events are the small ones. Before using a **nonlinear** method, we should be sure, however, that we have exploited the full power of linear methods. Information in the data is carried by the envelope functions, and these envelopes have not been included in the analysis so far. The **envelope**s can be used to make **weighting function**s. These weights are not weights on *residuals*, as in the routine `iner()` /prog:iner. These are weights on the *solution*. The $\lambda\mathbf{I}$ stabilization in routine `pe2()` /prog:pe2 applied uniform weights using the subroutine `ident()` /prog:ident, as has been explained. Here we simply apply variable weights Λ using the subroutine `diag()` /prog:diag. The weights themselves are the inverse of the envelope of input data (or the output of

a previous iteration). Where the envelope is small lies a familiar problem, which I approached in a familiar way—by adding a small constant. The result is shown in Figure 8.19. The top row is the same as Figure 8.13. The middle row shows the improvement that can be expected from weighting functions based on the inputs. So the middle row is the solution to a linear interpolation problem. Examining the envelope function on the middle left, we can see that it is a poor approximation to the envelope of the *output* data, but that is to be expected because it was estimated by smoothing the absolute values of the *input* data (with zeros on the unknown traces). The bottom row is a second stage of the process just described, where the new weighting function is based on the result in the middle row. Thus the bottom row is a **nonlinear** operation on the data.

When interpolating data, the number of unknowns is large. Here each row of data is 75 points, and there are 20 rows of missing data. So, theoretically, 1500 iterations might be required. I was getting good results with 15 conjugate-gradient iterations until I introduced weighting functions; then the required number of iterations jumped to about a hundred. The calculation takes seconds (unless the silly computer starts to underflow; then it takes me 20 times longer.)

I believe the size of the dynamic range in the weighting function has a con-

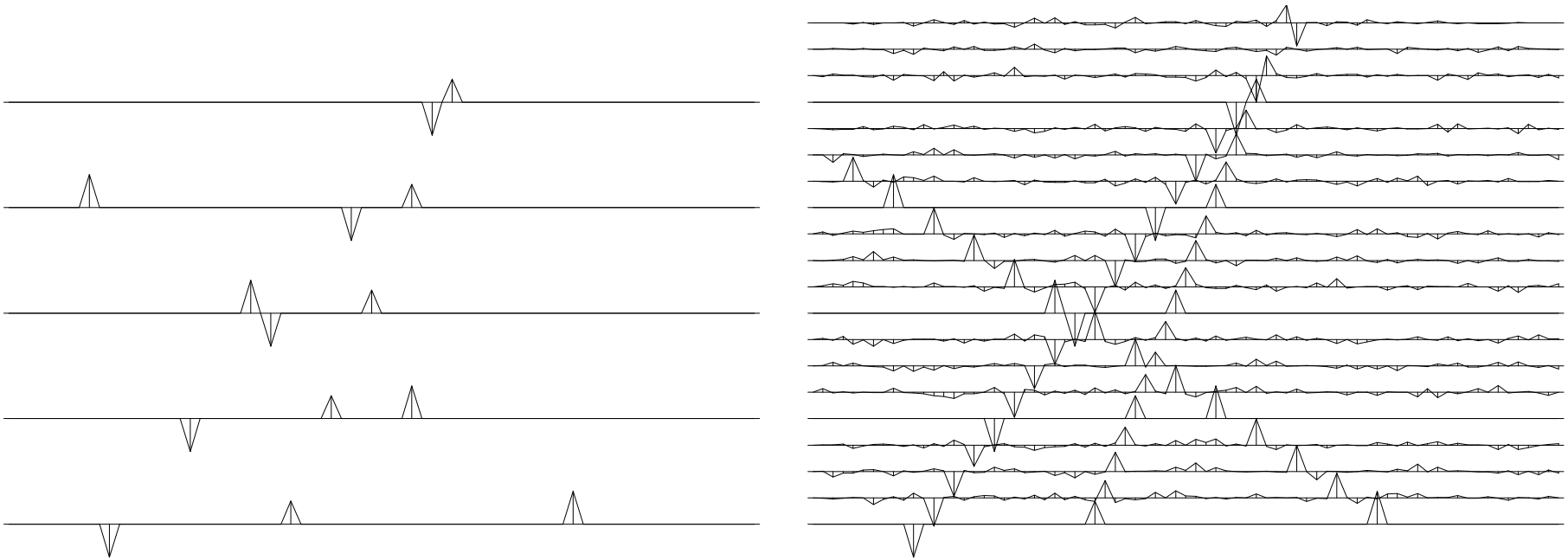


Figure 8.19: Top left is input. Top right is the interpolation with uniform weights. In the middle row are the envelope based on *input* data and the corresponding interpolated data. For the bottom row, the middle-row solution was used to design weights from which a near-perfect solution was derived. mis-wlace3 [ER]

trolling influence on the number of iterations. Before I made Figure 8.19, I got effectively the same result, and more quickly, using another method, which I abandoned because its philosophical foundation was crude. I describe this other method here only to keep alive the prospect of exploring the issue of the speed of convergence. First I moved the "`do iter`" line above the already indented lines to allow for the nonlinearity of the method. After running some iterations with $\Lambda = 0$ to ensure the emergence of some big interpolated values, I turned on Λ at values below a threshold. In the problem at hand, convergence speed is not important economically but is of interest because we have so little guidance as to how we can alter problem formulation in general to increase the speed of convergence.

8.4.6. Narrow-band data

Spitz's published procedure is to Fourier transform time to (ω, x), where, following Canales, he computes prediction filters along x for each ω. Spitz offers the insight that for a dipping event with stepout $p = k_x/\omega$, the prediction filter at trace separation Δx at frequency ω_0 should be identical to the prediction filter at trace separation $\Delta x/2$ at frequency $2\omega_0$. There is trouble unless both ω_0 and $2\omega_0$ have

reasonable signal-to-noise ratio. So a spectral band of good-quality data is required. It is not obvious that the same limitation applies to the interlacing procedure that I have been promoting, but I am certainly suspicious, and the possibility deserves inspection. Figure 8.20 shows a narrow-banded signal that is properly interpolated, giving an impressive result. It is doubtful that an observant human could have done as well. I found, however, that adding 10% noise caused the interpolation to fail.

On further study of Figure 8.20 I realized that it was not a stringent enough test. The signals obviously contain zero frequency, so they are not narrow-band in the sense of containing less than an octave. Much seismic data is narrow-band.

I have noticed that aspects of these programs are fragile. Allowing filters to be larger than they need to be to fit the waves at hand (i.e., allowing excess channels) can cause failure. We could continue to study the limitations of these programs. Instead, I will embark on an approach similar to the 1-D `missif()` /prog:missif program. That program is fundamentally **nonlinear** and so somewhat risky, but it offers us the opportunity to drop the idea of interlacing the filter. Interlacing is probably the origin of the requirement for good signal-to-noise ratio over a wide spectral band. Associated with interlacing is also a nagging doubt about plane waves that are imperfectly predictable from one channel to the next. When such data is interlaced,

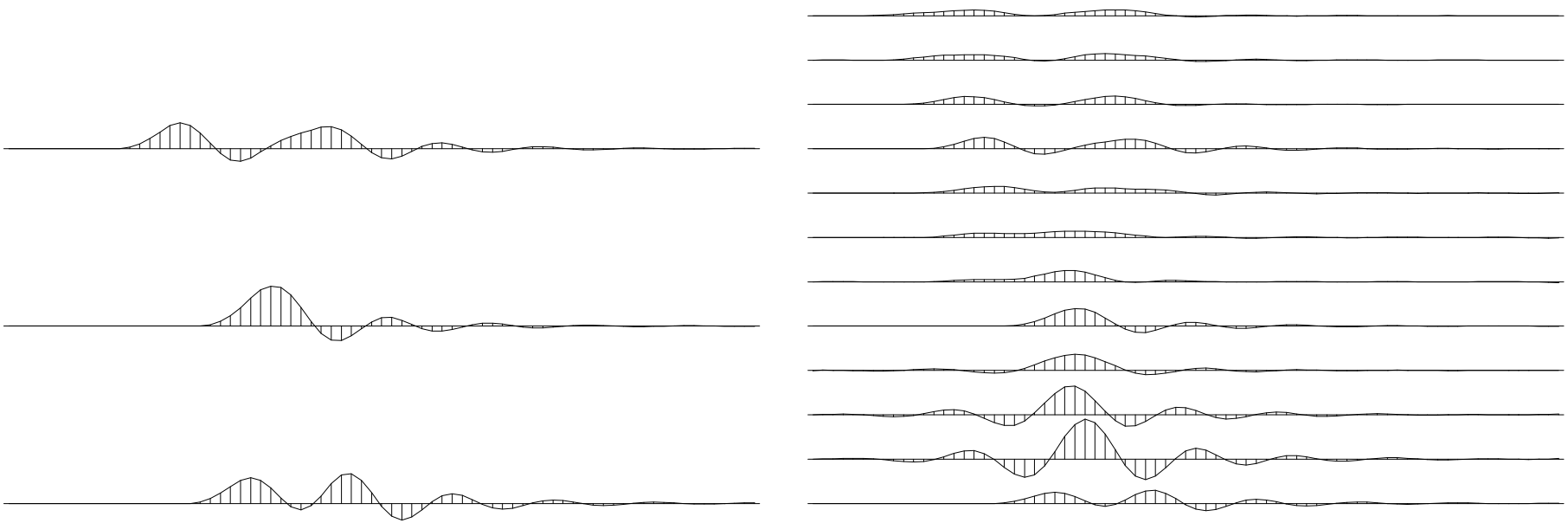


Figure 8.20: Narrow-banded signal (left) with interpolation (right). mis-lacenarrow [ER]

the PE filter really should change to account for the interlacing. Interlacing the PE filter is too simple a model. We can think of interlacing as merely the first guess in a nonlinear problem.

8.5. A FULLY TWO-DIMENSIONAL PE FILTER

The prediction-error filters we dealt with above are not genuinely two-dimensional because Fourier transform over time would leave independent, 1-D, spatial PE filters for each temporal frequency. What is a truly **two-dimensional prediction-error filter**?[1] This is a question we should answer in our quest to understand resonant signals aligned along various dips. Figure 8.11 shows that an interpolation-error filter is no substitute for a PE filter in one dimension. So we need to use special care in properly defining a 2-D PE filter. Recall the basic proof in chapter 7 (page **??**) that the output of a PE filter is white. The basic idea is that the output residual is uncorrelated with the input fitting functions (delayed signals); hence, by linear combination, the *output* is uncorrelated with the *past outputs* (because past outputs

[1] I am indebted to John P. Burg for some of these ideas.

are also linear combinations of past inputs). This is proven for *one* side of the autocorrelation, and the last step in the proof is to note that what is true for one side of the autocorrelation must be true for the other. Therefore, we need to extend the idea of "past" and "future" into the plane to divide the plane into two halves. Thus I generally take a 2-D PE filter to be of the form

$$\begin{array}{ccccccc} a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ a & a & a & a & a & a & a \\ \cdot & \cdot & \cdot & 1 & a & a & a \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{array} \tag{8.12}$$

where "$\cdot$" marks the location of a zero element and a marks the location of an element that is found by minimizing the output power. Notice that for each a, there is a point mirrored across the "1" at the origin, and the mirrored point is not in the filter. Together, all the a locations and their mirrors cover the plane. Obviously the plane can be bisected in other ways, but this way seems a natural one for the processes we have in mind. The **three-dimensional prediction-error filter** which embodies the same concept is shown in Figure 8.21.

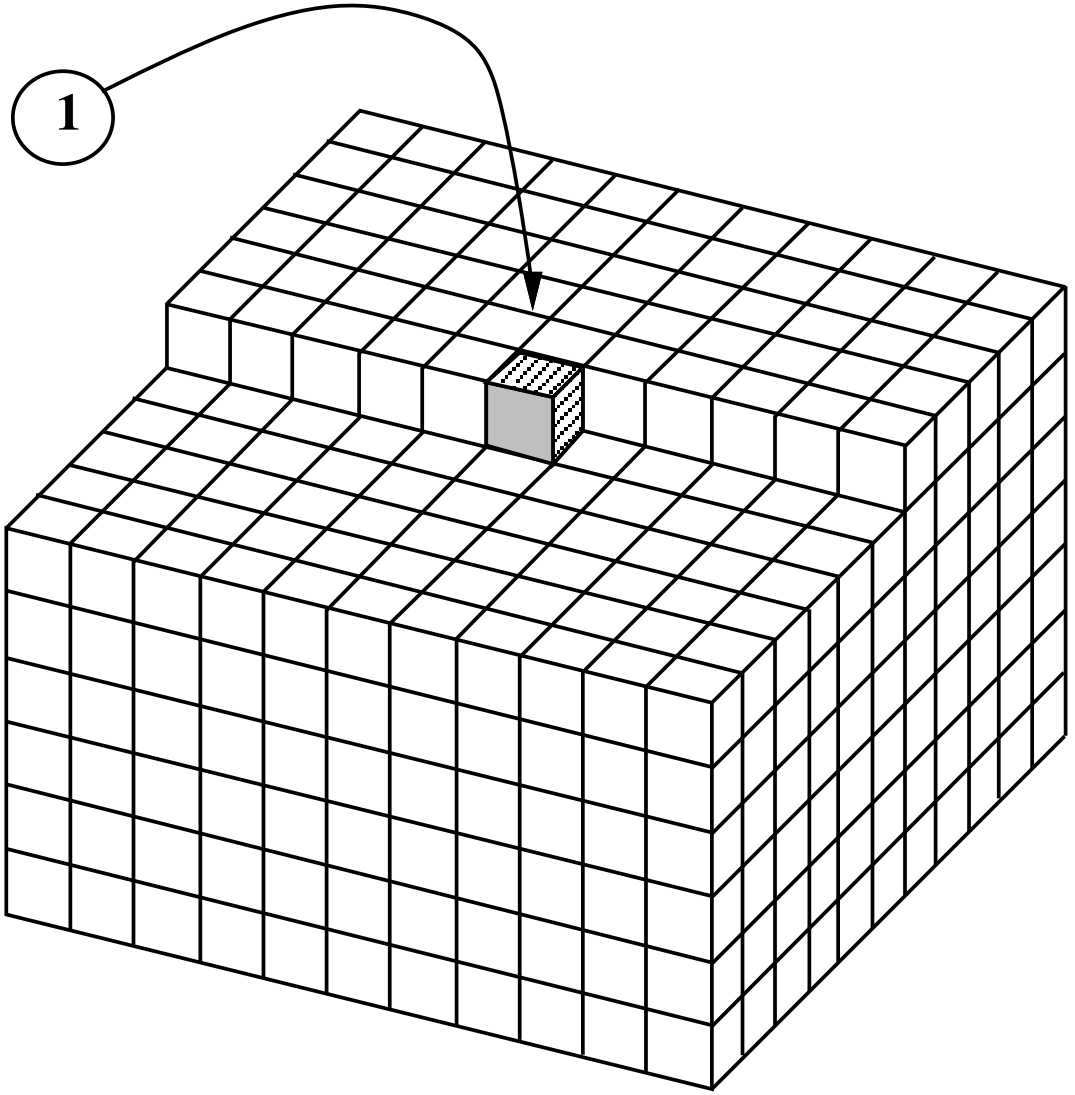


Figure 8.21: Three-dimensional prediction-error filter. mis-3dpef [NR]

Can "short" filters be used? Experience shows that a significant detriment to **white**ning with a PE filter is an underlying model that is not purely a polynomial division because it has a convolutional (moving average) part. The convolutional part is especially troublesome when it involves serious bandlimiting, as does convolution with bionomial coefficients (for example, the Butterworth filter, discussed in chapter 10). When bandlimiting occurs, it seems best to use a **gap**ped PE filter. I have some limited experience with 2-D PE filters that suggests using a gapped form like

$$\begin{array}{ccccccccc} a & a & a & a & a & a & a & a & a \\ a & a & a & a & a & a & a & a & a \\ a & a & a & a & a & a & a & a & a \\ a & a & a & a & a & a & a & a & a \\ \cdot & \cdot & \cdot & 1 & \cdot & \cdot & a & a & a \end{array} \tag{8.13}$$

With this kind of PE filter, the output traces are uncorrelated with each other, and the output plane is correlated with itself only for a short distance (the length of the gap) on the time axis.

EXERCISES:

1 Recall Figure 4.4. Explain how to do the job properly.

8.5.1. The hope method

We have examined the two-stage linear method of missing-data restoration, which calls for solving for a filter, interlacing it, and then solving for the missing data. I believe that method, with its interlacing, is unsuitable for data with a narrow spectral signal-to-noise ratio, such as we often encounter in practice. It would indeed be nice to be able to work with such data.

Recall equation (8.4):

$$\min_{P,A} \; (||PA|| \; + \; \lambda_9||P_0A|| \; + \; \lambda_{10}||PA_0|| \;)$$

Now we hope to solve the trace-interlace problem directly from this optimization. Without the training data P_0 and the high-pass filter A_0, however, the trace-interlace problem is highly **nonlinear**, and, as in the case of the one-dimensional problem, I found I was unable to descend to a satisfactory solution. Therefore, we must think about what the training data and prior filter might be. Our first guess might be that

P_0 is a low-pass dip filter and A_0 is a high-pass dip filter. Several representations for low- and high-pass dip filters are described in IEI. I performed a few tests with them but was not satisfied with the results.

Another possibility is that P_0 should be the solution as found by the interlacing method. Time did not allow me to investigate this promising idea.

Still another possibility is that these problems are so easy to solve (requiring workstation compute times of a few seconds only) that we should abandon traditional optimization methods and use **simulated annealing** (**Rothman**, 1985).

All the above ideas are hopeful. A goal of this study is to define and characterize the kinds of problems that we think should be solvable. A simple example of a dataset that I believe should be amenable to interpolation, even with substantial noise, is shown in Figure 8.22. I have not worked with this case yet.

To prepare the way, and to perform my preliminary (but unsatisfactory) tests, I prepared subroutine `hope()`, the two-dimensional counterpart to `missif()` /prog:missif and `misfip()` /prog:misfip. hope I found the jump-and-interlace 2-D convolution `cinjof()` /prog:cinjof unsuitable here because it does not align its output consistently with the aligning convolution `cinloi()` /prog:cinloi. So I wrote an aligning

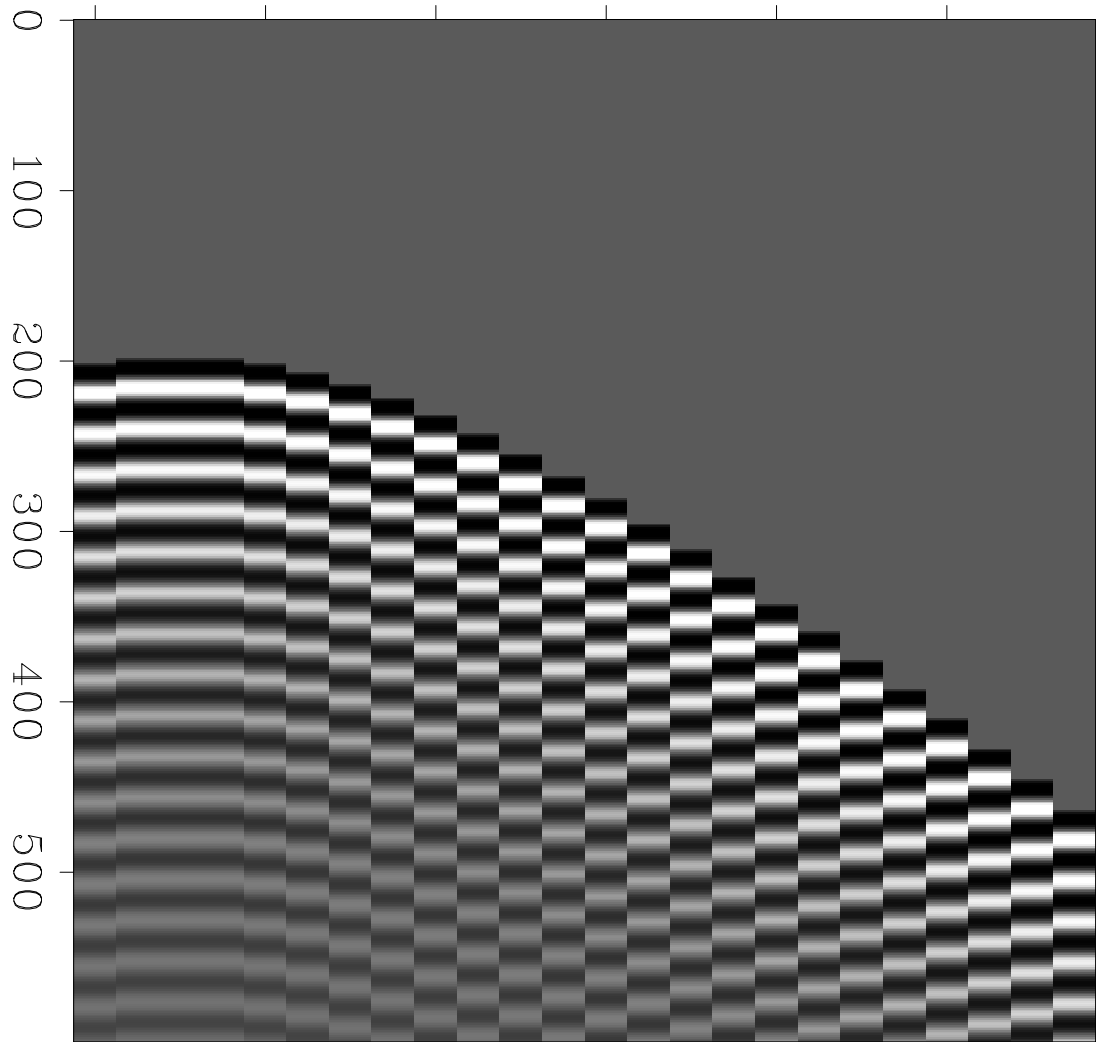


Figure 8.22: Narrow-banded data that skilled humans can readily interpolate. mis-alias [ER]

```
subroutine hope( gap, h1,h2,hh, t1,t2,tt, a1,a2,aa, p1,p2,pp, known, niter)
integer         h1,h2,h12,  t1,t2,t12,  a1,a2,a12,  p1,p2,p12
integer         i, gap, iter, niter, midpt, nx,nr, px,ax,  qr,tr,hr
real            hh(h1,h2), tt(t1,t2), aa(a1,a2), pp(p1*p2), known(p1*p2), dot
temporary real  x( p1*p2 +a1*a2),  rr( p1*p2 +p1*p2 +t1*t2)
temporary real  g( p1*p2 +a1*a2),  gg( p1*p2 +p1*p2 +t1*t2)
temporary real  s( p1*p2 +a1*a2),  ss( p1*p2 +p1*p2 +t1*t2)
p12 = p1*p2;    a12 = a1*a2;    t12 = t1*t2;    h12= h1*h2;
nx  = p12 + a12;          px= 1;        ax= 1+p12
nr  = p12 + p12 + t12;    qr= 1;        hr= 1+p12;      tr= 1+p12+p12
call zero( a12, aa);    midpt= a1/2;    aa( midpt, 1 ) = sqrt( dot( p12,pp,pp))
call zero( nx, x);      call zero( nr, rr);     call copy( p12, pp, x(px))
call zero( nx, g);      call zero( nr, gg);     call copy( a12, aa, x(ax))
do iter= 0, niter {
        call cinloi( 0, 0, midpt,1, a1,a2,aa, p1,p2,pp,    rr(qr))
        call cinloi( 0, 0, midpt,1, h1,h2,hh, p1,p2,pp,    rr(hr))
        call cinloi( 0, 0, midpt,1, a1,a2,aa, t1,t2,tt,    rr(tr))
        call scaleit (                      -1., nr,          rr    )
        call cinloi( 1, 0, midpt,1, a1,a2,aa, p1,p2,g(px), rr(qr))
        call cinlof( 1, 0, midpt,1, p1,p2,pp, a1,a2,g(ax), rr(qr))
        call cinloi( 1, 1, midpt,1, h1,h2,hh, p1,p2,g(px), rr(hr))
        call cinlof( 1, 1, midpt,1, t1,t2,tt, a1,a2,g(ax), rr(tr))
        do i= 1, p12       { if( known(i) != 0.)    g( i + (px-1)) = 0.}
        do i= 1, midpt+gap {                        g( i + (ax-1)) = 0.}
        call cinloi( 0, 0, midpt,1, a1,a2,aa, p1,p2,g(px), gg(qr))
        call cinlof( 0, 1, midpt,1, p1,p2,pp, a1,a2,g(ax), gg(qr))
        call cinloi( 0, 0, midpt,1, h1,h2,hh, p1,p2,g(px), gg(hr))
        call cinlof( 0, 0, midpt,1, t1,t2,tt, a1,a2,g(ax), gg(tr))
        call cgstep( iter, nx,  x, g, s, _
                           nr, rr,gg,ss )
        call copy( p12, x(px), pp)
        call copy( a12, x(ax), aa)
        }
return; end
```

Back

```
# CINLOF --- Convolution INternal with Lags.  Output is adjoint to FILTER.
#
subroutine cinlof( adj, add, lag1,lag2,  n1,n2,xx,  nb1,nb2,bb,  yy)
integer            adj, add, lag1,lag2,  n1,n2,     nb1,nb2
real                                      xx(n1,n2), bb(nb1,nb2), yy(n1,n2)
integer y1,y2, x1,x2, b1, b2
call adjnull(      adj, add,                         bb,nb1*nb2,  yy,n1*n2)
if( adj == 0 )
        do b2=1,nb2 { do y2= 1+nb2-lag2, n2-lag2+1 {  x2= y2 - b2 + lag2
        do b1=1,nb1 { do y1= 1+nb1-lag1, n1-lag1+1 {  x1= y1 - b1 + lag1
                                yy(y1,y2) = yy(y1,y2) + bb(b1,b2) * xx(x1,x2)
                                }} }}
else
        do b2=1,nb2 { do y2= 1+nb2-lag2, n2-lag2+1 {  x2= y2 - b2 + lag2
        do b1=1,nb1 { do y1= 1+nb1-lag1, n1-lag1+1 {  x1= y1 - b1 + lag1
                                bb(b1,b2) = bb(b1,b2) + yy(y1,y2) * xx(x1,x2)
                                }} }}
return; end
```

Back

convolution identical with `cinloi()` except that the *filter* is the adjoint. It is called `cinlof()`. cinlof

8.5.2. An alternative principle for 2-D interpolation

In principle, missing traces can be determined to simplify (ω, k)-space. Consider a wave field P composed of several linear events in (t, x)-space. A **contour** plot of energy in (ω, k)-space would show energy concentrations along lines of various $p = k/\omega$, much like Figure 8.18. Let the energy density be $E = \overline{P}\,P$. Along contours of constant E, we should also see $p = dk/d\omega$. The **gradient vector** $(\partial E/\partial\omega, \partial E/\partial k)$ is perpendicular to the contours. Thus the dot product of the vector (ω, k) with the gradient should vanish. I propose to solve the **regression** that the dot product of the vector (ω, k) with the gradient of the log energy be zero, or, formally,

$$0 \quad \approx \quad \omega \, \frac{\Re\, \overline{P}\, \frac{\partial}{\partial \omega} P}{\overline{P}\, P} + k \, \frac{\Re\, \overline{P}\, \frac{\partial}{\partial k} P}{\overline{P}\, P} \tag{8.14}$$

The variables in the regression are the values of the missing traces. Obviously, the numerator and the denominator should be smoothed in small windows in the

(ω, k)-plane. This makes conceptual sense but does not fit well with the idea of small windows in (t, x)-space. It should be good for some interesting discussions, though. For example, in Figure 8.18, what will happen where event lines cross? Is this formulation adequate there? Also, how should the Nyquist limitation on total bandwidth be expressed?

8.6. TOMOGRAPHY AND OTHER APPLICATIONS

Medical tomography avoids a problem that is unavoidable in earth-science **tomography**. In medicine it is not difficult to surround the target with senders and receivers. In earth science it is nearly impossible. It is well known that our reconstructions tend to be **indeterminate** along the dominant ray direction. Customarily, the indeterminacy is resolved by minimizing power in a roughened image. The roughening filter should be inverse in spectrum to the desired image spectrum. Unfortunately, that spectrum is unknown and arbitrary. Perhaps we can replace this arbitrary image smoothing by something more reasonable in the space of the miss-

ing data.

Recall the well-to-well tomography problem in chapter 5. Given a sender at depth z_s in one well, a receiver at depth z_g in the other well, and given traveltimes $t_k(z_s, z_g)$, the rays are predominantly horizontal. Theory says we need some rays around the vertical. Imagine the two vertical axes of the wells being supplemented by two horizontal axes, one connecting the tops of the wells and one connecting the bottoms, with missing data traveltimes $t_m(x_s, x_g)$. From any earth model, t_k and t_m are predicted. But what principles can give us t_m from t_k? Obviously something like we used in Figures 8.2–8.6. Data for the tomographic problem is two-dimensional, however: let the source location be measured as the distance along the perimeter of a box, where the two sides of the box are the two wells. Likewise, receivers may be placed along the perimeter. Analogous to the *midpoint* and *offset* axes of surface seismology (see IEI), we have midpoint and offset along the perimeter. Obviously there are discontinuities at the corners of the box, and everything is not as regular as in medical imaging, where sources and receivers are on a circle and their positions measured by angles. The box gives us a plane in which to lay out the data, not just the recorded data, but all the data that we think is required to represent the image. To fill in the missing data we can minimize the power out of some two-dimensional

filter, say, for example, the Laplacian filter $\partial_s^2 + \partial_g^2$. This would give us the two-dimensional equivalent of Figures 8.2–8.6.

Alas, this procedure cannot produce information where none was recorded. But it should yield an image that is not overwhelmed by the obvious heterogeneity of the data-collection geometry.

The traditional approach of "**geophysical inverse theory**" requires the inverse of the model **covariance matrix**. How is this to be found using our procedure? How are we to cope with the absence of rays in certain directions? Notice that whatever the **covariance matrix** may be, the resolution is very different in different parts of the model: it is better near the wells, best halfway down near a well, and worst halfway between the wells, especially near the top and bottom. How can this information be quantified in the model's inverse **covariance matrix**? This is a hard question, harder than the problem that we would solve if we were *given* the matrix. Most people simply give up and let the inverse covariance be a roughening operator like a Laplacian, constant over space.

With the filling of data space, will it still be necessary to smooth the model explicitly (by minimizing energy in a roughened model)? Mathematically, the question is one of the "completeness" of the data space. I believe there are analytic

solutions well known in medical imaging that prove that a circle of data is enough information to specify completely the image. Thus, we can expect that little or no arbitrary image smoothing is required to resolve the indeterminacy—it should be resolved by the assertion that statistics gathered from the known data are applicable to the missing data.

I suggest, therefore, that every data space be augmented until it has the dimensionality and completeness required to determine a solution. If this cannot be done fully, it should still be done to the fullest extent feasible.

The **covariance matrix** of the **residual** in data space (missing and observed) seems a reasonable thing to estimate—easier than the covariance matrix of the model. I think the model covariance matrix should not be thought of as a covariance matrix of the solution, but as a chosen interpolation function for plotting the solution.

8.6.1. Clash in philosophies

One philosophy of geophysical data analysis called "**inverse theory**" says that missing data is irrelevant. According to this philosophy, a good geophysical model only

needs to fit the real data, not interpolated or extrapolated data, so why bother with interpolated or extrapolated data? Even some experienced practitioners belong to this school of thought. My old friend Boris Zavalishin says, “Do not trust the data you have not paid for.”

I can justify data interpolation in both human and mathematical terms. In human terms, the solution to a problem often follows from the adjoint operator, where the data space has enough known values. With a good display of data space, people often apply the adjoint operator in their minds. Filling the data space prevents distraction and confusion. The mathematical justification is that inversion methods are notorious for slow convergence. Consider that matrix-inversion costs are proportional to the cube of the number of unknowns. Computers balk when the number of unknowns goes above one thousand; and our images generally have millions. By extending the operator (which relates the model to the data) to include missing data, we can hope for a far more rapid convergence to the solution. On the extended data, perhaps the adjoint alone will be enough. Finally, we are not falsely influenced by the “data not paid for” if we adjust it so that there is no residual between it and the final model.

8.6.2. An aside on theory-of-constraint equations

A theory exists for general **constraint**s in **quadratic form** minimization. I have not found the theory to be useful in any application I have run into so far, but it should come in handy for writing erudite theoretical articles.

Constraint equations are an underdetermined set of equations, say $\mathbf{d} = \mathbf{G}\mathbf{x}$ (the number of components in $\mathbf{x}$ exceeds that in $\mathbf{d}$), which must be solved exactly while some other set is solved in the **least-squares** sense, say $\mathbf{y} \approx \mathbf{B}\mathbf{x}$. This is formalized as

$$\min_{\mathbf{x}} \{Q_C(\mathbf{x}) = \lim_{\epsilon \to 0} [(\mathbf{y}-\mathbf{B}\mathbf{x})'(\mathbf{y}-\mathbf{B}\mathbf{x}) + \frac{1}{\epsilon}(\mathbf{d}-\mathbf{G}\mathbf{x})'(\mathbf{d}-\mathbf{G}\mathbf{x})]\} \tag{8.15}$$

In my first book (FGDP: see page 113), I minimized Q_C by power series, letting $\mathbf{x} = \mathbf{x}^{(0)} + \epsilon\,\mathbf{x}^{(1)}$, and hence $Q_C = Q^{(0)} + \epsilon\, Q^{(1)} + \cdots$. I minimized both $Q^{(0)}$ and $Q^{(1)}$ with respect to $\mathbf{x}^{(0)}$ and $\mathbf{x}^{(1)}$. After a page of algebra, this approach leads to the system of equations

$$\begin{bmatrix} \mathbf{B}'\mathbf{B} & \mathbf{G}' \\ \mathbf{G} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \lambda \end{bmatrix} = \begin{bmatrix} \mathbf{B}'\mathbf{y} \\ \mathbf{d} \end{bmatrix} \tag{8.16}$$

where $\mathbf{x}^{(1)}$ has been superseded by the variable $\lambda = \mathbf{G}\mathbf{x}^{(1)}$, which has fewer components than $\mathbf{x}^{(1)}$, and where $\mathbf{x}^{(0)}$ has simply been replaced by $\mathbf{x}$. The second of the two equations shows that the constraints are satisfied. But it is not obvious from equation (8.16) that (8.15) is minimized.

The great mathematician Lagrange apparently looked at the result, equation (8.16), and realized that he could arrive at it far more simply by extremalizing the following quadratic form:

$$Q_L(\mathbf{x},\lambda) \quad = \quad (\mathbf{y}-\mathbf{B}\mathbf{x})'(\mathbf{y}-\mathbf{B}\mathbf{x})+(\mathbf{d}-\mathbf{G}\mathbf{x})'\lambda+\lambda'(\mathbf{d}-\mathbf{G}\mathbf{x}) \qquad (8.17)$$

We can quickly verify that Lagrange was correct by setting to zero the derivatives with respect to $\mathbf{x}'$ and λ'. Naturally, everyone prefers to handle constraints by Lagrange's method. Unfortunately, Lagrange failed to pass on to the teachers of this world an intuitive reason *why* extremalizing (8.17) gives the same result as extremalizing (8.15). Lagrange's quadratic form is not even positive definite (that is, it cannot be written as something times its adjoint). In honor of Lagrange, the variables λ have come to be known as **Lagrange multiplier**s.

8.7. References

Canales, L.L., 1984, Random noise reduction: 54th Ann. Internat. Mtg., Soc. Explor. Geophys., Expanded Abstracts, 525-527.

Rothman, D., 1985, Nonlinear inversion, statistical mechanics, and residual statics estimation: Geophysics, **50**, 2784-2798

Spitz, S., 1991, Seismic trace interpolation in the F-X domain: Geophysics, **56**, 785-794.

Chapter 9

Hyperbola tricks

In exploration seismology much attention is given to all aspects of hyperbolas. My previous book (IEI) is filled with **hyperbola** lore, especially wave-equation solution methodology. That book, however, only touches questions of hyperbolas arising in

least-squares problems. I wish I could say this chapter organizes everything better, but in reality it is a miscellaneous collection of additional material in which hyperbolas are exploited with due regard to operator conjugacy and least squares.

9.1. PIXEL-PRECISE VELOCITY SCANNING

Traditionally, velocity scanning is done by the loop structure given in chapter 5, in which the concept of a velocity transform was introduced. This structure is

```
do v
    do tau
        do x
            t = sqrt( tau**2 + (x/v)**2 )
            velo( tau, v) = velo( tau, v)  +  data( t, x)
```

These loops transform source-receiver offset x to velocity v in much the same way as Fourier analysis transforms time to frequency. Here we will investigate a new alternative that gives conceptually the same result but differs in practical ways. It is to transform with the following loop structure:

```
do tau
    do t = tau, tmax
        do x
            v = sqrt( x**2  / ( t**2 - tau**2 ))
            velo( tau, v) = velo( tau, v)  +  data( t, x)
```

Notice that $t = \sqrt{\tau^2 + (x/v)^2}$ in the conventional code is algebraically equivalent to $v = x/\sqrt{t^2 - \tau^2}$ in the new code. The traditional method finds one value for each point in *output* space, whereas the new method uses each point of the *input* space exactly once.

The new method, which I have chosen to call the "**pixel-precise** method," differs from the traditional one in cost, smoothing, accuracy, and truncation. The cost of traditional velocity scanning is proportional to the product $N_t N_x N_v$ of the lengths of the axes of time, offset, and velocity. The cost of the new method is proportional to the product $N_t^2 N_x/2$. Normally $N_t/2 > N_v$, so the new method is somewhat more costly than the traditional one, but not immensely so, and in return we can have all the (numerical) resolution we wish in velocity space at no extra cost. The verdict is not in yet on whether the new method is better than the old one in routine practice, but the reasoning behind the new method teaches many lessons. Not ex-

amined here is the smooth envelope (page **??**) that is a postprocess to conventional velocity scanning.

Certain facts about aliasing must be borne in mind as one defines any velocity scan. A first concern arises because typical hyperbolas crossing a typical **mesh** encounter multiple points on the time axis for each point on the space axis. This is shown in Figure 9.1. An aliasing problem will be experienced by any program that selects only one signal value for each x instead of the multiple points that are shown. The extra boxes complicate traditional velocity scanning. Many programs ignore it without embarrassment only because low-velocity events contain only shallow information about the earth. (A cynical view is that field operations tend to oversample in offset space because of this limitation in some velocity programs.) A significant improvement is made by summing all the points in boxes. A still more elaborate analysis (which we will not pursue here) is to lay down a hyperbola on a mesh and interpolate a line integral from the traces on either side of the line.

A second concern arises from the sampling in velocity space. Traditionally people question whether to sample velocity uniformly in velocity, slowness, or slowness squared. Difficulty arises first on the widest-offset trace. When jumping from one velocity to the next, the time on the wide-offset trace should not jump so far that it

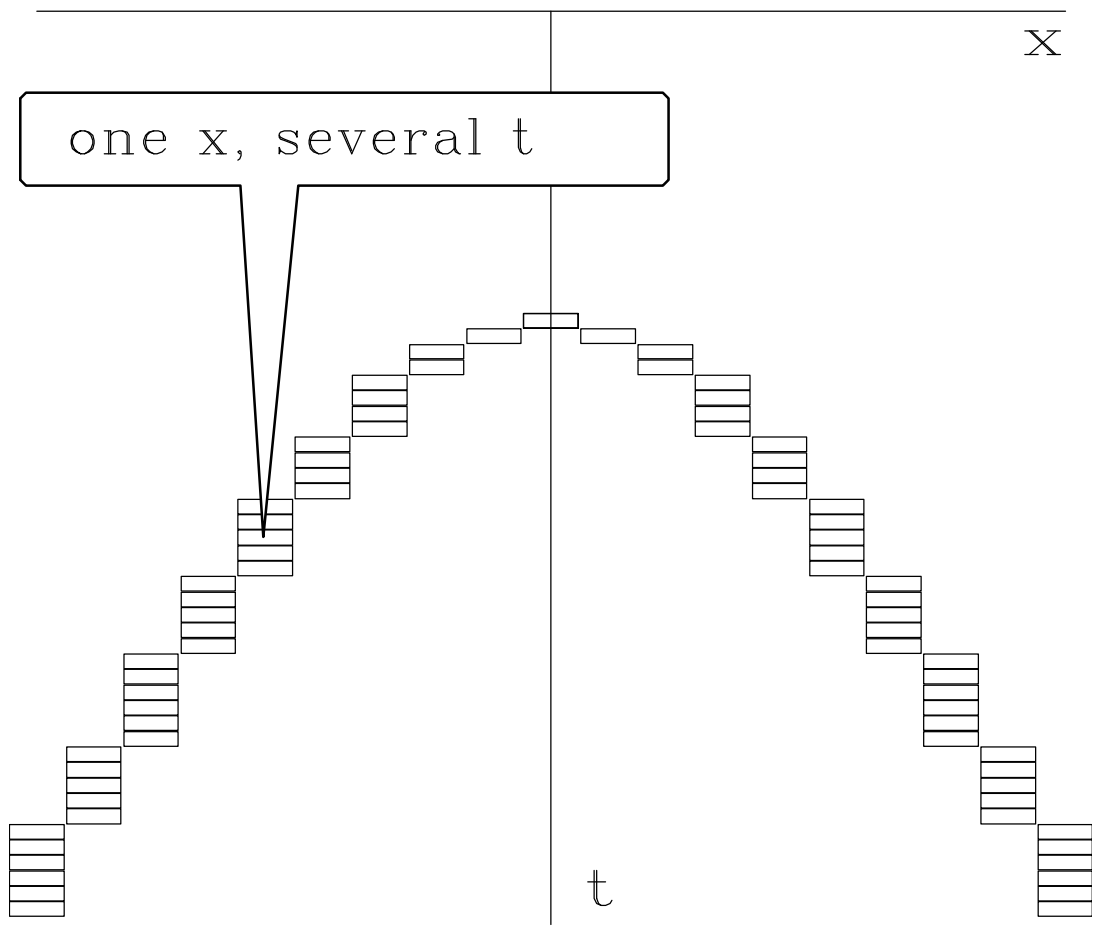


Figure 9.1: A typical hyperbola crossing a typical mesh. Notice that the curve is represented by multiple time points for each x. hyp-lineint [NR]

leaves a gap, as shown in Figure 9.2.

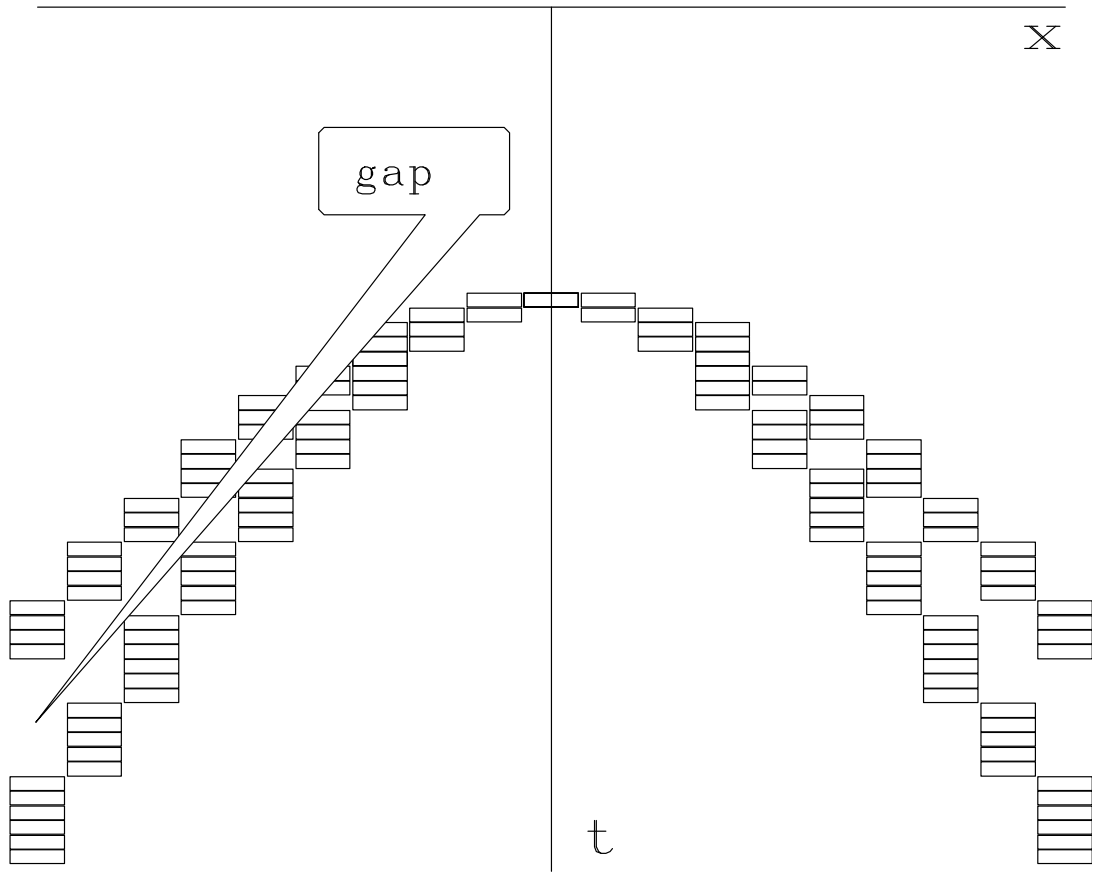


Figure 9.2: Too large an interval in velocity will leave a gap between the hyperbolic scans. hyp-deltavel [NR]

With the new method there is no chance of missing a point on the wide-offset trace. For each depth τ, every point below τ in the input-data space (including the

wide-offset trace) is summed exactly once into velocity space (whether that space is discretized uniformly in velocity or slowness). Also, the inner trace enters *only* once.

The new method also makes many old interpolation issues irrelevant. New questions arise, however. The (t,x)-position of the input data is exact, as is τ. Interpolation becomes a question only on v. Since velocity scanning in this way is independent of the number of points in velocity, we could sample densely and use nearest-neighbor interpolation (or any other form of interpolation). A disadvantage is that some points in (τ,v)-space may happen to get *no* input data, especially if we refine v too much.

The result of the new velocity transformation is shown in Figure 9.3. The figure includes some scaling that will be described later. The code that generated Figure 9.3 is just like the pseudocode above except that it parameterizes velocity in uniform samples of inverse velocity squared, $s = v^{-2}$. A small advantage of using s-space instead of v-space is that the trajectories we see in (τ,s)-space are readily recognized as parabolas, namely $\tau^2 = t^2 - x^2 s$, where each parabola comes from a particular point in (t,x).

To exhibit all the artifacts as clearly as possible, I changed all signal values to

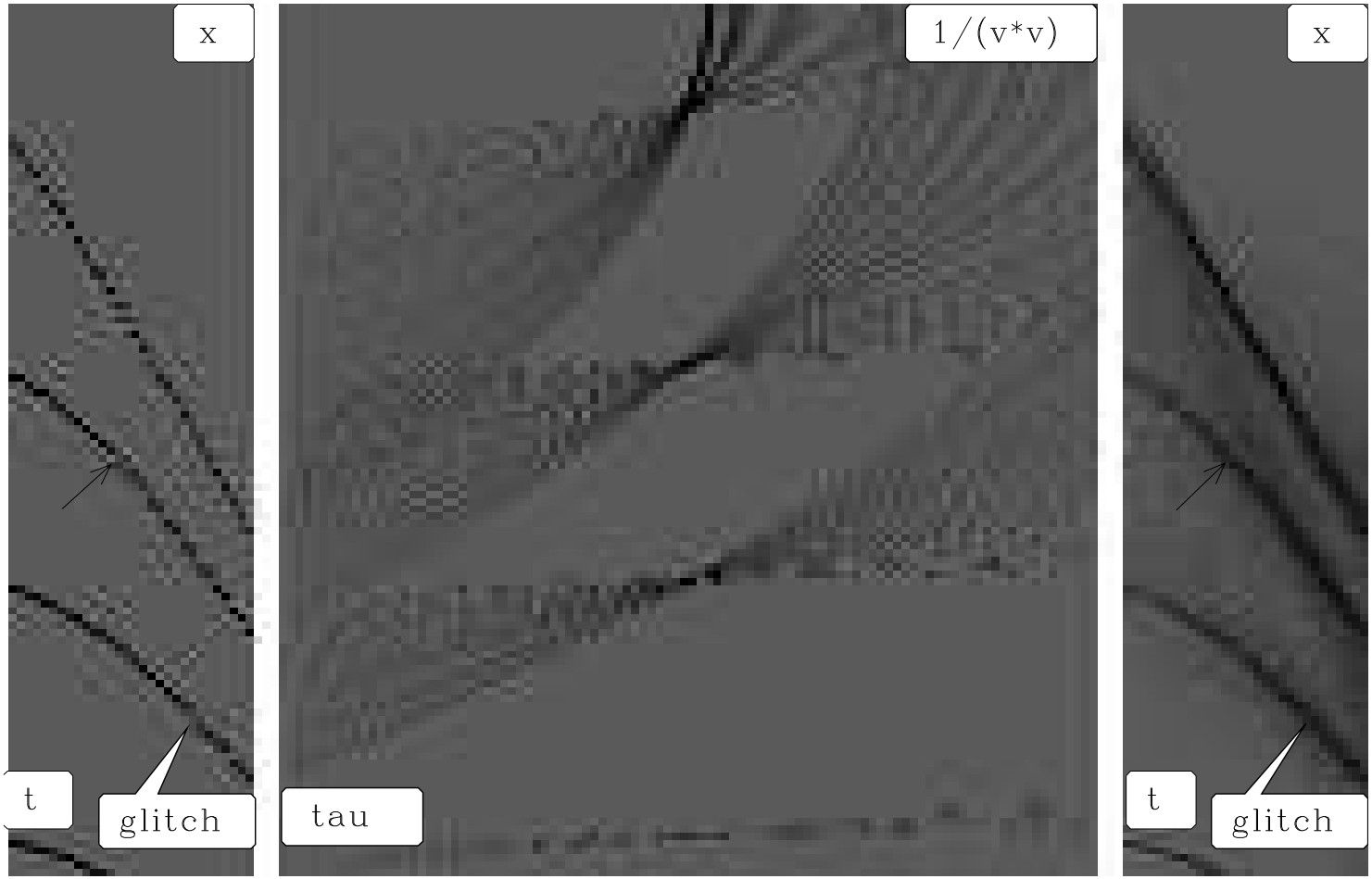


Figure 9.3: Offset to slowness squared and back to offset. hyp-vspray1 [NR]

their signed square roots before plotting brightness. This has the effect of making the plots look noisier than they really are. I also chose Δt to be unrealistically large to enable you to see each point. The synthetic input data was made with nearest-neighbor NMO. Notice that resulting timing irregularities in the input are also present in the reconstruction. This shows a remarkable precision.

Balancing the pleasing result of Figure 9.3 is the poor result from the same program shown in Figure 9.4. The new figure shows that points in velocity space map to bits of hyperbolas in offset space—not to entire hyperbolas. It also shows that *small*-offset points become *sparsely* dotted lines in velocity space.

The problem of hyperbolas being present only discontinuously is solvable by smearing over any axis, t, x, τ, or v, but we would prefer intelligent smoothing over the appropriate axis.

9.1.1. Smoothing in velocity

To get smoother results I took the time axis to be continuous and the signal value at (t,x) to be distributed between the two points $t_- = t - \Delta t/2$ and $t_+ = t + \Delta t/2$. The two time points $t_\pm$ and the x-value are mapped to two slownesses $s_\pm$. The signal

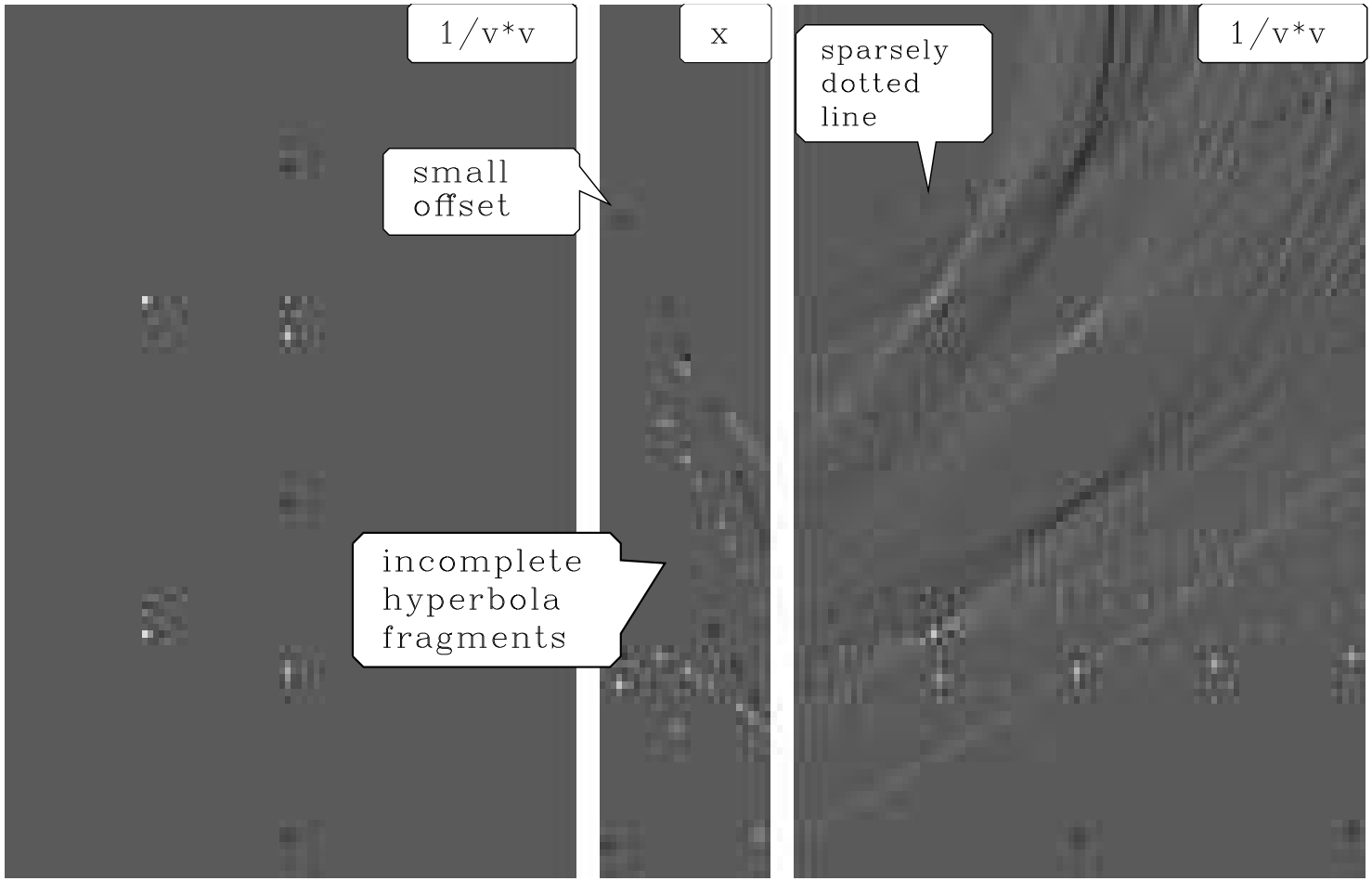


Figure 9.4: Slowness squared to offset and back to slowness squared. hyp-vspray2 [NR]

```
subroutine vspray( adj,  nt,dt,t0,  nx,dx,x0, tx,  ns,ds,s0, zs)
integer adj, it, nt, iz, nz, ix, nx, is, ns, isp, ism
real    tx(nt,nx), zs(nt,ns), scale
real    z,dz,z0, t,dt,t0, x,dx,x0, s,ds,s0, sm,sp, xm,xp, tm,tp
nz=nt;  dz=dt;  z0=t0;
call adjnull( adj, 0,              tx, nt*nx,     zs, nz*ns)
if( adj == 0)  { do ix=1,nx;  call halfdif ( 1, nt, tx(1,ix), tx(1,ix) )}
do iz=  1, nz {  z = z0 + dz*(iz-1)
do ix=  1, nx {  x = x0 + dx*(ix-1)
do it= iz, nt {  t = t0 + dt*(it-1)
        tm = t-dt/2;    xm = x
        tp = t+dt/2;    xp = x
        sm = (tm**2 -z**2)/xp**2;  ism = 1.5+(sm-s0)/ds
        sp = (tp**2 -z**2)/xm**2;  isp = 1.5+(sp-s0)/ds
        if( ism<2 ) next
        if( isp>ns) next
        scale = sqrt( t / (1.+isp-ism) ) / ( abs(x) + abs(dx)/2.)
        do is= ism, isp {
                if( adj == 0)
                        zs(iz ,is) = zs(iz ,is) + tx(it ,ix) * scale
                else
                        tx(it ,ix) = tx(it ,ix) + zs(iz ,is) * scale
                }
        } } }
if( adj != 0)  { do ix=1,nx;  call halfdif ( 0, nt, tx(1,ix), tx(1,ix) )}
return; end
```

Back

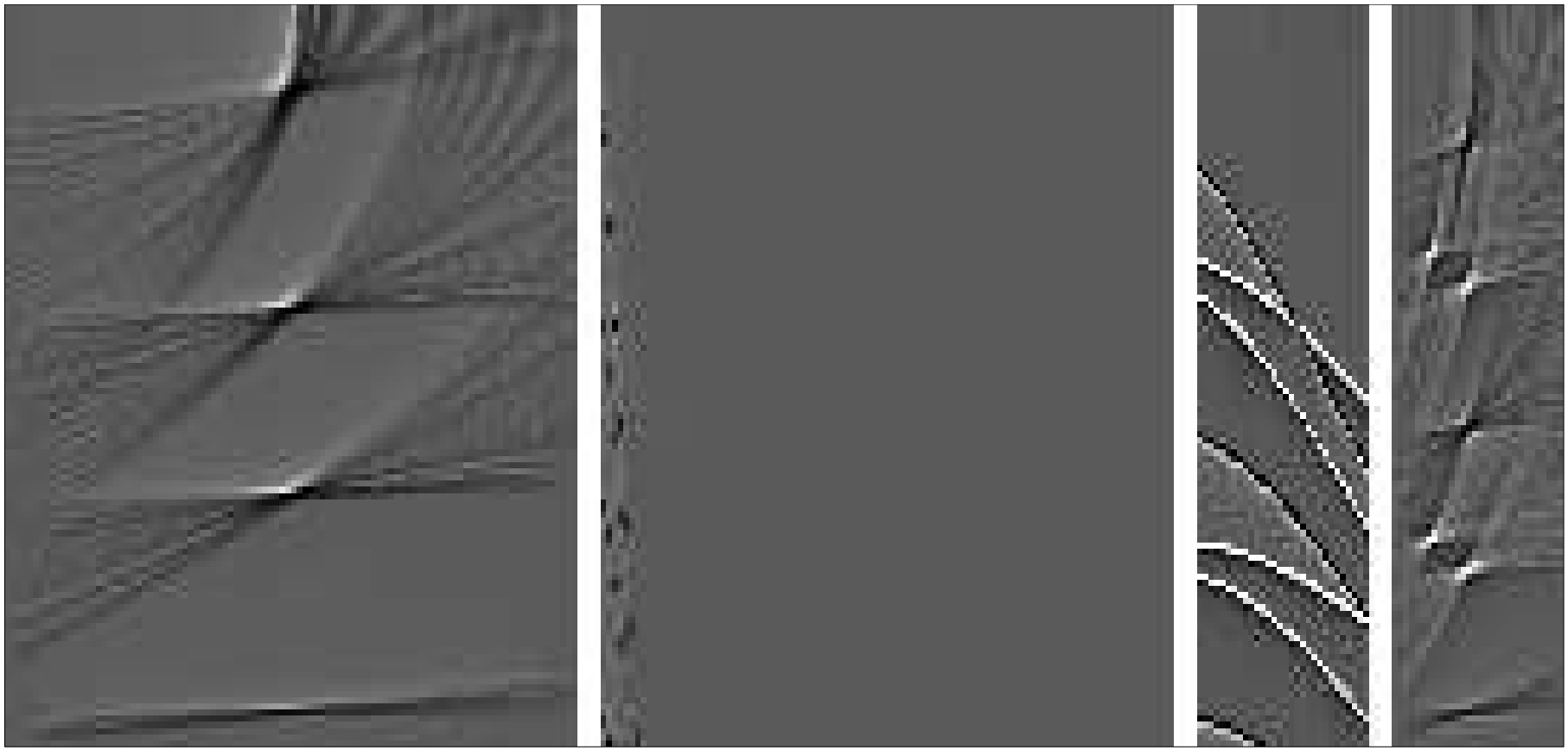


Figure 9.5: Horizontal line method. Compare the left to Figure 9.3 and the right to 9.4. hyp-vspray4 [ER]

from the (t,x)-pixel is sprayed into the horizontal line $(\tau, s_\pm)$. To enable you to reproduce the result, I include the `vspray()` subroutine. vspray Figure 9.5 shows the result for the same inputs as used in Figures 9.3 and 9.4.

9.1.2. Rho filter

Notice the dark **halo** around the reconstruction in Figure 9.3. It was suppressed in Figure 9.5 by the subroutine `halfdifa()`. Recall that slant-stack inversion (see IEI for an example) requires an $|\omega|$ filter. Without doing any formal analysis I guessed that the same filter would be helpful here because the dark halo has a strong spectral component at $\omega = 0$ which would be extinguished by an $|\omega|$ filter. The $|\omega|$ filter is sometimes called a "**rho filter**." Because of the close relation of slant-stack inversion to wave propagation and **causality**, I found it appealing to factor $|\omega|$ into a causal $\sqrt{-i\omega}$ part and an anticausal $\sqrt{i\omega}$ part. I applied a causal $\sqrt{-i\omega}$ after generating the (t,x)-space and an anticausal $\sqrt{i\omega}$ before making the (τ, v^{-2})-space. I implemented the causality by taking the square root of a Fourier domain representation of causal **differentiation**, namely, $\sqrt{1-Z}$. I show this in subroutine `halfdifa()`. halfdifa

```
# Half order causal derivative.  OK to equiv(xx,yy)
#
subroutine halfdifa( adj, add, n, xx,    yy  )
integer n2, i,       adj, add, n
real omega,                       xx(n), yy(n)
complex cz, cv(4096)
n2=1; while(n2<n) n2=2*n2;          if( n2 > 4096) call erexit('halfdif memory')
do i= 1, n2 {             cv(i) = 0.}
do i= 1, n
        if( adj == 0)   { cv(i) = xx(i)}
        else            { cv(i) = yy(i)}
call adjnull(       adj, add,    xx,n,  yy,n)
call ftu( +1., n2, cv)
                        do i= 1, n2 {
                                omega = (i-1.) * 2.*3.14159265 / n2
                                cz = csqrt( 1. - cexp( cmplx( 0., omega)))
                                if( adj != 0)   cz = conjg( cz)
                                cv(i) = cv(i) * cz
                                }
call ftu( -1., n2, cv)
do i= 1, n
        if( adj == 0)   { yy(i) = yy(i) + cv(i)}
        else            { xx(i) = xx(i) + cv(i)}
return; end
```

Back

Notice also that `vspray()` includes a scaling variable named `scale`. I have not developed a theory for this scale factor, but if you omit it, amplitudes in the reconstructions will be far out of amplitude balance with the input.

9.2. GEOMETRY-BASED DECON

In chapter 7 **deconvolution** was considered to be a one-dimensional problem. We ignored spatial issues. The one-dimensional approach seems valid for waves from a source and to a receiver in the same location, but an obvious correction is required for shot-to-receiver *spatial offset*. A first approach is to apply normal-moveout correction to the data before deconvolution. Previous figures have applied a t^2 amplitude correction to the deconvolution *input*. (Simple theory suggests that the amplitude correction should be t, not t^2, but experimental work, summarized along with more complicated theory in IEI, suggests t^2.) Looking back to Figure **??**, we see that the quality of the deconvolution deteriorated with offset. To test the idea that deconvolution would work better after normal moveout, I prepared Figure 9.6. Looking in the region of Figure 9.6 outlined by a rectangle, we can conclude that NMO should be done before deconvolution. The trouble with this conclusion is that

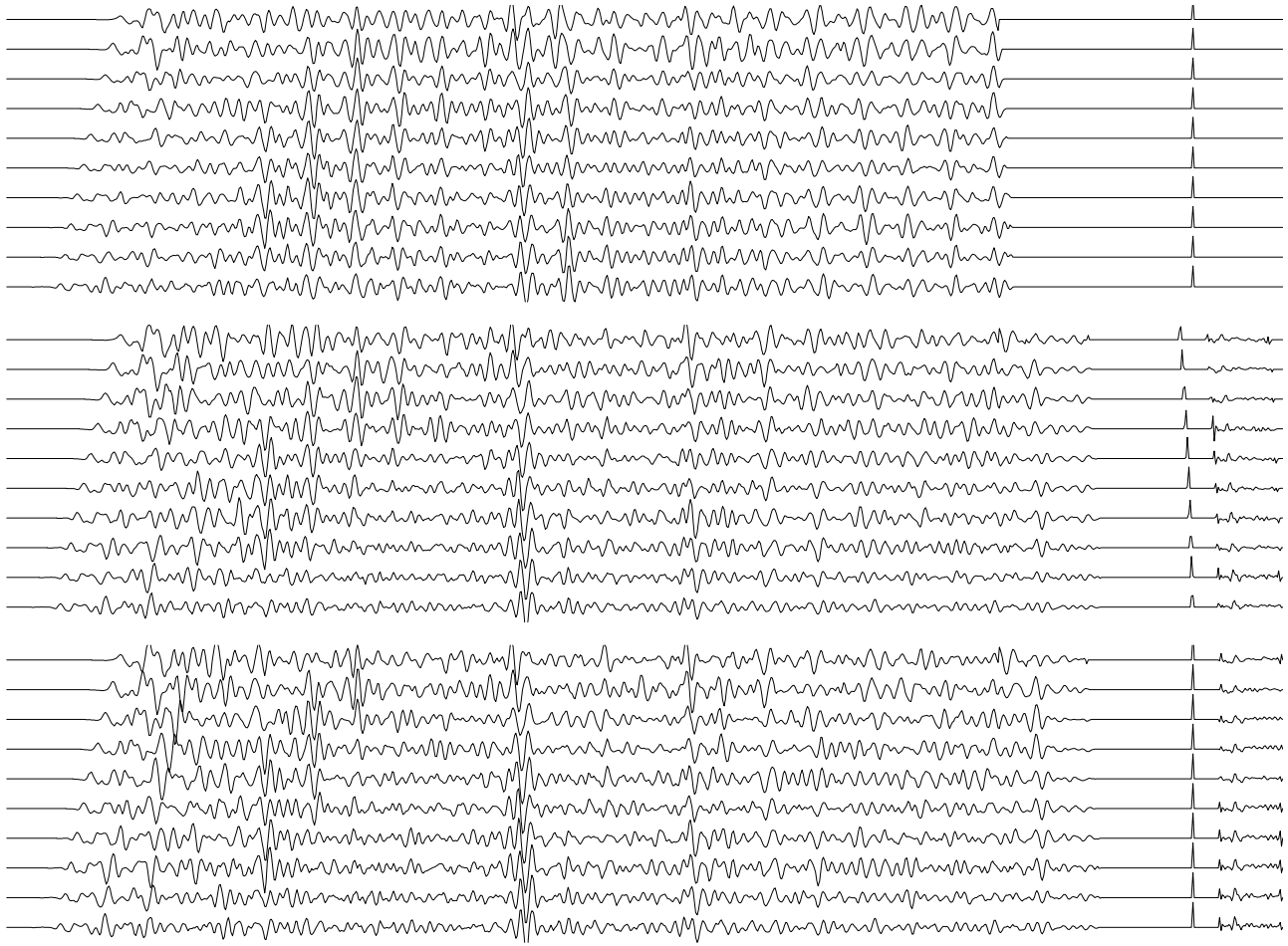


Figure 9.6: Data from Yilmaz and Cumro dataset 27 after t^2 gain illustrates deconvolution working better after NMO. hyp-wz27nmo [NR]

data comes in many flavors. On the wider offsets of any data (such as Figure **??**), it can be seen that NMO destroys the wavelet. A source of confusion is that the convolutional model can occur in two different forms from two separate physical causes, as we will see next.

9.2.1. A model with both signature and reverberation

Convolution occurs in data modeling both before and after moveout correction. Two different deconvolution processes that deal with the two ways convolution occurs are called “**designature**" and “**dereverberation**."

- **Reverberation**

Reverberation is the **multiple** bouncing of waves between layers. Waves at vertical incidence in a water layer over the earth can develop clear, predictable, periodic echos. FGDP gives a detailed theory for this. At nonzero shot-to-geophone offset, the perfect periodicity is destroyed, i.e., multiple reflections no longer have a uniform reverberation period. In a model earth with velocity constant in depth, normal-moveout correction restores the uniform reverberation period. Mathematical tech-

niques for dealing with reverberation in the presence of depth-variable velocity are described in considerable detail in IEI.

- **Signature**

Seismic “signature" is defined to be a convolutional filtering on impulse-source data. This convolution models the nonimpulsive nature of real sources. Imagine the oscillation of a marine **airgun**’s bubble. On land, the earth’s near surface can have a very slow velocity. There Snell’s law will bend all rays to very near vertical incidence. Mathematically, such reverberations in such layers are indistinguishable from source signature. For example, in California the near-surface **soil**s often have a velocity near the air velocity (340 m/s) that grades toward the water velocity (1500 m/s). A buried shot typically has a free-surface reflection ghost whose time delay is virtually independent of angle. Thus the ghost is more like signature than multiple.

Synthetic data in Figure 9.7 shows the result of convolution before and after NMO. An event labeled “G" marks the tail-end of the source signature. The main idea illustrated by the figure is that some events are equally spaced *before* NMO, while other events are equally spaced *after* NMO. We will see that proper deconvolution requires a delicious mixture of NMO and deconvolution principles.

Figure 9.7: Example of convolution before and after NMO. The raw data shows a uniform primary-to-tail interval, while the NMO'ed data shows uniform multiple reverberation. The letters *F*, *G*, and *V* are adjustable parameters in the interactive program controlling water depth, signature tail, and velocity. hyp-deep [NR]

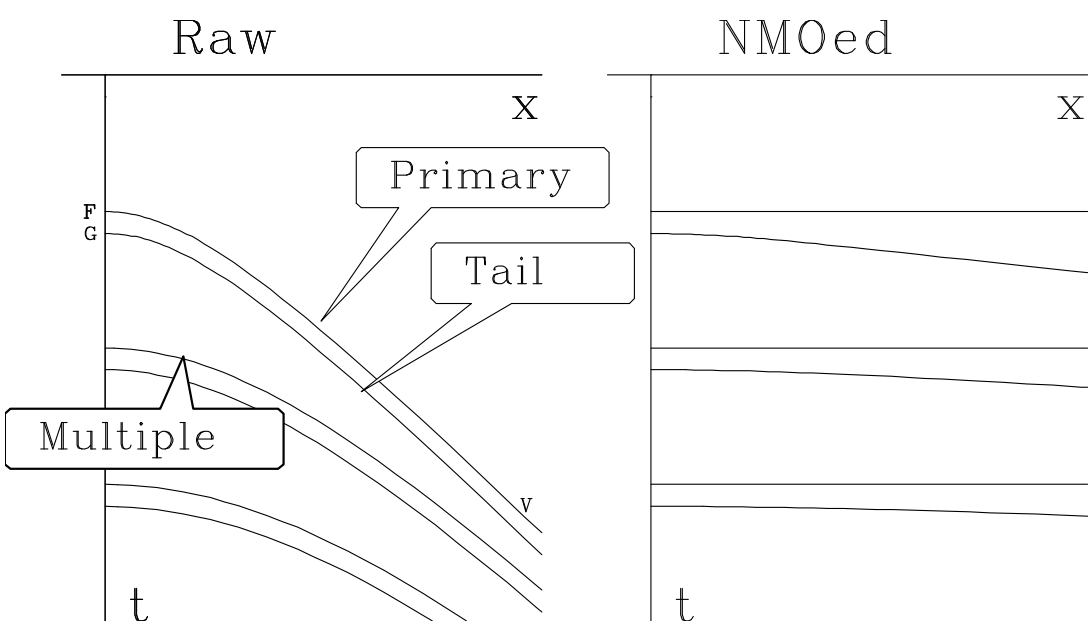


Figure 9.7 happens to have a short time constant with the signature and a longer one with the reverberation. The time constants would be reversed in water shallow compared with the gun's quieting time. This is shown in Figure 9.8. This figure

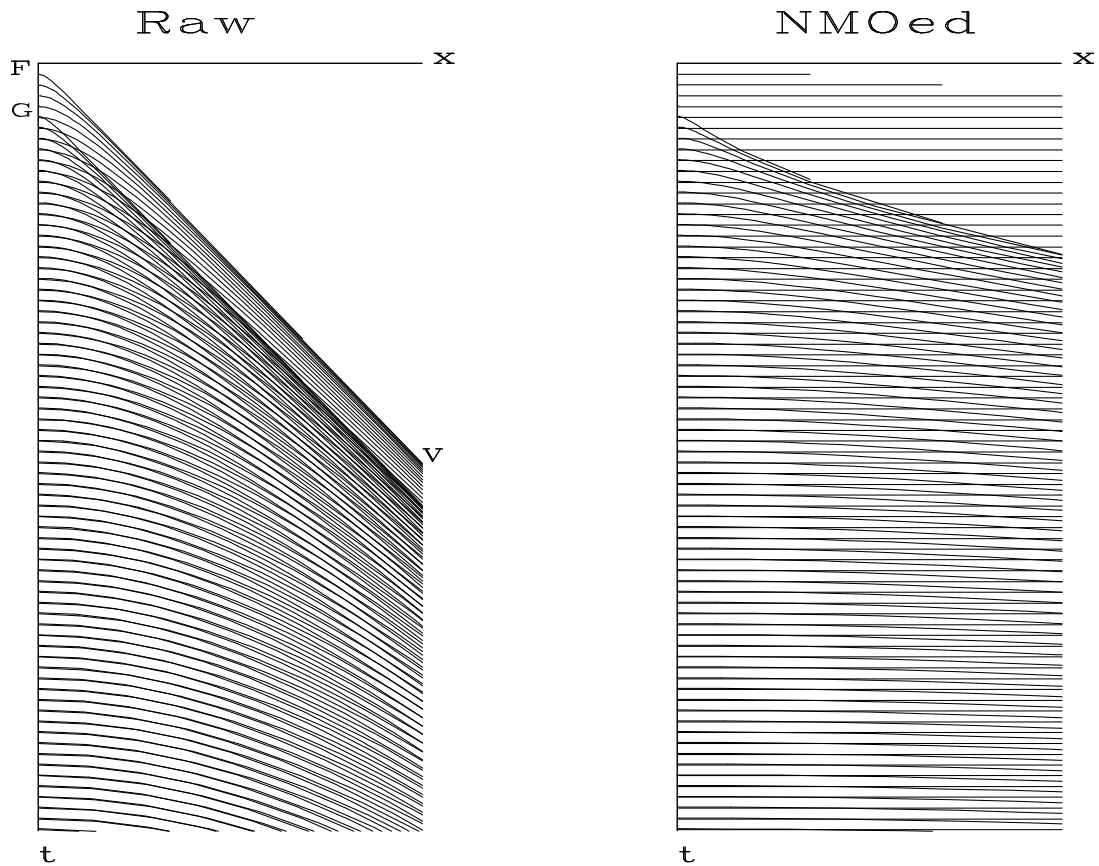


Figure 9.8: Model in water shallow compared to gun quieting time. hyp-shallow [NR]

shows an interesting interference pattern that could also show up in amplitude versus offset studies.

9.2.2. Regressing simultaneously before and after NMO

Before launching into a complicated theory for suppressing both reverberation and signature, let us make some guesses. Let $\mathbf{d}$ denote an original data panel like the left sides of Figure 9.7 and 9.8, and let $\mathbf{d}$ be moved out like the right sides of those figures. If we had only *signature* to contend with, we might formulate the problem as $\mathbf{d} \approx \sum_i \alpha_i \mathbf{x}_i$, where the $\mathbf{x}_i$ are delayed versions of the data, containing $\mathbf{d}(t-i)$, and where the α_i are the scaling coefficients to be found. If we had only *reverberation* to contend with, we might formulate the problem as $\mathbf{d} \approx \sum_i \bar{\alpha}_i \bar{\mathbf{x}}_i$, where the $\bar{\mathbf{x}}_i$ are delayed versions of the *moved-out* data, and the $\bar{\alpha}_i$ are more unknowns. To suppress both signature and reverberation simultaneously, we need to express both "statements of wishes" in the same domain, either moved out or not. Letting $\mathbf{N}$ be the moveout operator, and choosing the moved-out domain, we write the statement of wishes as

$$\bar{\mathbf{d}} \quad \approx \quad \sum_i \bar{\alpha}_i \bar{\mathbf{x}}_i \; + \; \sum_i \alpha_i \mathbf{N} \mathbf{x}_i \tag{9.1}$$

Why not estimate the filters sequentially instead of simultaneously? What fails if we first process raw data by blind deconvolution for the source signature, then do NMO, and finally do blind deconvolution again for reverberation?

At vertical incidence, both filters are convolutional, and they are indistinguishable. At vertical incidence, doing a stage of deconvolution for each process leads to nonsensical answers. Whichever stage is applied first will absorb all the color in the data, leaving nothing for the second stage. The color will not be properly distributed between the stages. In principle, at nonzero offset the information is present to distinguish between the stages, but the first stage will always tend to absorb the color attributable to both. A simpler expression of the same concept arises when we are regressing two theoretical signals against some data. If the regressors are orthogonal, such as a mean value and a sinusoid, then we tend to get the same result regardless of the order in which we subtract them from the signal. If the regressors resemble one another, as a mean can resemble a trend, then they must be estimated simultaneously.

9.2.3. A model for convolution both before and after NMO

Here we will develop a formal theory for (9.1). By formalizing the theory, we will see better how it can be made more precise, and how the wishes expressed by (9.1) are a linearization of a nonlinear theory.

For a formal model, we will need definitions. Simple multiple reflections are generated by $1/(1+cZ^n)$, where c is a reflection coefficient and Z^n is the two-way traveltime to the water bottom. We will express reflectivity as an unspecified filter $R(Z)$, so the reverberation operator as a whole is $1/(1+R(Z))$, where $R(Z)$ is like the adjustable coefficients in a gapped filter. This form is partly motivated by the idea that $1 > |R|$. Taking x_t to denote the reflection coefficients versus depth or the multiple-free seismogram, and taking y_t to denote the one-dimensional seismogram with multiples, we find that the relation between them is conveniently expressed with Z-transforms as $Y(Z) = X(Z)/(1+R(Z))$.

Likewise, we will express the source signature not as a convolution but as an inverse polynomial (so designature turns into convolution). Suppose that source signature as a whole is given by the operator $1/(1+S(Z))$. The final data $D(Z)$ is related to the impulse-source seismogram $Y(Z)$ by $D(Z) = Y(Z)/(1+S(Z))$.

The trouble with the definitions above is that they are in the Fourier domain.

Since we are planning to mix in the NMO operator, which stretches the time axis, we will need to reexpress everything in the time domain. Instead of $X(Z) = Y(Z)(1 + R(Z))$ and $Y(Z) = D(Z)(1 + S(Z))$, we will use shifted-column matrices to denote convolution. Thus our two convolutions can be written as

$$\mathbf{x} = (\mathbf{I} + \mathbf{R})\mathbf{y} \tag{9.2}$$

$$\mathbf{y} = (\mathbf{I} + \mathbf{S})\mathbf{d} \tag{9.3}$$

where **I** is an identity matrix. Combining these two, we have a transformation from the data to the reflection coefficients for a one-dimensional seismogram. Departures from one-dimensionality arise from NMO and from spherical **divergence** of amplitude. Simple theory (energy distributed on the area of an expanding sphere) suggests that the scaling factor t converts the amplitude of **y** to **x**. So we define a matrix **T** to be a diagonal with the weight t distributed along it.

We need also to include the time shifts of NMO. In chapter 5 we saw that NMO is a matrix in which the diagonal line is changed to a hyperbola. Denote this matrix by **N**. Let $\mathbf{y}_0$ be the result of attempting to generate a zero-offset signal from a signal at any other offset by correcting for divergence and moveout:

$$\mathbf{y_0} = \mathbf{NTy} \tag{9.4}$$

The NMO operator can be interpreted in two ways, depending on whether we plan to find one filter for all offsets, or one for each. In other words, we can decide if we want one set of earth reflection coefficients applicable to all offsets, or if we want a separate reflection coefficient at each offset. From chapter 7 we recall that the more central question is whether to include summation over offset in the NMO operator. If we choose to include summation, then the adjoint sprays the same one-dimensional seismogram out to each offset at the required moveout. This choice determines if we have one filter for each offset, or if we use the same filter at all offsets.

Equation (9.2) actually refers only to zero offset. Thus it means $\mathbf{x} = (\mathbf{I}+\mathbf{R})\mathbf{y_0}$. Merging this with equations (9.3) and (9.4) gives

$$\mathbf{x} = (\mathbf{I}+\mathbf{R})\mathbf{NT}(\mathbf{I}+\mathbf{S})\mathbf{d} \tag{9.5}$$

$$\mathbf{x} = \mathbf{NTd}+\mathbf{RNTd}+\mathbf{NTSd}+\mathbf{RNTSd} \tag{9.6}$$

Now it is time to think about what is known and what is unknown. The unknowns will be the reverberation operators $\mathbf{R}$ and $\mathbf{S}$. Since we can only solve nonlinear problems by iteration, we linearize by dropping the term that is the product of unknowns, namely, the last term in (9.6). This is justified if the unknowns are small,

and they might be small, since they are predictions. Otherwise, we must iterate, which is the usual solution to a nonlinear problem by a sequence of linearizations. The linearization is

$$\mathbf{x} \quad = \quad (\mathbf{NTd} + \mathbf{RNTd} + \mathbf{NTSd}). \tag{9.7}$$

When a product of Z-transforms is expressed with a shifted-column matrix, we have a choice of which factor to put in the matrix and which in the vector. The unknown belongs in the vector so that simultaneous equations can be the end result. We need, therefore, to rearrange the capital and lower-case letters in (9.7) to place all unknowns in vectors. Also, besides the original data $\mathbf{d}$, we will be regressing on processed data $\bar{\mathbf{d}}$, defined by

$$\bar{\mathbf{d}} \quad = \quad \mathbf{NTd} \tag{9.8}$$

Equation (9.7) thus becomes

$$\mathbf{x} \quad = \quad \bar{\mathbf{d}} + \bar{\mathbf{D}}\mathbf{r} + \mathbf{NTDs} \tag{9.9}$$

Now the unknowns are vectors.

Recall that the unknowns are like prediction filters. Everything in $\mathbf{x}$ that is predictable by $\mathbf{r}$ and $\mathbf{s}$ is predicted in an effort to minimize the power in $\mathbf{x}$. During

the process we can expect $\mathbf{x}$ to tend to whiteness. Thus our statement of wishes is

$$\mathbf{0} \quad \approx \quad \bar{\mathbf{d}} + \bar{\mathbf{D}}\mathbf{r} + \mathbf{NTDs} \tag{9.10}$$

Equation (9.10) is about the same as (9.1). To see this, associate $-\mathbf{r}$ with $\bar{\alpha}$ and associate $-\mathbf{s}$ with α. To make (9.10) look more like a familiar overdetermined system, I write it as

$$\bar{\mathbf{d}} \quad \approx \quad \begin{bmatrix} -\bar{\mathbf{D}} & -\mathbf{NTD} \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ \mathbf{s} \end{bmatrix} \tag{9.11}$$

Some years ago I tested this concept on a small selection of data, including Yilmaz and Cumro dataset 27, used in Figure 9.6. The signature waveform of this dataset was hardly measurable, and almost everything was in the reverberation. Thus, results nearly equal to Figure 9.6 could be obtained by omitting the deconvolution before NMO. Although I was unable to establish by field-data trials that simultaneous deconvolution is necessary, I feel that theory and synthetic studies would show that it is.

9.2.4. Heavy artillery

In Figure 9.6, we can see that events remain which look suspiciously like multiple reflections. Careful inspection of the data (rapid blinking on a video screen) convinced me that the problem lay in imperfect modeling of depth-variable velocity. It is not enough to use a depth-variable velocity in the NMO (a constant velocity was used in Figure 9.6), because primary and multiple reflections have different velocities at the same time. I used instead a physical technique called “diffraction" (explained in detail in IEI) to make the regressors. Instead of simply shifting on the time axis, diffraction shifts on the depth axis, which results in subtle changes in hyperbola curvature.

The downward-continuation result is significantly better than the NMO result, but it does contain some suspicious reflections (boxed). My final effort, shown on the right, includes the idea that the data contains random noise which could be windowed away in velocity space. To understand how this was done, recall that the basic model is $\mathbf{d} \approx \sum_i \alpha_i \mathbf{x}_i$, where $\mathbf{d}$ is the left panel, α_i are constants determined by least squares, and $\mathbf{x}_i$ are the regressors, which are panels like $\mathbf{d}$ but delayed and diffracted. Let $\mathbf{V}$ denote an operator that transforms to velocity space. Instead of solving the regression $\mathbf{d} \approx \sum_i \alpha_i \mathbf{x}_i$, I solved the regression $\mathbf{V}\mathbf{d} \approx \sum_i \alpha_i \mathbf{V}\mathbf{x}_i$ and used

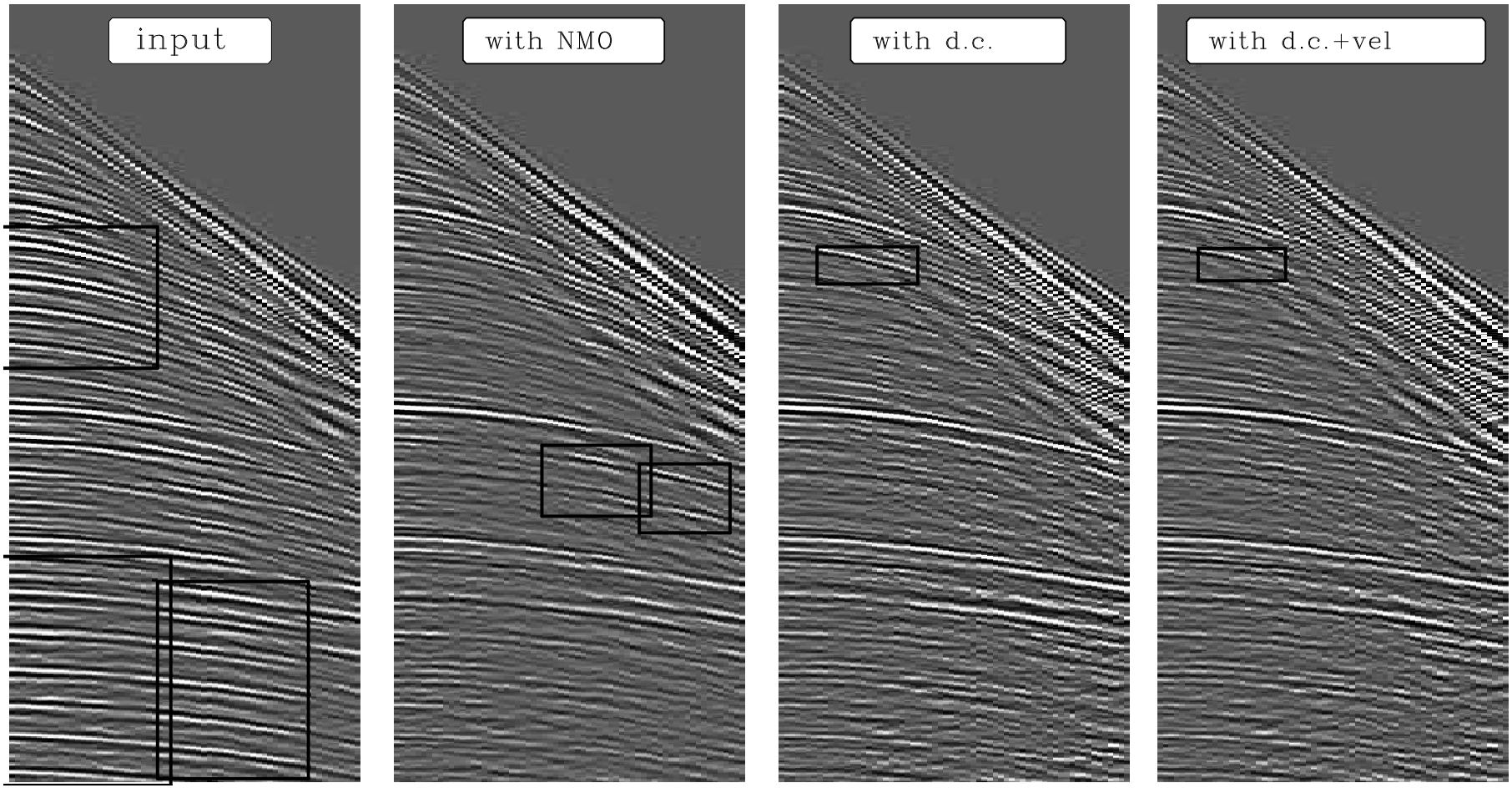


Figure 9.9: Left is the original data. Next is the result of using NMO in the regressors. Next, the result of downward continuation in the regressors. On the right, velocity scans were also used. Rectangles outline certain or likely multiple reflections. hyp-veld [NR]

the resulting values of α_i in the original (t, x)-space. (Mathematically, I did the same thing when making Figure **??**.) This procedure offers the possible advantage that a weighting function can be used in the velocity space. Applying all these ideas, we see that a reflector remains which looks more like a multiple than a primary.

A regression ($\mathbf{d} \approx \sum_i \alpha_i \mathbf{x}_i$) can be done in any space. You must be able to transfer into that space (that is, to make $\mathbf{Vd}$ and $\mathbf{Vx}_i$) but you do not need to be able to transform back from that space (you do not need $\mathbf{V}^{-1}$). You should find the α_i in whatever space you are able to define the most meaningful weighting function.

A proper "industrial strength" attack on multiple reflections involves all the methods discussed above, wave-propagation phenomena described in IEI, and judicious averaging in the space of source and receiver distributions.

9.3. References

Claerbout, J.F., 1986, Simultaneous pre-normal moveout and post-normal moveout

deconvolution: Geophysics, **51**, 1341-1354.

Chapter 10

Spectrum and phase

In this chapter we will examine

- 90° phase shift, analytic signal, and Hilbert transform.

- spectral factorization, i.e., finding a minimum-phase wavelet to fit any spectrum.
- a “cookbook” for Butterworth causal bandpass filters.
- phase delay, group delay, and beating.
- where the name “minimum phase” came from.
- what minimum phase implies for energy delay.

10.1. HILBERT TRANSFORM

Chapter 9 explains that many plots in this book have various interpretations. Superficially, the plot pairs represent cosine transforms of real even functions. But since the functions are even, their negative halves are not shown. An alternate interpretation of the plot pairs is that one signal is real and **causal**. This is illustrated in full detail in Figure 10.1. Half of the values in Figure 10.1 convey no information: these are the zero values at negative time, and the negative frequencies of the FT. In other

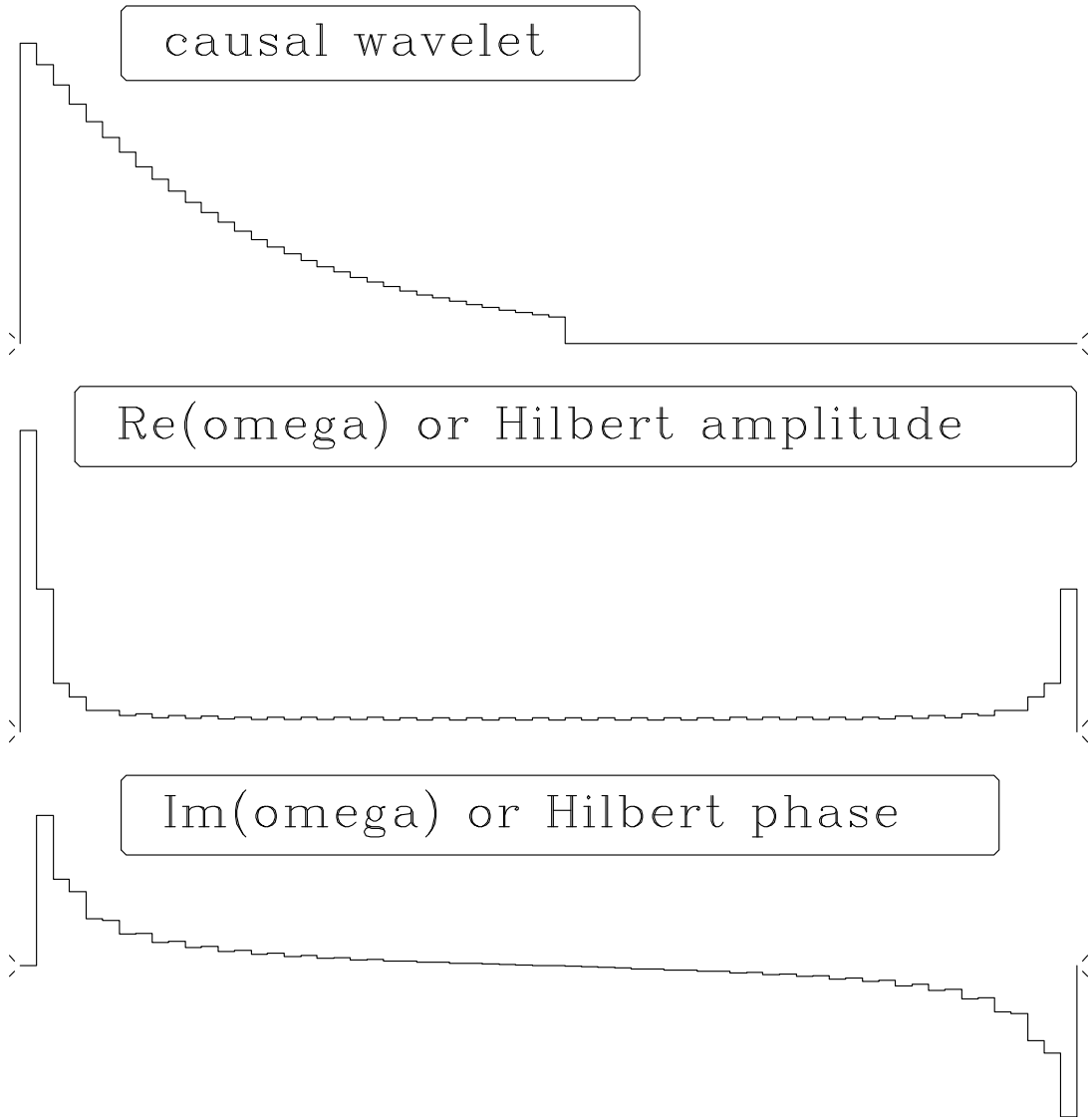


Figure 10.1: Both positive and negative times and frequencies of a real causal response (top) and real (mid) and imaginary (bottom) parts of its FT. spec-intro [NR]

words, the right half of Figure 10.1 is redundant, and is generally not shown. Likewise, the bottom plot, which is the imaginary part, is generally not shown, because it is derivable in a simple way from given information. Computation of the unseen imaginary part is called "**Hilbert transform**." Here we will investigate details and applications of the Hilbert transform. These are surprisingly many, including 90° phase-shift filtering, envelope functions, the instantaneous frequency function, and relating amplitude spectra to phase spectra.

Ordinarily a function is specified entirely in the time domain or entirely in the frequency domain. The Fourier transform then specifies the function in the other domain. The **Hilbert transform** arises when half the information is in the time domain and the other half is in the frequency domain. (Algebraically speaking, any fractional part could be given in either domain.)

10.1.1. A Z-transform view of Hilbert transformation

Let x_t be an even function of t. The definition $Z = e^{i\omega}$ gives $Z^{-n} + Z^n = 2\cos\omega n$; so

$$X(Z) \quad = \quad \cdots + x_1 Z^{-1} + x_0 + x_1 Z + x_2 Z^2 + \cdots \tag{10.1}$$

$$X(Z) \quad = \quad x_0 + 2x_1 \cos\omega + 2x_2 \cos 2\omega + \cdots \tag{10.2}$$

Now make up a new function $Y(Z)$ by replacing cosine by sine in (10.2):

$$Y(Z) \quad = \quad 2x_1 \sin\omega + 2x_2 \sin 2\omega + \cdots \tag{10.3}$$

Recalling that $Z = \cos\omega + i \sin\omega$, we see that all the negative powers of Z cancel from $X(Z) + iY(Z)$, giving a **causal** $C(Z)$:

$$C(Z) \quad = \quad \frac{1}{2}[X(Z) + iY(Z)] \quad = \quad \frac{1}{2}x_0 + x_1 Z + x_2 Z^2 + \cdots \tag{10.4}$$

Thus, for plot pairs, the causal response is c_t, the real part of the FT is equation (10.2), and the imaginary part not usually shown is given by equation (10.3).

10.1.2. The quadrature filter

Beginning with a causal response, we switched cosines and sines in the frequency domain. Here we do so again, except that we interchange the time and frequency domains, getting a more physical interpretation.

A filter that converts sines into cosines is called a “90° phase-shift filter" or a “**quadrature filter**." More specifically, if the input is $\cos(\omega t + \phi_1)$, then the output

should be $\cos(\omega t + \phi_1 - \pi/2)$. An example is given in Figure 10.2. Let $U(Z)$ denote

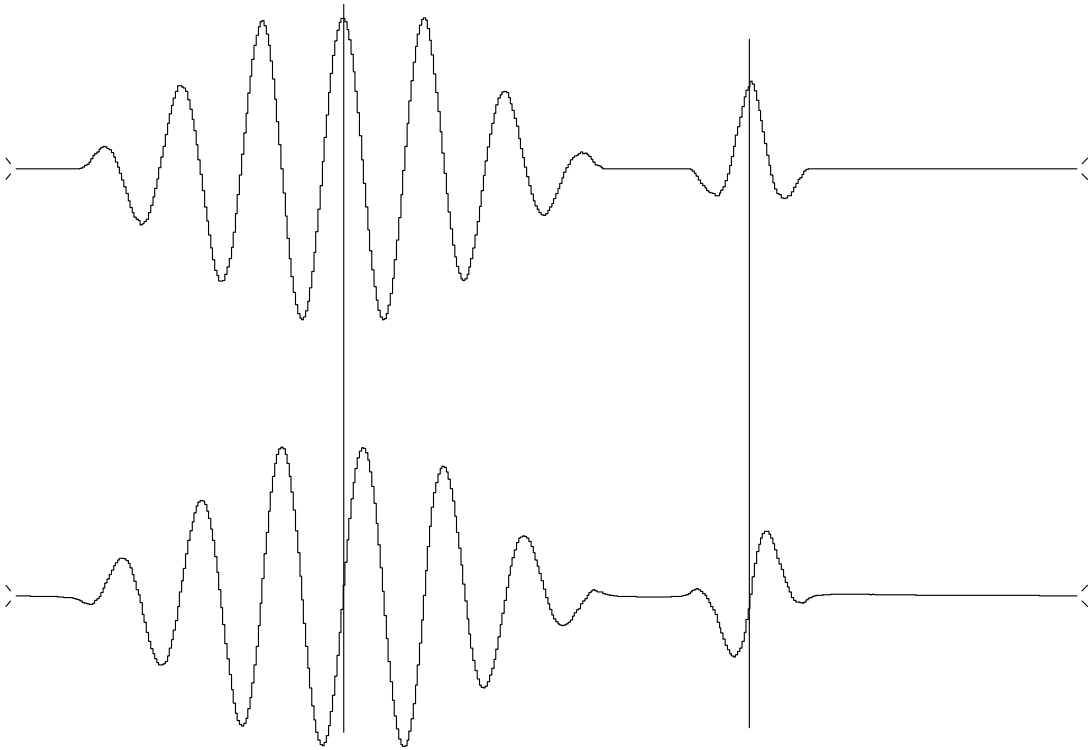


Figure 10.2: with quadrature filter yields phase-shifted signal (bottom).] Input (top) filtered with quadrature filter yields phase-shifted signal (bottom). spec-hilb0 [NR]

the Z-transform of a real signal input and $Q(Z)$ denote a quadrature filter. Then the output signal is

$$V(Z) \quad = \quad Q(Z)\, U(Z) \tag{10.5}$$

Let us find the numerical values of q_t. The time-derivative operation has the 90° phase-shifting property we need. The trouble with a differentiator is that higher frequencies are amplified with respect to lower frequencies. Recall the FT and take its time derivative:

$$b(t) = \int B(\omega) e^{-i\omega t} d\omega \tag{10.6}$$

$$\frac{db}{dt} = \int -i\omega B(\omega) e^{-i\omega t} d\omega \tag{10.7}$$

Thus we see that time differentiation corresponds to the weight factor $-i\omega$ in the frequency domain. The weight $-i\omega$ has the proper phase but the wrong amplitude. The desired weight factor is

$$Q(\omega) = \frac{-i\omega}{|\omega|} = -i \operatorname{sgn} \omega \tag{10.8}$$

where **sgn** is the "**signum**" or "sign" function. Let us transform $Q(\omega)$ into the domain of sampled time $t = n$:

$$q_n = \frac{1}{2\pi} \int_{-\pi}^{\pi} Q(\omega) e^{-i\omega n} d\omega \tag{10.9}$$

$$
\begin{aligned}
&= \frac{i}{2\pi}\int_{-\pi}^{0} e^{-i\omega n}d\omega - \frac{i}{2\pi}\int_{0}^{\pi} e^{-i\omega n}d\omega \\
&= \frac{i}{2\pi}\left(\frac{e^{-i\omega n}}{-in}\Big|_{-\pi}^{0} - \frac{e^{-i\omega n}}{-in}\Big|_{0}^{\pi} \right) \\
&= \frac{1}{2\pi n}\left(-1 + e^{+in\pi} + e^{-in\pi} - 1\right) \\
&= \begin{cases} 0 & \text{for } n \text{ even} \\ \frac{-2}{\pi n} & \text{for } n \text{ odd} \end{cases}
\end{aligned}
\tag{10.10}
$$

Examples of filtering with q_n are given in Figure 10.2 and 10.3.

Since q_n does not vanish for negative n, the quadrature filter is nonrealizable (that is, it requires future inputs to create its present output). If we were discussing signals in continuous time rather than sampled time, the filter would be of the form $1/t$, a function that has a singularity at $t = 0$ and whose integral over positive t is divergent. Convolution with the filter coefficients q_n is therefore painful because the infinite sequence drops off slowly. Convolution with the filter q_t is called “Hilbert transformation."

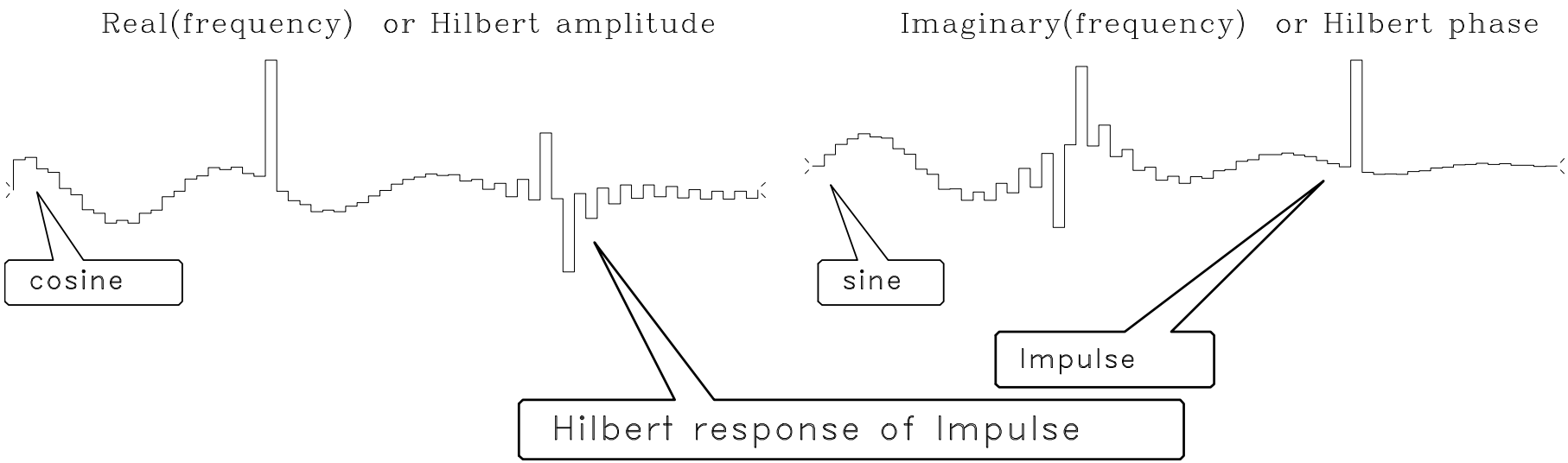


Figure 10.3: A Hilbert-transform pair. spec-hilb [NR]

10.1.3. The analytic signal

The so-called **analytic signal** can be constructed from a real-valued time series u_t and itself 90° phase shifted, i.e., v_t can be found using equation (10.5). The analytic signal is g_t, where

$$G(Z) \quad = \quad U(Z) + i\,V(Z) \quad = \quad [1 + i\,Q(Z)]\,U(Z) \tag{10.11}$$

In the time domain, the filter $[1 + i\,Q(Z)]$ is $\delta_t + i\,q_t$, where δ_t is an impulse function at time $t = 0$. The filter $1 + i\,Q(Z) = 1 + \omega/|\omega|$ vanishes for negative ω. Thus it is a real *step* function in the *frequency* domain. The values all vanish at **negative frequency**.

We can guess where the name "**analytic signal**" came from if we think back to Z-transforms and causal functions. Causal functions are free of **pole**s inside the unit circle, so they are "analytic" there. Their causality is the Fourier dual to the one-sidedness we see here in the frequency domain.

A function is "analytic" if it is one-sided in the dual (Fourier) domain.

10.1.4. Instantaneous envelope

The **quadrature filter** is often used to make the **envelope** of a signal. The envelope signal can be defined by $e_t = \sqrt{u_t^2 + v_t^2}$. Alternatively, with the **analytic signal** $g_t = u_t + i\,v_t$, the squared envelope is $e_t^2 = g_t \bar{g}_t$.

A quick way to accomplish the 90° **phase-**shift operation is to use Fourier transformation. Begin with $u_t + i \cdot 0$, and transform it to the frequency domain. Then multiply by the step function. Finally, inverse transform to get $g_t = u_t + i\,v_t$, which is equivalent to $(\delta_t + i\,q_t) * u_t$.

Sinusoids have smooth **envelope** functions, but that does not mean real seismograms do. Figure 10.4 gives an example of a field profile and unsmoothed and smoothed envelopes. Before **smoothing**, the stepout (alignment) of the reflections is quite clear. In the practical world, alignment is considered to be a manifestation of phase. An envelope should be a smooth function, such as might be used to scale data without altering its phase. Hence the reason for smoothing the envelope.

If you are interested in wave propagation, you might recognize the possibility of using **analytic signal**s. Energy stored as potential energy is 90° out of phase with kinetic energy, so u_t might represent scaled pressure while v_t represents scaled

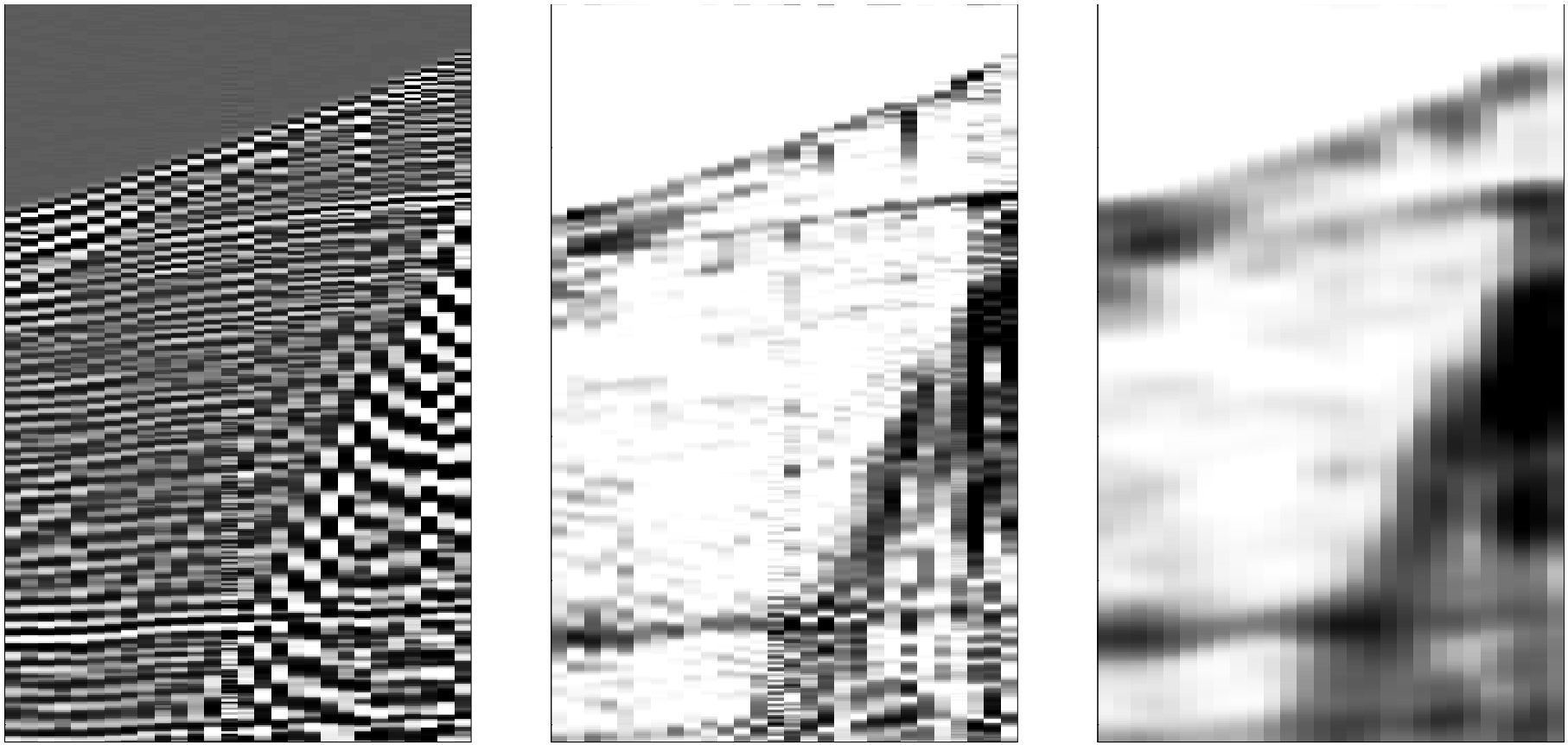


Figure 10.4: Left is a field profile. Middle is the unsmoothed envelope function. Right is the smoothed envelope. The vertical axis is time and the horizontal axis is space. Independent time-domain calculations are done at each point in space. spec-envelope [ER]

velocity. Then $\bar{w}_t w_t$ is the **instantaneous energy**. (The scales are the square root of compressibility and the square root of density.)

10.1.5. Instantaneous frequency

The phase ϕ_t of a **complex-valued signal** $g_t = u_t + i\, v_t$ is defined by $\phi_t = \arctan(v_t/u_t)$. The **instantaneous frequency** is $d\phi/dt$. Before forming the derivative, recall the definition of a complex logarithm of g:

$$\begin{aligned} g &= r e^{i\phi} \\ \ln g &= \ln|r| + \ln e^{i\phi} \\ &= \ln|r| + i\phi \end{aligned} \tag{10.12}$$

Hence, $\phi = \Im \ln g$. The **instantaneous frequency** is

$$\omega_{\text{instantaneous}} = \frac{d\phi}{dt} = \Im \frac{d}{dt} \ln g(t) = \Im \frac{1}{g}\frac{dg}{dt} \tag{10.13}$$

For a signal that is a pure sinusoid, such as $g(t) = g_0 e^{i\omega t}$, equation (10.13) clearly gives the right answer. When various frequencies are simultaneously present, we can hope that (10.13) gives a sensible average.

Trouble can arise in (10.13) when the denominator g gets small, which happens whenever the **envelope** of the signal gets small. This difficulty can be overcome by careful **smoothing**. Rationalize the denominator by multiplying by the conjugate signal, and then smooth locally a little (as indicated by the summation sign below):

$$\hat{\omega}_{\text{smoothed}} \quad = \quad \Im \frac{\sum \bar{g}(t) \, \frac{d}{dt} g(t)}{\sum \bar{g}(t) \, g(t)} \tag{10.14}$$

(Those of you who have studied **quantum mechanics** may recognize the notion of "expectation of an operator." You will also see why the wave probability function of quantum **physics** must be complex valued: as a consequence of the **analytic signal** eliminating negative frequencies from the average. If the negative frequencies were not eliminated, then the average frequency would be zero.)

What range of times should be smoothed in equation (10.14)? Besides the nature of the data, the appropriate smoothing depends on the method of representing $\frac{d}{dt}$. To prepare a figure, I implemented $\frac{d}{dt}$ by multiplying by $-i\omega$. (This is more accurate than finite differences at high frequencies, but has the disadvantage that the discontinuity in slope at the Nyquist frequency gives an extended transient in the time domain.) The result is shown in Figure 10.5. Inspection of the figure shows that

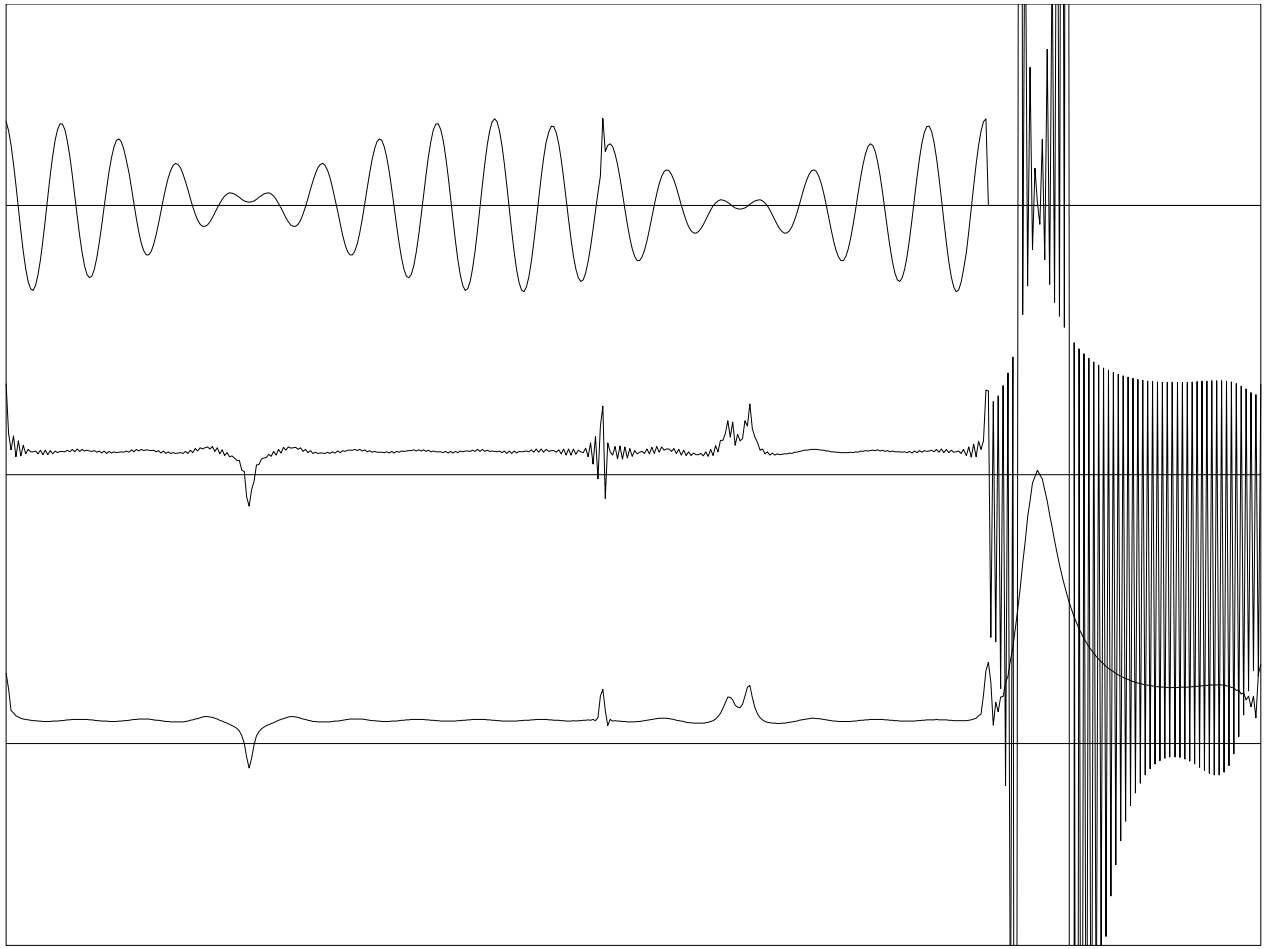


Figure 10.5: A sum of three sinusoids (top), unsmoothed instantaneous frequency (middle), and smoothed instantaneous frequency (bottom). spec-node [NR]

smoothing is even more necessary for instantaneous frequency than for envelopes, and this is not surprising because the presence of $\frac{d}{dt}$ makes the signal rougher. Particularly notice times in the range 400-512 where the sinusoids are truncated. There the unsmoothed instantaneous frequency becomes a large rapid oscillation near the Nyquist frequency. This roughness is nicely controlled by $(1, 2, 1)$ smoothing.

It is gratifying to see that a spike added to the sinusoids (at point 243) causes a burst of high frequency. Also interesting to notice is where an oscillation approaches the axis and then turns away just before or just after crossing the axis.

An example of **instantaneous frequency** applied to field data is shown in Figure 10.6.

The instantaneous-frequency idea can also be applied to the space axis. This will be more easily understood by readers familiar with the methodology of imaging and migration. Instead of temporal frequency $\omega = d\phi/dt$, we compute the spatial frequency $k_x = d\phi/dx$. Figure 10.7 gives an example. Analogously, we could make plots of local dip k_x/ω.

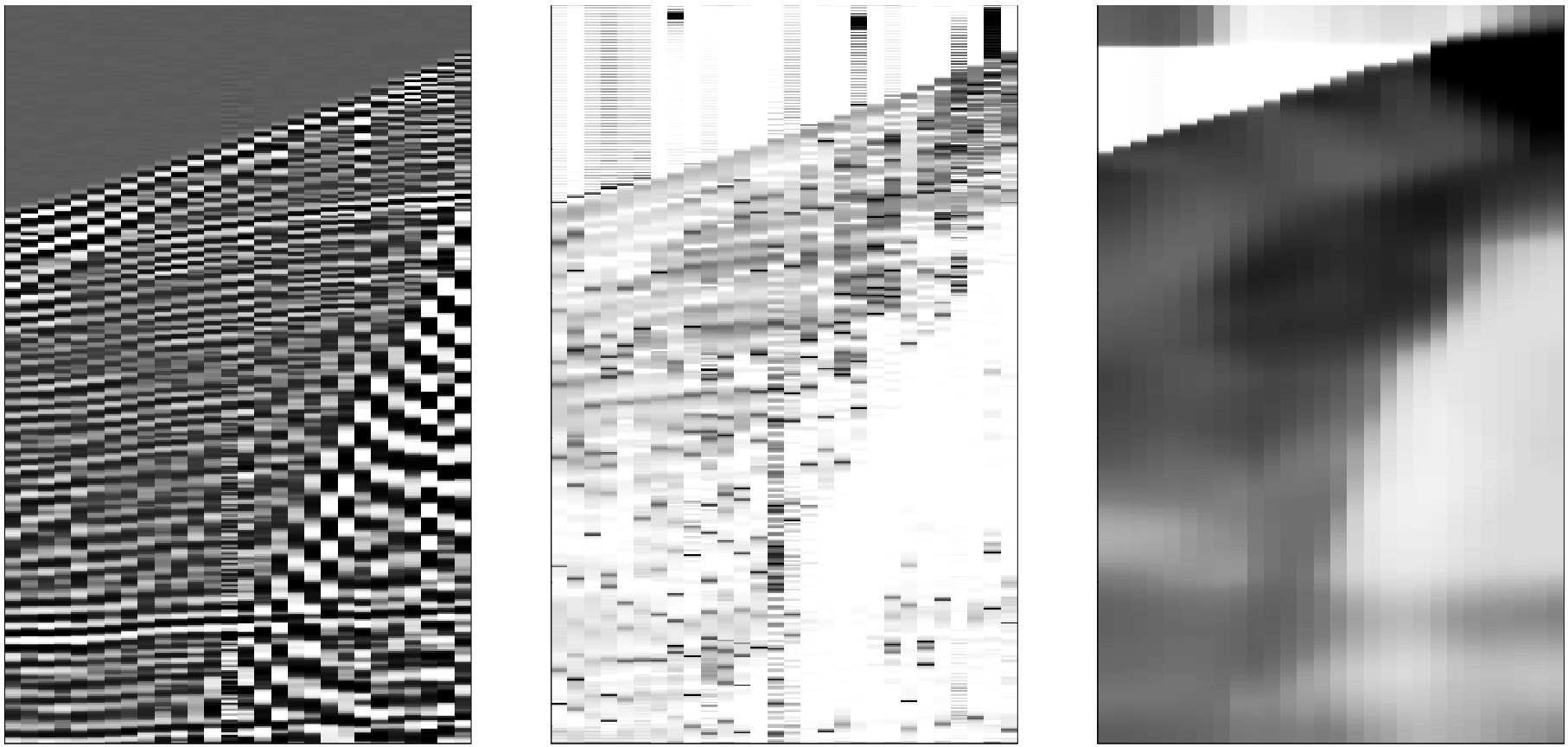


Figure 10.6: A field profile (left), instantaneous frequency smoothed only with (1,2,1) (middle), and smoothed more heavily (right). spec-frequency [ER]

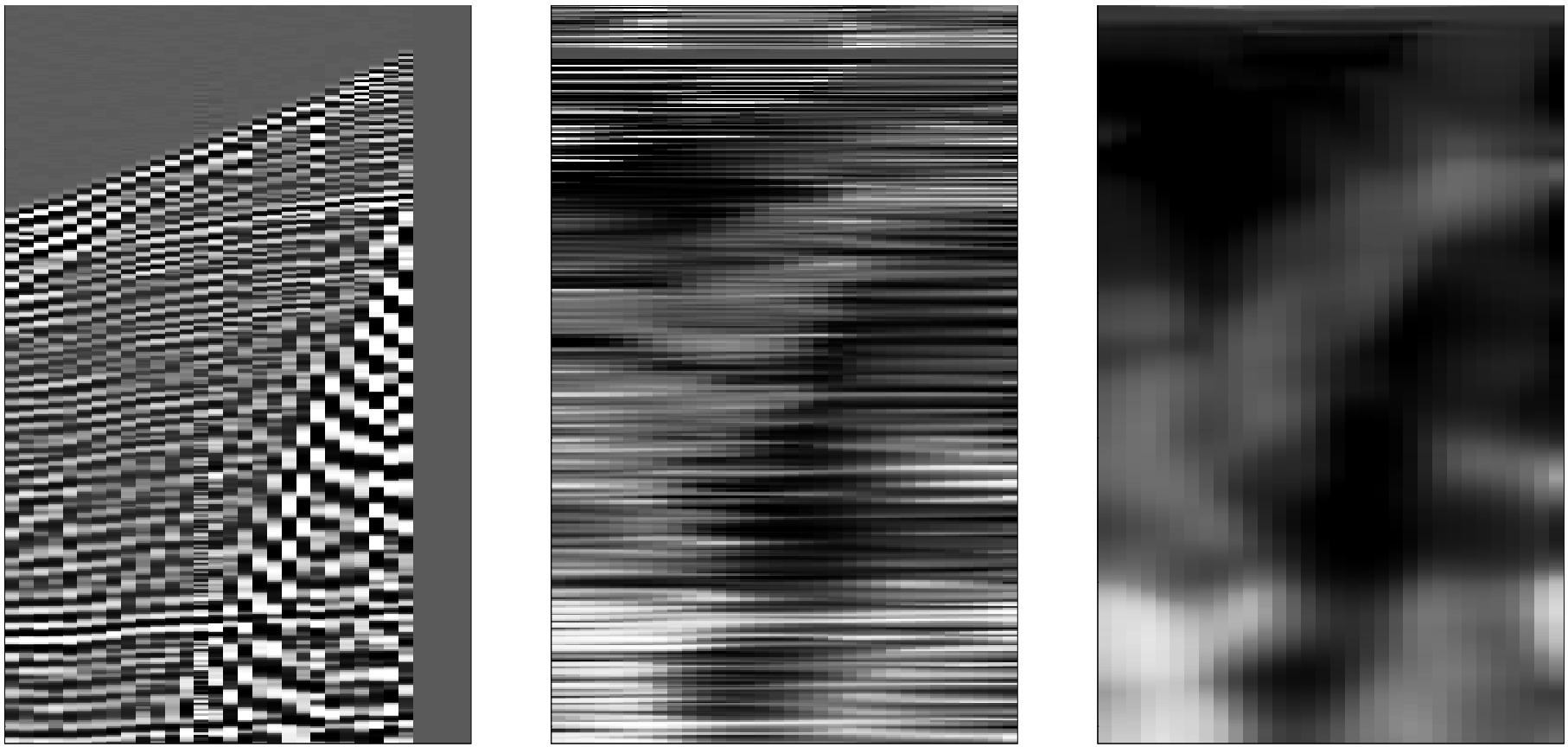


Figure 10.7: A field profile (left), k_x smoothed over x only (center), and smoothed over t and x (right). spec-kx [ER]

EXERCISES:

1 Let c_t be a causal complex-valued signal. How does $X(Z)$ change in equation (10.2), and how must $Y(Z)$ in equation (10.3) be deduced from $X(Z)$?

2 Figure 10.3 shows a Hilbert-transform pair, the real and imaginary parts of the Fourier transform of a causal response. Describe the causal response.

3 Given $Y(Z) = Q(Z)X(Z)$, prove that the envelope of y_t is the same as the envelope of x_t.

4 Using partial fractions, convolve the waveform

$$\frac{2}{\pi}\left(\ldots, -\frac{1}{5}, 0, -\frac{1}{3}, 0, -1, 0, 1, 0, \frac{1}{3}, 0, \frac{1}{5}, \ldots\right)$$

with itself. What is the interpretation of the fact that the result is $(\ldots, 0, 0, -1, 0, 0, \ldots$
(HINT: $\pi^2/8 = 1 + \frac{1}{9} + \frac{1}{25} + \frac{1}{49} + \ldots$.)

5 Using the fast-Fourier-transform matrix, we can represent the **quadrature filter** $Q(\omega)$ by the column vector

$$-i(0, 1, 1, 1, \ldots, 0, -1, -1, -1, \ldots, -1)'$$

Multiply this vector into the inverse-transform matrix to show that the transform is proportional to $(\cos \pi k/N)/(\sin \pi k/N)$. What is the scale factor? Sketch the scale factor for $k \ll N$, indicating the limit $N \to \infty$. (HINT: $1 + x + x^2 + \ldots + x^N = (1 - x^{N+1})/(1 - x)$.)

10.2. SPECTRAL FACTORIZATION

The "spectral factorization" problem arises in a variety of physical contexts. It is this: given a spectrum, find a minimum-phase wavelet that has that spectrum. We will see how to make this wavelet, and we will recognize that it is unique. (It is unique except for a trivial aspect. The negative of any wavelet has the same spectrum as the wavelet, and, more generally, any wavelet can be multiplied by any complex number of unit magnitude, such as $\pm i$, etc.)

First consider the simpler problem in which the wavelet need not be causal. We can easily find a symmetric wavelet with any spectrum (which by definition is an energy or power). We simply take the square root of the spectrum—this is the **amplitude spectrum**. We then inverse transform the amplitude spectrum to the time domain, and we have a symmetric wavelet with the desired spectrum.

The **prediction-error filter** discussed in chapter 7 is theoretically obtainable by spectral factorization of an inverse spectrum. The **Kolmogoroff** method of spectral factorization, which we will be looking at here, is much faster than the time-domain, least-squares methods considered in chapter 7 and the least-squares methods given in FGDP. Its speed motivates its widespread practical use.

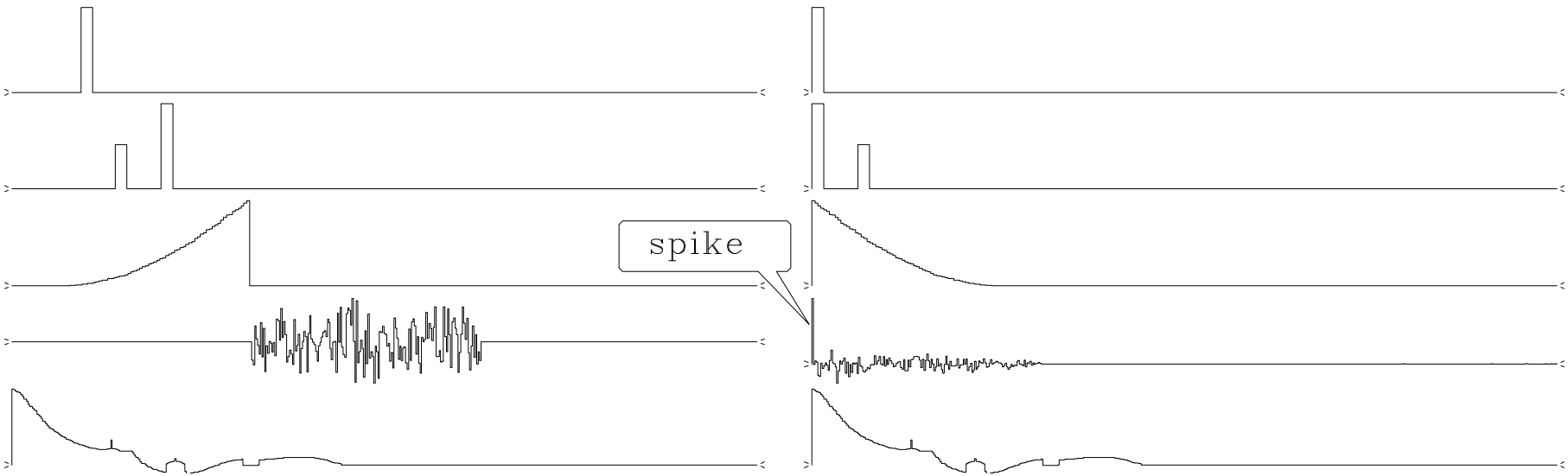


Figure 10.8: Left are given wavelets, and right are minimum-phase equivalents. spec-mpsamples [NR]

Some simple examples of spectral factorization are given in Figure 10.8. For all but the fourth signal, the spectrum of the minimum-phase wavelet clearly matches that of the input. Wavelets are shifted to $t = 0$ and turned backwards. In the fourth case, the waveshape changes into a big pulse at zero lag. As the **Robinson** theorem introduced on page 605 suggests, minimum-phase wavelets tend to decay rapidly after a strong onset. I imagined that hand-drawn wavelets with a strong onset would rarely turn out to be perfectly minimum-phase, but when I tried it, I was surprised at how easy it seemed to be to draw a minimum-phase wavelet. This is shown on the bottom of Figure 10.8.

To begin understanding spectral factorization, notice that the polar form of any complex number puts the phase into the exponential, i.e., $x + iy = |r|e^{i\phi} = e^{\ln|r| + i\phi}$. So we look first into the behavior of exponentials and logarithms of Fourier transforms.

10.2.1. The exponential of a causal is causal.

Begin with a **causal** response c_t and its associated $C(Z)$. The Z-transform $C(Z)$ could be evaluated, giving a complex value for each real ω. This complex value

could be exponentiated to get another value, say

$$B(Z(\omega)) \quad = \quad e^{C(Z(\omega))} \tag{10.15}$$

Next, we could inverse transform $B(Z(\omega))$ back to b_t. We will prove the amazing fact that b_t must be causal too.

First notice that if $C(Z)$ has no negative powers of Z, then $C(Z)^2$ does not either. Likewise for the third power or any positive integer power, or sum of positive integer powers. Now recall the basic power-series definition of the exponential function:

$$e^x \quad = \quad 1+x+\frac{x^2}{2}+\frac{x^3}{2\cdot 3}+\frac{x^4}{2\cdot 3\cdot 4}+\frac{x^5}{2\cdot 3\cdot 4\cdot 5}+\cdots \tag{10.16}$$

Including equation (10.15) gives the **exponential of a causal**:

$$B(Z) \quad = \quad e^{C(Z)} \quad = \quad 1+C(Z)+\frac{C(Z)^2}{2}+\frac{C(Z)^3}{2\cdot 3}+\frac{C(Z)^4}{2\cdot 3\cdot 4}+\cdots \tag{10.17}$$

Each term in the infinite series corresponds to a causal response, so the sum, b_t, is causal. (If you have forgotten the series for the exponential function, then recall that the solution to $dy/dx = y$ is the definition of the exponential function $y(x) = e^x$,

and that the power series satisfies the differential equation term by term, so it must be the exponential too. The factorials in the denominators assure us that the power series always converges, i.e., it is finite for any finite x.)

Putting one polynomial into another or one infinite series into another is an onerous task, even if it does lead to a wavelet that is exactly causal. In practice we do operations that are conceptually the same, but for speed we do them with discrete Fourier transforms. The disadvantage is periodicity, i.e., negative times are represented computationally like negative frequencies. Negative times are the last half of the elements of a vector, so there can be some blurring of late times into negative ones.

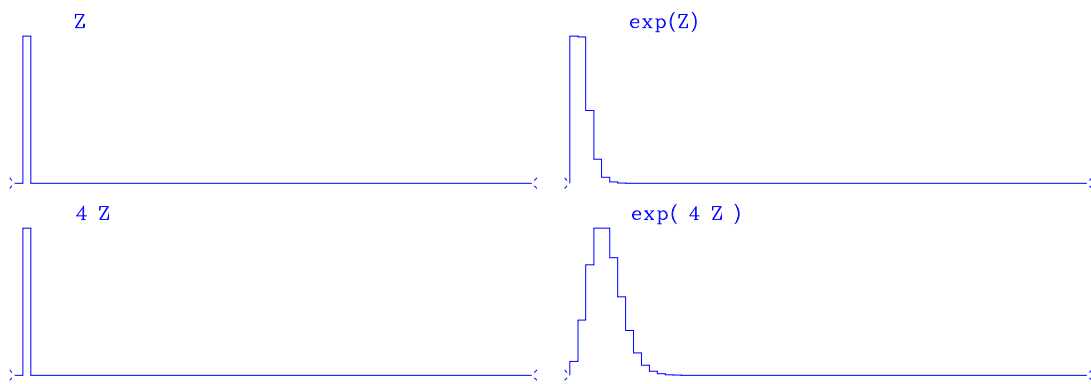


Figure 10.9: Exponentials. spec-eZ [NR]

Figure 10.9 gives examples of equation (10.17) for $C = Z$ and $C = 4Z$. Un-

fortunately, I do not have an analytic calculation to confirm the validity of these examples.

10.2.2. Finding a causal wavelet from a prescribed spectrum

To find a causal wavelet from a prescribed **spectrum**, we will need to form the logarithm of the spectrum. Since a spectrum can easily vanish, and since the logarithm of zero is infinite, there is a pitfall. To prepare ourselves, we first examine the log spectra example given in Figure 10.10. On the infinite domain, the FT of a box function is a sinc whose zeros become minus infinities in the logarithm. On the discrete domain, exact zeros may occur or not. The transform of a triangle is a sinc squared, but since this triangle was imperfectly drawn (by me), its transform does not go identically to zero. The sinc function drops off as ω, and sinc squared drops off as ω^2. We confirm this on the logarithm plot: sinc squared drops off *twice* as much.

Now for the task of going from a spectrum to a causal wavelet. Take as given the spectrum of the causal wavelet $B(Z)$. This means that we are not given $B(Z)$

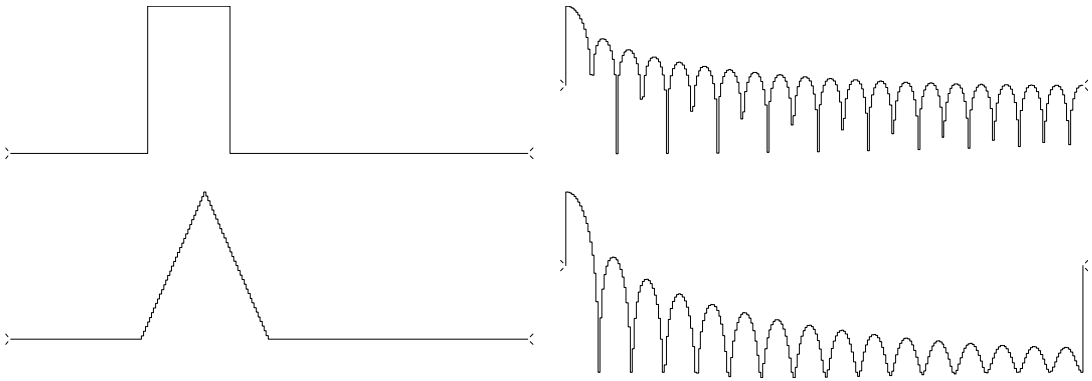


Figure 10.10: Log spectra of a box function and a triangle function. spec-logspec [NR]

itself, but we are given $S(Z) = \overline{B}(1/Z)B(Z)$. Assuming no zeros in the spectrum $S(\omega)$, it is easy to find the log of the spectrum $U = \ln S$. The spectrum may be specified as **autocorrelation** coefficients or values on the unit circle. Thus,

$$\overline{B}(1/Z)\,B(Z) = S(Z) = e^{\ln S(Z)} = e^{U(Z)} = e^{\overline{C}(1/Z)+C(Z)} = e^{\overline{C}(1/Z)}\,e^{C(Z)} \quad (10.18)$$

Given the spectrum $S(Z)$ for each value on the unit circle, we could deduce the **log spectrum** $U(Z) = \ln S(Z)$ at each point on the unit circle:

$$U(Z) \quad = \quad \ln[S(Z)] \quad = \quad \overline{C}(1/Z) + C(Z) \quad (10.19)$$

This is the answer we have been looking for. Given $U(Z)$ for all real values of ω, we could inverse transform to the time domain, obtaining the two-sided function

$u_t = \bar{c}_{-t} + c_t$. Setting to zero the coefficients at negative times eliminates $\bar{c}_{-t}$, leaving just c_t; hence $C(Z)$. And we already know that the exponential of $C(Z)$ gives $B(Z)$ with a causal b_t. This method is known as "**Kolmogoroff spectral factorization**," after the mathematician who discovered it.

The program `mpwave()` below begins with a wavelet, forms its spectrum, and then calls `kolmogoroff()` to factor the spectrum. The program `kolmogoroff()` first takes the logarithm of the spectrum, then returns to the time domain and sets to zero the noncausal part. It returns to frequency, exponentiates, and returns to the time domain with a wavelet that will be proven to be **minimum-phase**. mpwave kolmogoroff Between the times when negative lags are set to zero and positive lags are left untouched are two points that are scaled by half. The overall scaling was chosen to preserve the scale of the input wavelet.

The first test I tried on this program was the input wavelet $(1, 2, 0, 0)$. The desired result is that the wavelet should time-reverse itself to $(2, 1, 0, 0)$. The actual result was (1.9536, 1.0837, 0.0464, -0.0837), imperfect because the four-point Fourier transform is a summation around the unit circle, whereas theoretically an integration is called for. Therefore, better results can be obtained by padding additional zeros after the input wavelet. Also, you might notice that the program is

```
subroutine mpwave( n, cx)       # minimum phase equivalent wavelet
integer i,         n            # input:  cx = any wavelet
complex cx(n)                   # output: cx = min phase wavelet
call ftu( 1., n, cx)            #               with same spectrum.
call scaleit( sqrt(1.*n), 2*n, cx)
do i= 1, n
        cx(i) = cx(i) * conjg( cx(i))
call kolmogoroff( n, cx)
return; end
```

Back

```
subroutine kolmogoroff( n, cx)  # Spectral factorization.
integer i,              n       # input:  cx = spectrum
complex cx(n)                   # output: cx = min phase wavelet
do i= 1, n
        cx(i) = clog( cx(i) )
call ftu( -1., n, cx);          call scaleit( sqrt(1./n), 2*n, cx)
cx(1    ) = cx(1    ) / 2.
cx(1+n/2) = cx(1+n/2) / 2.
do i= 1+n/2+1, n
        cx(i) = 0.
call ftu( +1., n, cx);          call scaleit( sqrt(1.*n), 2*n, cx)
do i= 1, n
        cx(i) = cexp( cx(i))
call ftu( -1., n, cx);          call scaleit( sqrt(1./n), 2*n, cx)
return; end
```

Back

designed for complex-valued signals. As typical of Fourier transform with single-word **precision**, the imaginary parts were about 10^{-6} of the real parts instead of being precisely zero.

Some examples of **spectral factorization** are given in Figure 10.11.

10.2.3. Why the causal wavelet is minimum-phase

Next we see why the causal wavelet $B(Z)$, which we have made from the prescribed spectrum, turns out to be minimum-phase. First return to the original definition of minimum-phase: a causal wavelet is minimum-phase if and only if its inverse is causal. We have our wavelet in the form $B(Z) = e^{C(Z)}$. Consider another wavelet $A(Z) = e^{-C(Z)}$, constructed analogously. By the same reasoning, a_t is also causal. Since $A(Z)B(Z) = 1$, we have found a causal, inverse wavelet. Thus the b_t wavelet is **minimum-phase**.

Since the **phase** is a Fourier series, it must be periodic; that is, it cannot increase indefinitely with ω as it does for the nonminimum-phase wavelet (see Figure 10.19).

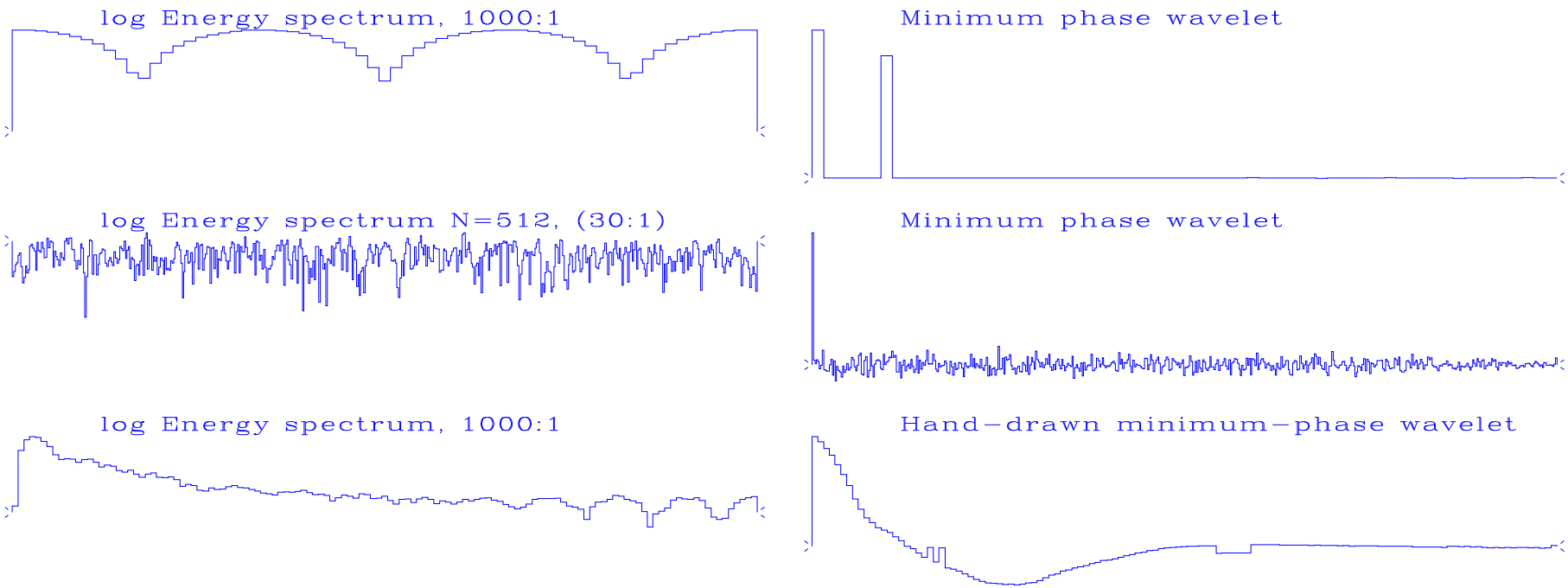


Figure 10.11: Examples of log spectra and their associated minimum-phase wavelets. spec-example [NR]

10.2.4. Pathological examples

The **spectral-factorization** algorithm fails when an infinity is encountered. This happens when the spectrum becomes zero, so that its logarithm becomes minus infinity. This can occur in a benign way—for example, in the case of the spectrum of the wavelet $(1, 1)$, where the infinity occurs at the Nyquist frequency. We could smooth the spectrum near the Nyquist before we take the logarithm. On the other hand, the pathology can be more extreme. Convolving $(1, 1)$ with itself N times, we see that the result and its spectrum tend to **Gaussian**s. So, at the Nyquist frequency, smoothing would only replace zero by a very tiny number.

Figure 10.12 shows functions whose spectra contain zeros, along with their minimum-phase equivalents. When the logarithm of zero arises during the computation, it is replaced by the log of 10^{-30}. It is surprising that the triangle suffered so much less than the other two functions. It seems that minor imperfection in specifying the triangle resulted in a spectrum that did not have the theoretical zeros of sinc squared.

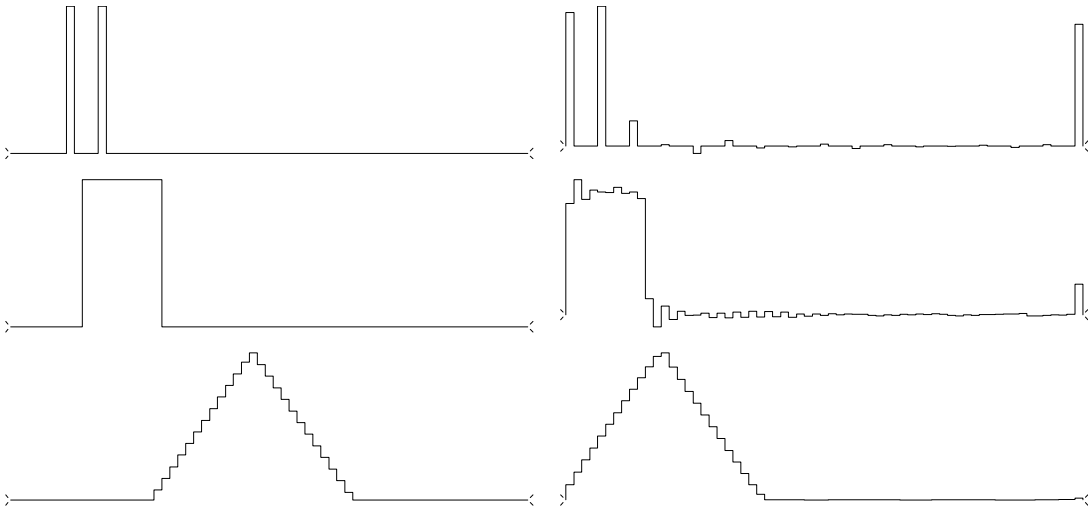


Figure 10.12: Functions whose spectra contain zeros, along with their minimum-phase equivalents, as computed by discrete Fourier transform. spec-patho [NR]

10.2.5. Relation of amplitude to phase

As we learned from equation (10.19), a minimum-phase function is determined completely from its spectrum. Thus its **phase** is determinable from its **spectrum**. Likewise, we will see that, except for a scale, the spectrum is determinable from the phase.

So far we have not discussed the fact that spectral factorization implicitly uses **Hilbert transform**ation. Somehow we simply generated a phase. To see how the

phase arose, recall equation (10.18) and (10.19):

$$S_k = e^{\ln S_k} = e^{U_k} = e^{(U_k - i\Phi_k)/2} e^{(U_k + i\Phi_k)/2} = e^{\overline{C}_k} e^{C_k} = \overline{B}_k B_k \qquad (10.20)$$

Where did Φ_k come from? We took $U_k + i0$ to the time domain, obtaining u_t. Then we multiplied u_t by a real-valued step function of time. This multiplication in the time domain is what created the phase, because multiplication in the time domain implies a convolution in the frequency domain. Recall that the Fourier transform of a real-valued step function arises with Hilbert transform. Multiplying in time with a step means that, in frequency, U_k has been convolved with $\delta_{k=0} + i \times (90°$ phase-shift filter). So U_k is unchanged and a phase, Φ_k, has been generated. This explanation will be somewhat clearer if you review the Z-transform approach discussed at the beginning of the chapter, because there we can see both the frequency domain and the time domain in one expression.

To illustrate different classes of discontinuity, pulse, step, and slope, Figure 10.13 shows another Hilbert-transform pair.

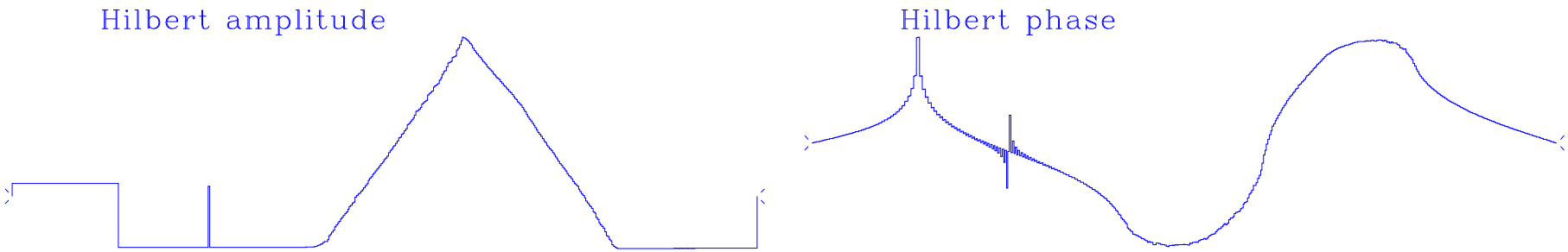


Figure 10.13: A Hilbert-transform pair. spec-hilb2 [NR]

EXERCISES:

1 What is the meaning of *minimum-phase waveform* if the roles of the time and frequency domains are interchanged?

2 Show how to do the inverse Hilbert transform: given ϕ, find u. What is the interpretation of the fact that we cannot get u_0?

3 Consider a model of a portion of the earth where x is the north coordinate, $+z$ represents altitude above the earth, and magnetic bodies are distributed in the earth, creating no component of **magnetic field** in the east-west direction. We

can show that the magnetic field h above the earth is represented by

$$\begin{bmatrix} h_x(x,z) \\ h_z(x,z) \end{bmatrix} = \int_{-\infty}^{+\infty} F(k) \begin{bmatrix} -ik \\ |k| \end{bmatrix} e^{ikx-|k|z}\, dk$$

Here $F(k)$ is some spatial frequency spectrum.

(a) By using Fourier transforms, how do you compute $h_x(x,0)$ from $h_z(x,0)$ and vice versa?

(b) Given $h_z(x,0)$, how do you compute $h_z(x,z)$?

(c) Notice that, at $z=0$,

$$f(x) = h_z(x) + i h_x(x) = \int_{-\infty}^{+\infty} e^{ikx} F(k)(|k|+k)\, dk$$

and that $F(k)(|k|+k)$ is a one-sided function of k. With a total field magnetometer we observe that

$$h_x^2(x) + h_z^2(x) = w(x)\bar{w}(x)$$

What can you say about obtaining $F(k)$ from this?

(d) How unique are $h_x(x)$ and $h_z(x)$ if $f(x)\,\overline{f}(x)$ is given?

4 Test this idea: write code to factor $X(Z)$ into $X(Z) = A(Z)B(Z)$, where $B(Z)$ is minimum-phase and $A(Z)$ is maximum-phase. Maximum-phase means that $Z^N A(1/Z)$ is minimum-phase. First compute $U(\omega) = \ln X(\omega)$. Then remove a linear trend in the phase of $U(\omega)$ to get N. Then split U with its trend removed into causal and anticausal parts $U(Z) = C^-(1/Z) + C^+(Z)$. Finally, form $B(Z) = \exp C^+(Z)$ and $Z^N A(1/Z) = \exp(C^-(Z))$.

10.3. A BUTTERWORTH-FILTER COOKBOOK

An ideal bandpass filter passes some range of frequencies without distortion and suppresses all other frequencies. Further thought shows that what we think of as the ideal bandpass filter, a rectangle function of frequency, is instead far from ideal, because its time-domain representation $(\sin \omega_0 t)/(\omega_0 t)$ is noncausal and decays much too slowly with time for many practical uses. The appropriate bandpass filter is one whose time decay can be chosen to be reasonable (in combination with a reasonable necessary compromise on the shape of the rectangle). **Butterworth filter**s fulfill

these needs. They are causal and of various orders, the lowest order being best (shortest) in the time domain, and the higher orders being better in the frequency domain. Well-engineered projects often include Butterworth filters. Unfortunately they are less often used in experimental work because of a complicated setting-up issue that I am going to solve for you here. I will give some examples and discuss **pitfall**s as well.

The main problem is that there is no simple mathematical expression for the filter coefficients as a function of order and cutoff frequency.

Analysis starts from an equation that for large-order n is the equation of a box:

$$\overline{B(\omega)}B(\omega) \quad = \quad \frac{1}{1+\left(\frac{\omega}{\omega_0}\right)^{2n}} \tag{10.21}$$

When $|\omega| < \omega_0$, this Butterworth low-pass spectrum is about unity. When $|\omega| > |\omega_0|$, the spectrum drops rapidly to zero. The magnitude $|B(\omega)|$ (with some truncation effects to be described later) is plotted in Figure 10.14 for various values of n.

Conceptually, the easiest form of Butterworth filtering is to take data to the frequency domain and multiply by equation (10.21), where you have selected some

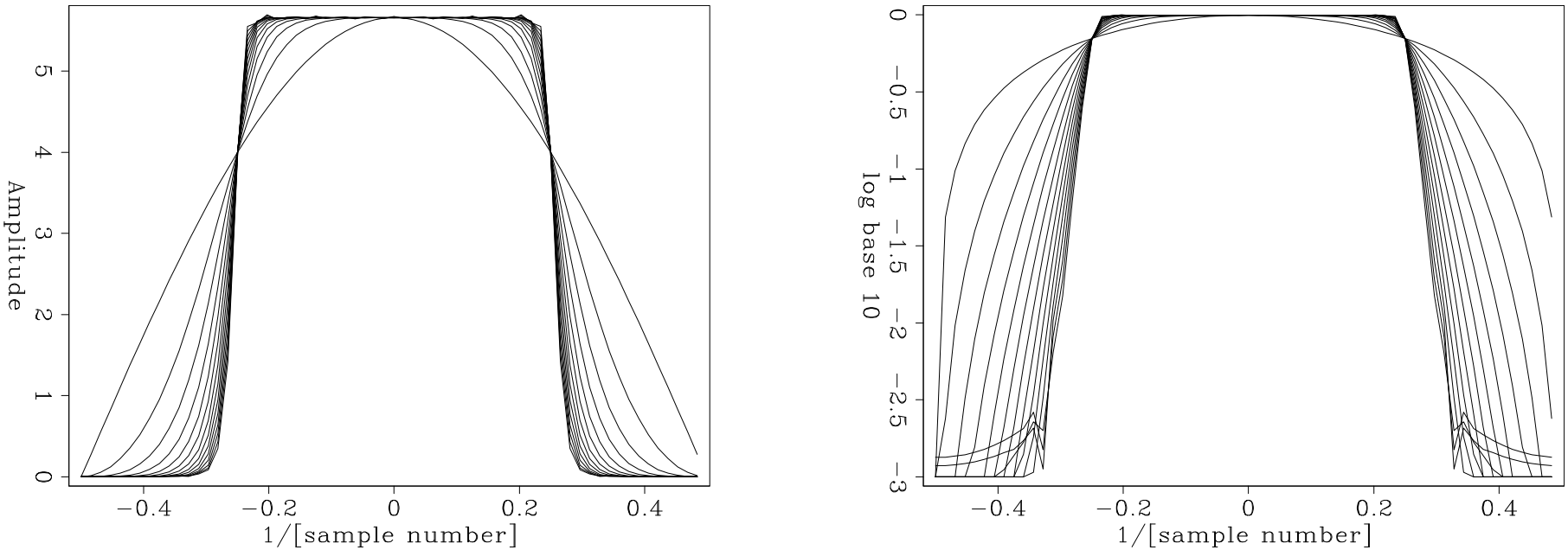


Figure 10.14: Spectra of Butterworth filters of various-order n. spec-butf [NR]

value of n to compromise between the demands of the frequency domain (sharp cutoff) and the time domain (rapid decay). Of course, the time-domain representation of equation (10.21) is noncausal. If you prefer a causal filter, you could take the Butterworth spectrum into a spectral-factorization program such as `kolmogoroff()`.

The time-domain response of the Butterworth filter is infinitely long, although a Butterworth filter of degree n can be well approximated by a ratio of n^{th}-order polynomials. Since, as we will see, n is typically in the range 2-5, time-domain filtering is quicker than FT. To proceed, we need to express ω in terms of Z, where $Z = e^{i\omega\Delta t}$. This is done in an approximate way that is valid for frequencies far from the Nyquist frequency. Intuitively we know that time differentiation is implied by $-i\omega$. We saw that in sampled time, differentiation is generally represented by the bilinear transform, equation (**??**): $-i\hat{\omega}\Delta t = 2(1-Z)/(1+Z)$. Thus a sampled-time representation of $\omega^2 = (i\omega)(-i\omega)$ is

$$\omega^2 \quad = \quad 4\,\frac{1-Z^{-1}}{1+Z^{-1}}\,\frac{1-Z}{1+Z} \tag{10.22}$$

Substituting equation (10.22) into (10.21) we find

$$B\left(\frac{1}{Z}\right)B(Z) = \frac{[(1+Z^{-1})(1+Z)]^n}{[(1+Z^{-1})(1+Z)]^n + [\frac{4}{\omega_0^2}(1-Z^{-1})(1-Z)]^n} \quad (10.23)$$

$$B\left(\frac{1}{Z}\right)B(Z) = \frac{N(Z^{-1})N(Z)}{D(Z^{-1})D(Z)} \quad (10.24)$$

where the desired, causal, Butterworth, discrete-domain filter is $B(Z) = N(Z)/D(Z)$. You will be able to appreciate the enormity of the task represented by these equations when you realize that the denominator in (10.23) must be factored into the product of a function of Z times the same function of Z^{-1} to get equation (10.24). Since the function is positive, it can be considered to be a spectrum, and factorization must be possible.

10.3.1. Butterworth-filter finding program

To express equation (10.23) in the Fourier domain, multiply every parenthesized factor by $\sqrt{Z}$ and recall that $\sqrt{Z}+1/\sqrt{Z}=2\cos(\omega/2)$. Thus,

$$\overline{B(\omega)}B(\omega) \quad = \quad \frac{(2\cos\omega/2)^{2n}}{(2\cos\omega/2)^{2n} \;+\; (\frac{4}{\omega_0}\sin\omega/2)^{2n}} \tag{10.25}$$

An analogous equation holds for high-pass filters. Subroutine `butter()` /prog:butter does both equations. First, the denominator of equation (10.25) is set up as a spectrum and factored. The numerator could be found in the same way, but the result is already apparent from the numerator of (10.23), i.e., we need the coefficients of $(1+Z)^n$. In subroutine `butter()` they are simply obtained by Fourier transformation. The occurrence of a tangent in the program arises from equation (**??**). butter

10.3.2. Examples of Butterworth filters

Spectra and log spectra of various orders of Butterworth filters are shown in Figure 10.14. They match a rectangle function that passes frequencies below the half-

```
# Find the numerator and denominator Z-transforms of the Butterworth filter.
#   hilo={1.,-1.} for {low,high}-pass filter
#   cutoff in Nyquist units, i.e. cutoff=1 for (1,-1,1,-1...)
#
subroutine butter( hilo, cutoff, npoly, num, den)
integer npoly, nn, nw, i
real hilo, cutoff, num(npoly), den(npoly), arg, tancut, pi
complex cx(2048)
pi = 3.14159265;        nw=2048;        nn = npoly - 1
tancut = 2. * tan( cutoff*pi/2. )
do i= 1, nw {
        arg = (2. * pi * (i-1.) / nw) / 2.
        if( hilo > 0. )                                 # low-pass filter
                cx(i) = (2.*cos(arg)              ) **(2*nn) +
                        (2.*sin(arg) * 2./tancut ) **(2*nn)
        else                                            # high-pass filter
                cx(i) = (2.*sin(arg)              ) **(2*nn) +
                        (2.*cos(arg) * tancut/2. ) **(2*nn)
        }
call kolmogoroff( nw, cx)       # spectral factorization
do i= 1, npoly
        den(i) = cx(i)
do i= 1, nw                     # numerator
        cx(i) = (1. + hilo * cexp( cmplx(0., 2.*pi*(i-1.)/nw))) ** nn
call ftu( -1., nw, cx)
do i= 1, npoly
        num(i) = cx(i)
return; end
```

Back

Nyquist. Convergence is rapid with order. The logarithm plot shows a range of 0-3, meaning an amplitude ratio of $10^3 = 1000$. Tiny glitches near the bottom for high-order curves result from truncating the time axis in the time domain shown in Figure 10.15. The time-domain truncation also explains a slight roughness on the

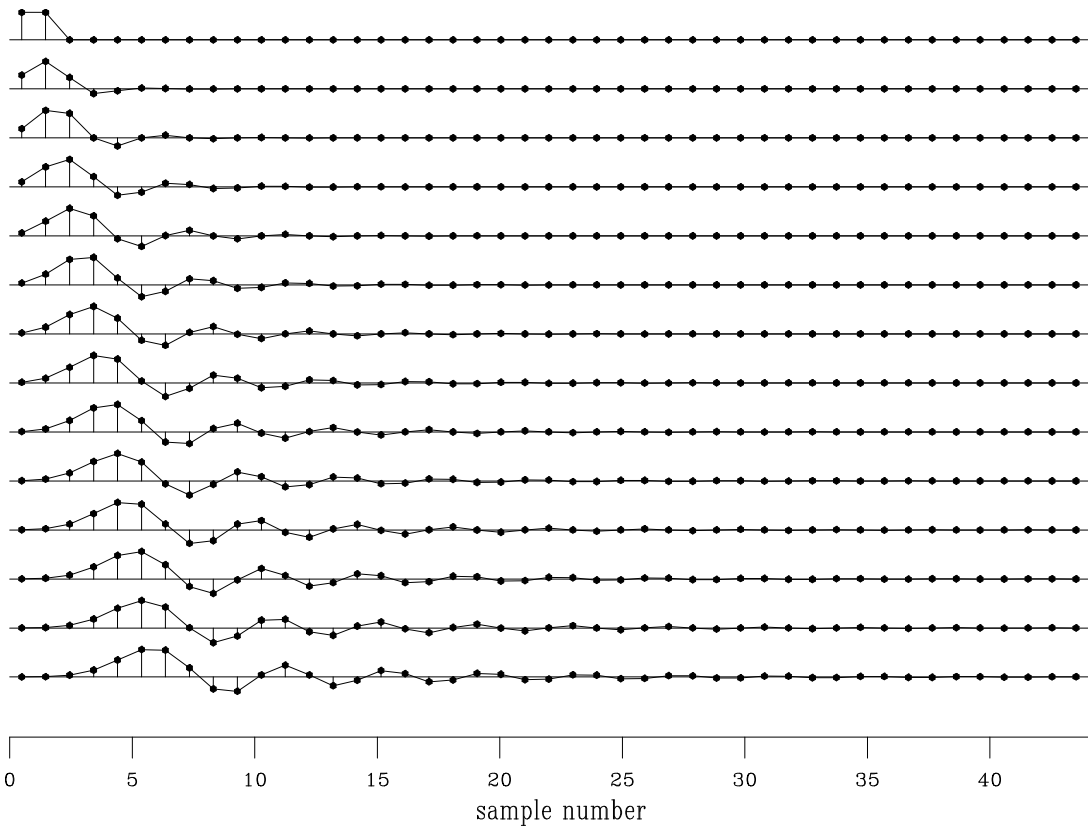


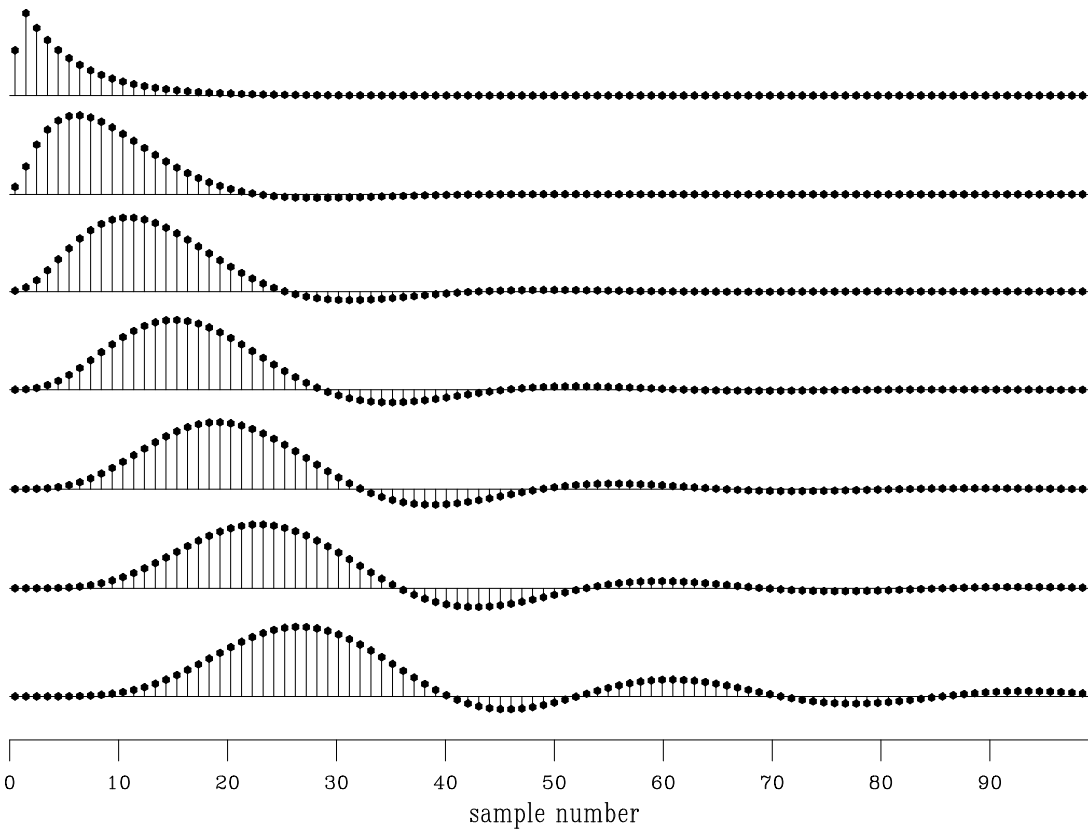
Figure 10.15: Butterworth-filter time responses for half-Nyquist low pass. spec-butm [ER]

top of the rectangle function.

In practice, the filter is sometimes run both forward and backward to achieve a phaseless symmetrical response. This squares the spectral amplitudes, resulting in convergence twice as fast as shown in the figure. Notice that the higher-order curves in the time domain (Figure 10.15) have undesirable sidelobes which ring longer with higher orders. Also, higher-order curves have increasing delay for the main signal burst. This delay is a consequence of the binomial coefficients in the numerator.

Another example of a low-pass Butterworth filter shows some lurking **instability**. This is not surprising: a causal bandpass operator is almost a contradiction in terms, since the word "**bandpass**" implies multiplying the spectrum by zero outside the chosen band, and the word "**causal**" implies a well-behaved spectral logarithm. These cannot coexist because the logarithm of zero is minus infinity. All this is another way of saying that when we use Butterworth filters, we probably should not use high orders. Figure 10.16 illustrates that an instability arises in the seventh-order Butterworth filter, and even the sixth-order filter looks suspicious. If we insist on using high-order filters, we can probably go to an order about twice as high as we began with by using double precision, increasing the spectral width `nw,` and,

Figure 10.16: Butterworth time responses for a narrow-band low-pass filter. spec-butl [ER]



if we are really persistent, using the method of the exercises below. My favorite Butterworth filters for making synthetic seismograms have five coefficients (fourth order). I do one pass through a low cut at `cutoff=.1` and another through a high cut at `cutoff=.4`.

EXERCISES:

1 Above we assumed that a bandpass filter should be made by cascading a low-pass and a high-pass filter. Suggest a revised form of equation (10.21) for making bandpass filters directly.

2 Notice that equation (10.21) can be factored analytically. Individual factors could be implemented as the Z-transform filters, and the filters cascaded. This prevents the instability that arises when many poles are combined. Identify the poles of equation (10.21). Which belong in the causal filter and which in its time reverse?

10.4. PHASE DELAY AND GROUP DELAY

The Fourier-domain ratio of a wave seen at B divided by a wave seen at A can be regarded as a filter. The propagation velocity is the distance from A to B divided by the delay. There are at least two ways to *define* the delay, however.

10.4.1. Phase delay

Whenever we put a sinusoid into a filter, a sinusoid must come out. The only things that can change between input and output are the amplitude and the phase. Comparing a zero crossing of the input to a zero crossing of the output measures the so-called **phase** delay. To quantify this, define an input, $\sin\omega t$, and an output, $\sin(\omega t - \phi)$. Then the phase delay t_p is found by solving

$$\begin{aligned} \sin(\omega t - \phi) &= \sin\omega(t - t_p) \\ \omega t - \phi &= \omega t - \omega t_p \\ t_p &= \frac{\phi}{\omega} \end{aligned} \tag{10.26}$$

A problem with phase delay is that the phase can be ambiguous within an additive constant of $2\pi N$, where N is any integer. In wave-propagation theory, "phase ve-

locity" is defined by the distance divided by the phase delay. There it is hoped that the $2\pi N$ ambiguity can be resolved by observations tending to zero frequency or physical separation.

10.4.2. Group delay

A more interesting kind of delay is "**group delay**," corresponding to **group velocity** in wave-propagation theory. Often the group delay is nothing more than the phase delay. This happens when the phase delay is independent of frequency. But when the phase delay depends on frequency, then a completely new velocity, the "group velocity," appears. Curiously, the group velocity is *not* an average of phase velocities.

The simplest analysis of group delay begins by defining a filter input x_t as the sum of two frequencies:

$$x_t \quad = \quad \cos\omega_1 t + \cos\omega_2 t \tag{10.27}$$

By using a trigonometric identity,

$$x_t \quad = \quad 2 \underbrace{\cos(\frac{\omega_1 - \omega_2}{2} t)}_{\text{beat}} \cos(\frac{\omega_1 + \omega_2}{2} t) \tag{10.28}$$

we see that the sum of two cosines looks like a cosine of the average frequency multiplied by a cosine of half the difference frequency. Since the frequencies in Figure 10.17 are taken close together, the difference frequency factor in (10.28)

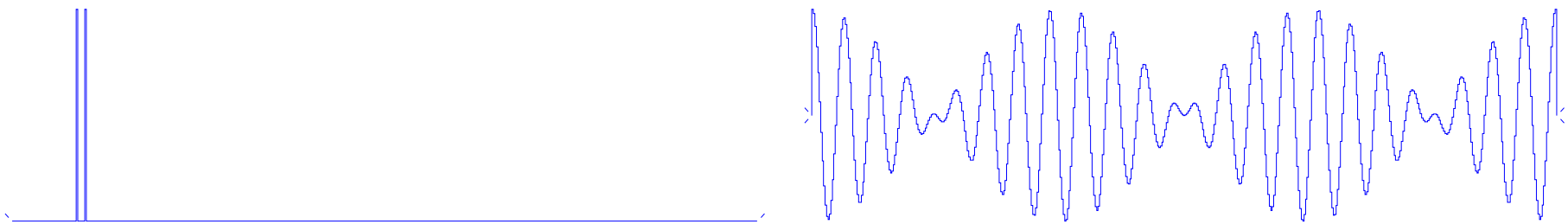


Figure 10.17: Two nearby frequencies beating. spec-beat [NR]

represents a slowly variable amplitude multiplying the average frequency. The

slow (difference frequency) modulation of the higher (average) frequency is called "**beating**."

The beating phenomenon is also called "**interference**," although that word is deceptive. If the two sinusoids were two wave beams crossing one another, they would simply cross *without* interfering. Where they are present simultaneously, they simply add.

Each of the two frequencies could be delayed a different amount by a filter, so take the output of the filter y_t to be

$$y_t = \cos(\omega_1 t - \phi_1) + \cos(\omega_2 t - \phi_2) \tag{10.29}$$

In doing this, we have assumed that neither frequency was attenuated. (The **group velocity** concept loses its simplicity and much of its utility in dissipative media.) Using the same trigonometric identity on (10.29) as we used to get (10.28), we find that

$$y_t = 2 \underbrace{\cos\left(\frac{\omega_1 - \omega_2}{2} t - \frac{\phi_1 - \phi_2}{2}\right)}_{\text{beat}} \cos\left(\frac{\omega_1 + \omega_2}{2} t - \frac{\phi_1 + \phi_2}{2}\right) \tag{10.30}$$

Rewriting the beat factor in terms of a time delay t_g, we now have

$$\cos[\frac{\omega_1-\omega_2}{2}(t-t_g)] = \cos(\frac{\omega_1-\omega_2}{2}t-\frac{\phi_1-\phi_2}{2}) \quad (10.31)$$

$$(\omega_1-\omega_2)t_g = \phi_1-\phi_2$$

$$t_g = \frac{\phi_1-\phi_2}{\omega_1-\omega_2} = \frac{\Delta\phi}{\Delta\omega} \quad (10.32)$$

For a continuum of frequencies, the **group delay** is

$$t_g = \frac{d\phi}{d\omega} \quad (10.33)$$

10.4.3. Group delay as a function of the FT

We will see that the group delay of a filter P is a simple function of the Fourier transform of the filter. I have named the filter P to remind us that the theorem strictly applies only to all-pass filters, though in practice a bit of energy absorption might be OK. The phase angle ϕ could be computed as the arctangent of the ratio of imaginary to real parts of the Fourier transform, namely, $\phi(\omega) = \arctan[\Im P(\omega)/\Re P(\omega)]$.

As with (10.12), we use $\phi = \Im \ln P$; and from (10.33) we get

$$t_g \quad = \quad \frac{d\phi}{d\omega} \quad = \quad \Im \frac{d}{d\omega} \ln P(\omega) \quad = \quad \Im \frac{1}{P} \frac{dP}{d\omega} \tag{10.34}$$

which could be expressed as the Fourier dual to equation (10.14).

10.4.4. Observation of dispersive waves

Various formulas relate energy delay to group delay. This chapter illuminates those that are one-dimensional. In observational work, it is commonly said that "what you see is the group velocity." This means that when we see an apparently sinusoidal wave train, its distance from the source divided by its traveltime (group delay) is the group velocity. An interesting example of a dispersive wave is given in FGDP (Figure 1-11).

10.4.5. Group delay of all-pass filters

We have already discussed (page **??**) all-pass filters, i.e., filters with constant unit spectra. They can be written as $P(Z)\overline{P}(1/Z) = 1$. In the frequency domain, $P(Z)$

can be expressed as $e^{i\phi(\omega)}$, where ϕ is real and is called the "phase shift." Clearly, $P\overline{P} = 1$ for all real ϕ. It is an easy matter to make a filter with any desired phase shift—we merely Fourier transform $e^{i\phi(\omega)}$ into the time domain. If $\phi(\omega)$ is arbitrary, the resulting time function is likely to be two-sided. Since we are interested in physical processes that are causal, we may wonder what class of functions $\phi(\omega)$ corresponds to one-sided time functions. The answer is that the **group delay** $\tau_g = d\phi/d\omega$ of a causal **all-pass filter** must be positive.

Proof that $d\phi/d\omega > 0$ for a causal all-pass filter is found in FGDP; there is no need to reproduce the algebra here. The proof begins from equation (**??**) and uses the imaginary part of the logarithm to get phase. Differentiation with respect to ω yields a form that is recognizable as a spectrum and hence is always positive.

A single-pole, single-zero all-pass filter passes all frequency components with constant gain and a phase shift that can be adjusted by the placement of the pole. Taking Z_0 near the unit circle causes most of the phase shift to be concentrated near the frequency where the pole is located. Taking the pole farther away causes the delay to be spread over more frequencies. Complicated phase shifts or group delays can be built up by cascading single-pole filters.

The above reasoning for a single-pole, single-zero all-pass filter also applies to

many roots, because the phase of each will add, and the sum of $\tau_g = d\phi/d\omega > 0$ will be greater than zero.

The Fourier dual to the positive group delay of a causal all-pass filter is that the instantaneous frequency of a certain class of analytic signals must be positive. This class of analytic signals is made up of all those with a constant envelope function, as might be approximated by field data after the process of automatic gain control.

EXERCISES:

1 Let x_t be some real signal. Let $y_t = x_{t+3}$ be another real signal. Sketch the phase as a function of frequency of the cross-spectrum $X(1/Z)Y(Z)$ as would a computer that put all arctangents in the principal quadrants $-\pi/2 <$ arctan $< \pi/2$. Label the axis scales.

2 Sketch the amplitude, phase, and group delay of the all-pass filter $(1-\overline{Z_0}Z)/(Z_0 - Z)$, where $Z_0 = (1+\epsilon)e^{i\omega_0}$ and ϵ is small. Label important parameters on the curve.

3 Show that the coefficients of an all-pass, phase-shifting filter made by cascading $(1-\overline{Z_0}Z)/\ (Z_0 - Z)$ with $(1 - Z_0 Z)/(\overline{Z_0} - Z)$ are real.

4 A continuous signal is the impulse response of a continuous-time, all-pass filter. Describe the function in both time and frequency domains. Interchange the words “time" and “frequency" in your description of the function. What is a physical example of such a function? What happens to the statement, the group delay of an all-pass filter is positive?

5 A graph of the group delay $\tau_g(\omega)$ shows τ_g to be positive for all ω. What is the area under τ_g in the range $0 < \omega < 2\pi$? (HINT: This is a trick question you can solve in your head.)

10.5. PHASE OF A MINIMUM-PHASE FILTER

In chapter 3 we learned that the inverse of a causal filter $B(Z)$ is causal if $B(Z)$ has no roots inside the unit circle. The term “**minimum phase**” was introduced there without motivation. Here we examine the phase, and learn why it is called “minimum."

10.5.1. Phase of a single root

For real ω, a plot of real and imaginary parts of Z is the circle $(x, y) = (\cos\omega, \sin\omega)$. A smaller circle is $.9Z$. A right-shifted circle is $1 + .9Z$. Let Z_0 be a complex number, such as $x_0 + iy_0$, or $Z_0 = e^{i\omega_0}/\rho$, where ρ and ω_0 are fixed constants. Consider the complex Z plane for the two-term filter

$$B(Z) = 1 - \frac{Z}{Z_0} \tag{10.35}$$

$$B(Z(\omega)) = 1 - \rho e^{i(\omega-\omega_0)} \tag{10.36}$$

$$B(Z(\omega)) = 1 - \rho\cos(\omega-\omega_0) - i\rho\sin(\omega-\omega_0) \tag{10.37}$$

Real and imaginary parts of B are plotted in Figure 10.18. Arrows are at frequency ω intervals of 20°. Observe that for $\rho > 1$ the sequence of arrows has a sequence of angles that ranges over 360°, whereas for $\rho < 1$ the sequence of arrows has a sequence of angles between ±90°. Now let us replot equation (10.37) in a more conventional way, with ω as the horizontal axis. Whereas the **phase** is the angle of an arrow in Figure 10.18, in Figure 10.19 it is the arctangent of $\Im B/\Re B$. Notice how different is the phase curve in Figure 10.19 for $\rho < 1$ than for $\rho > 1$.

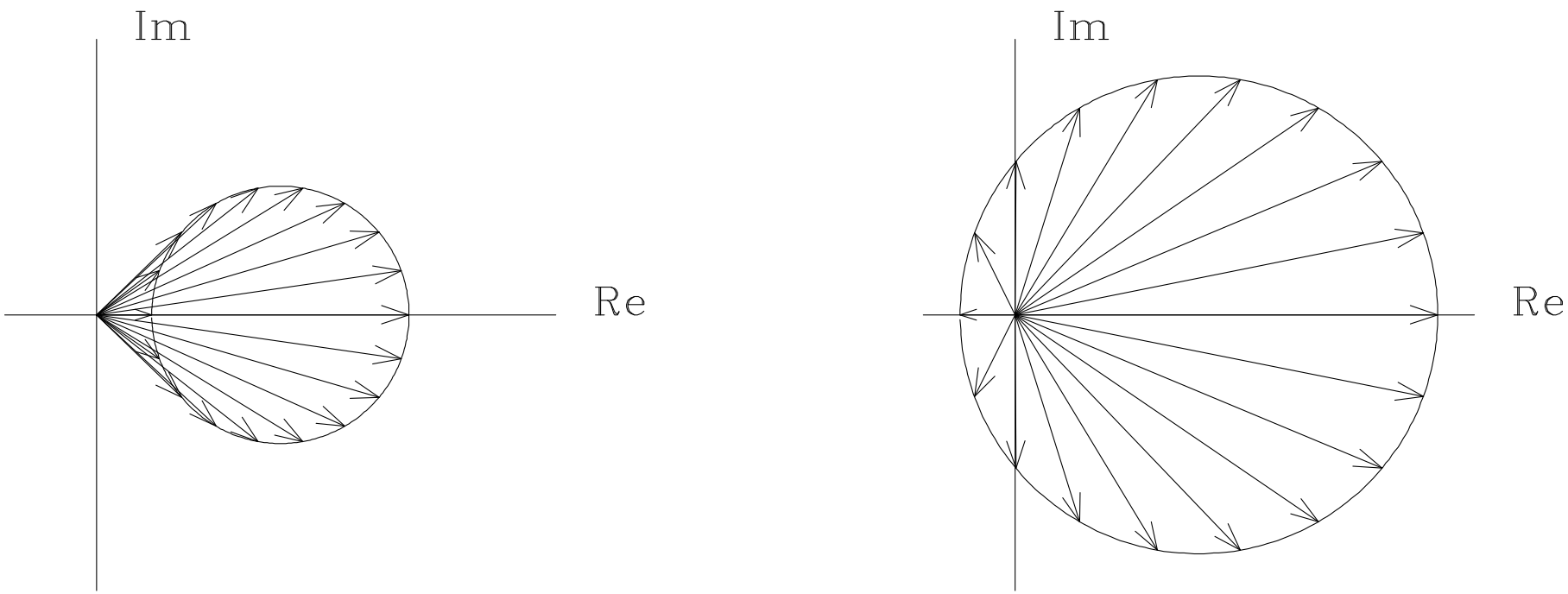


Figure 10.18: Left, complex B plane for $\rho < 1$. Right, for $\rho > 1$. spec-origin [ER]

Real and imaginary parts of B are *periodic* functions of the frequency ω, since $B(\omega) = B(\omega + 2\pi)$. We might be tempted to conclude that the phase would be periodic too. Figure 10.19 shows, however, that for a nonminimum-phase filter, as ω ranges from $-\pi$ to π, the phase ϕ increases by 2π (because the circular path in Figure 10.18 surrounds the origin). To make Figure 10.19 I used the Fortran arctangent function that takes two arguments, x, and y. It returns an angle between $-\pi$ and $+\pi$. As I was plotting the nonminimum phase, the phase suddenly jumped discontinuously from a value near π to $-\pi$, and I needed to add 2π to keep the curve continuous. This is called "**phase unwinding**."

You would use phase unwinding if you ever had to solve the following problem: given an **earthquake** at location (x, y), did it occur in country X? You would circumnavigate the country—compare the circle in Figure 10.18—and see if the phase angle from the earthquake to the country's boundary accumulated to 0 (yes) or to 2π (no).

The word "minimum" is used in "minimum phase" because delaying a filter can always add more phase. For example, multiplying any polynomial by Z delays it and adds ω to its phase.

For the minimum-phase filter, the group delay $d\phi/d\omega$ applied to Figure 10.19

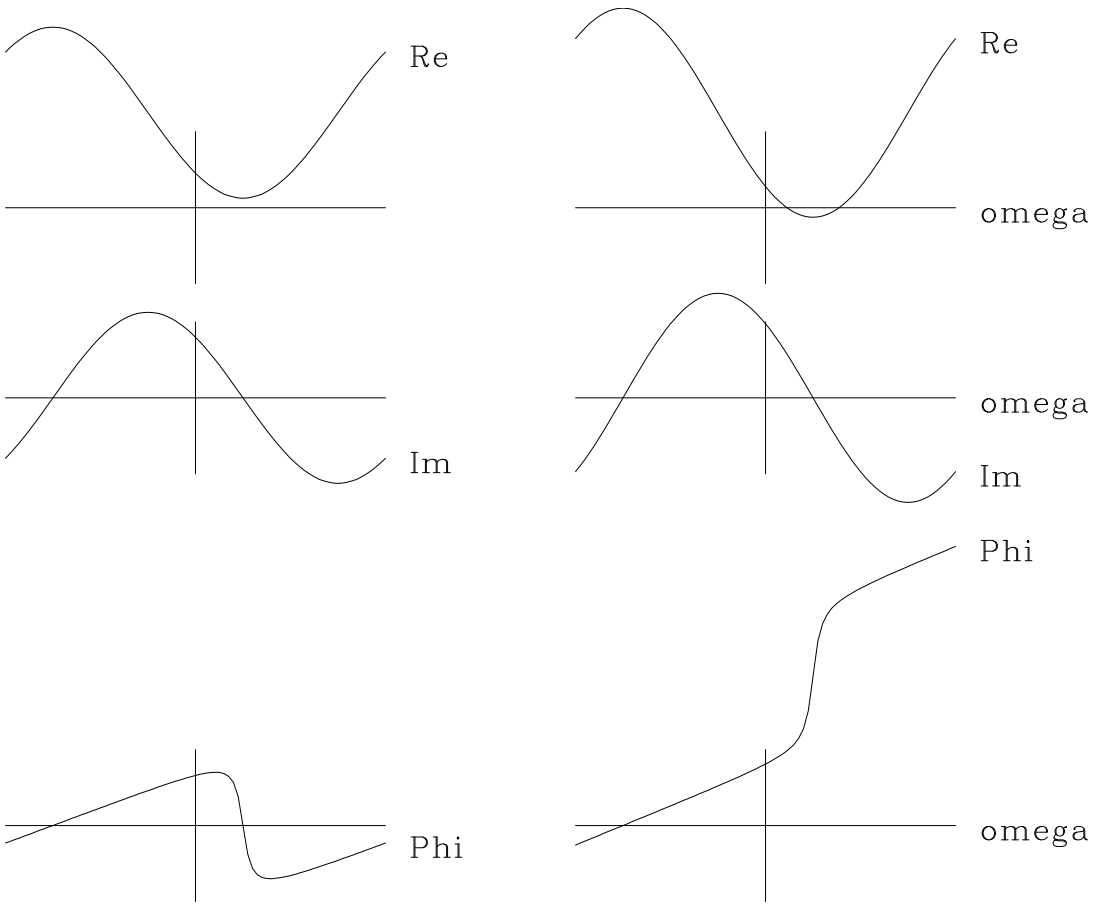


Figure 10.19: Left shows real and imaginary parts and phase angle of equation ((10.37)), for $\rho < 1$. Right, for $\rho > 1$. Left is minimum-phase and right is nonminimum-phase. spec-phase [ER]

is a periodic function of ω. For the nonminimum-phase filter, group delay happens to be a monotonically increasing function of ω. Since it is not an all-pass filter, the monotonicity is accidental.

Because **group delay** $d\phi/d\omega$ is the Fourier dual to **instantaneous frequency** $d\phi/dt$, we can now go back to Figure 10.5 and explain the discontinuous behavior of instantaneous frequency where the signal amplitude is near zero.

10.5.2. Phase of a rational filter

Now let us sum up the behavior of phase of the **rational filter**

$$B(Z) = \frac{(Z - c_1)(Z - c_2)\cdots}{(Z - a_1)(Z - a_2)\cdots} \tag{10.38}$$

By the rules of complex-number multiplication, the phase of $B(Z)$ is the sum of the phases in the numerator minus the sum of the phases in the denominator. Since we are discussing realizable filters, the denominator factors must all be minimum-phase, and so the denominator phase curve is a sum of periodic phase curves like the lower left of Figure 10.19.

The numerator factors may or may not be minimum-phase. Thus the numerator phase curve is a sum of phase curves that may resemble either type in Figure 10.19. If any factors augment phase by 2π, then the phase is not periodic, and the filter is nonminimum-phase.

10.6. ROBINSON'S ENERGY-DELAY THEOREM

Here we will see that a **minimum-phase** filter has less **energy delay** than any other one-sided filter with the same spectrum. More precisely, the energy summed from zero to any time t for the minimum-phase wavelet is greater than or equal to that of any other wavelet with the same spectrum.

Here is how I prove **Robinson**'s energy-delay theorem: compare two wavelets, F_{in} and F_{out}, that are identical except for one zero, which is outside the unit circle for F_{out} and inside for F_{in}. We can write this as

$$F_{\text{out}}(Z) = (b + s\,Z)\,F(Z) \tag{10.39}$$

$$F_{\text{in}}(Z) = (s + b\,Z)\,F(Z) \tag{10.40}$$

where b is bigger than s, and F is arbitrary but of degree n. Proving the theorem for

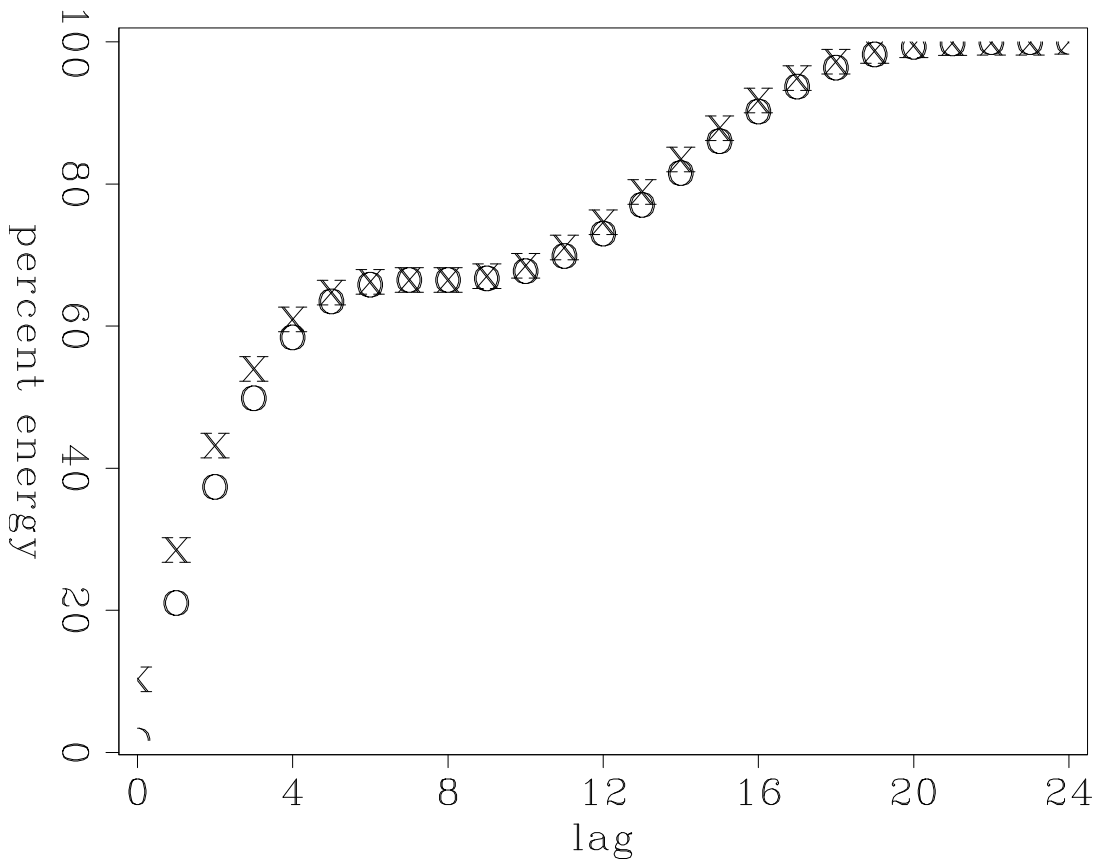
complex-valued b and s is left as an exercise. Notice that the spectrum of $b+sZ$ is the same as that of $s+bZ$. Next, tabulate the terms in question.

t	F_{out}	F_{in}	$F_{\text{out}}^2 - F_{\text{in}}^2$	$\sum_{k=0}^{t}(F_{\text{out}}^2 - F_{\text{in}}^2)$
0	bf_0	sf_0	$(b^2-s^2)\,{f_0}^2$	$(b^2-s^2)\,{f_0}^2$
1	bf_1+sf_0	sf_1+bf_0	$(b^2-s^2)({f_1}^2-{f_0}^2)$	$(b^2-s^2)\,{f_1}^2$
$\vdots$	$\vdots$			
k	bf_k+sf_{k-1}	sf_k+bf_{k-1}	$(b^2-s^2)({f_k}^2-f_{k-1}^2)$	$(b^2-s^2)\,{f_k}^2$
$\vdots$	$\vdots$			
$n+1$	sf_n	bf_n	$(b^2-s^2)(-{f_n}^2)$	0

The difference, which is given in the right-hand column, is always positive. An example of the result is shown in Figure 10.20.

Notice that $(s+bZ)/(b+sZ)$ is an all-pass filter. Multiplying by an all-pass filter does not change the amplitude spectrum but instead introduces a zero and a pole. The pole could cancel a preexisting zero, however. To sum up, multiplying by a causal/anticausal all-pass filter can move zeros inside/outside the unit circle. Each time we eliminate a zero inside the unit circle, we cause the energy of the filter to come out earlier. Eventually we run out of zeros inside the unit circle, and

Figure 10.20: p ercentage versus time. ‘x’ for minimum-phase wavelet. ‘o’ for nonminimum phase.] Total energy percentage versus time. ‘x’ for minimum-phase wavelet. ‘o’ for nonminimum phase. spec-robinson [ER]



the energy comes out as early as possible.

EXERCISES:

1 Repeat the proof of Robinson's minimum-energy-delay theorem for complex-valued b, s, and f_k. (HINT: Does $F_{\text{in}} = (s + bZ)F$ or $F_{\text{in}} = (\overline{s} + \overline{b}Z)F$?)

10.7. FILTERS IN PARALLEL

We have seen that in a **cascade of filters** the Z-transform polynomials are multiplied together. For **filters in parallel** the polynomials add. See Figure 10.21.

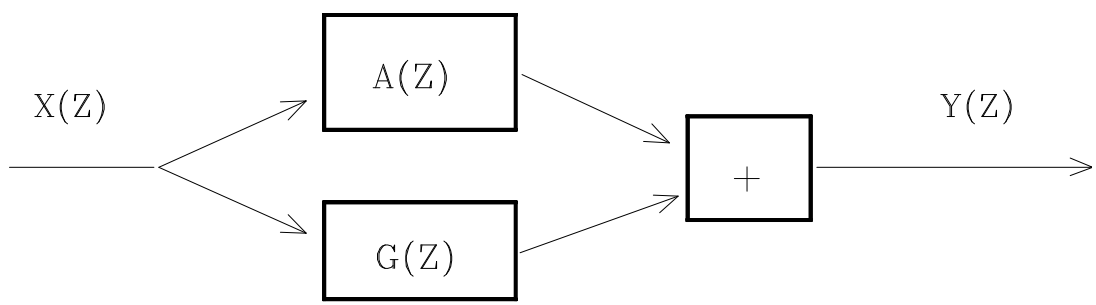


Figure 10.21: Filters operating in parallel. spec-parallel [NR]

We have seen also that a cascade of filters is minimum-phase if, and only if, each element of the product is minimum-phase. Now we will find a condition that

is sufficient (but not necessary) for a sum $A(Z)+G(Z)$ to be minimum-phase. First, assume that $A(Z)$ is minimum-phase. Then write

$$A(Z)+G(Z)=A(Z)\left(1+\frac{G(Z)}{A(Z)}\right) \tag{10.41}$$

The question as to whether $A(Z)+G(Z)$ is minimum-phase is now reduced to determining whether $A(Z)$ and $1+G(Z)/A(Z)$ are both minimum-phase. We have assumed that $A(Z)$ is minimum-phase. Before we ask whether $1+G(Z)/A(Z)$ is minimum-phase, we need to be sure that it is causal. Since $1/A(Z)$ is expandable in positive powers of Z only, then $G(Z)/A(Z)$ is also causal. We will next see that a sufficient condition for $1+G(Z)/A(Z)$ to be minimum-phase is that the spectrum of A exceed that of G at all frequencies. In other words, for any real ω, $|A|>|G|$. Thus, if we plot the curve of $G(Z)/A(Z)$ in the complex plane, for real $0\le\omega\le 2\pi$, it lies everywhere inside the unit circle. Now, if we add unity, obtaining $1+G(Z)/A(Z)$, then the curve will always have a positive real part as in Figure 10.22. Since the curve cannot enclose the origin, the phase must be that of a minimum-phase function.

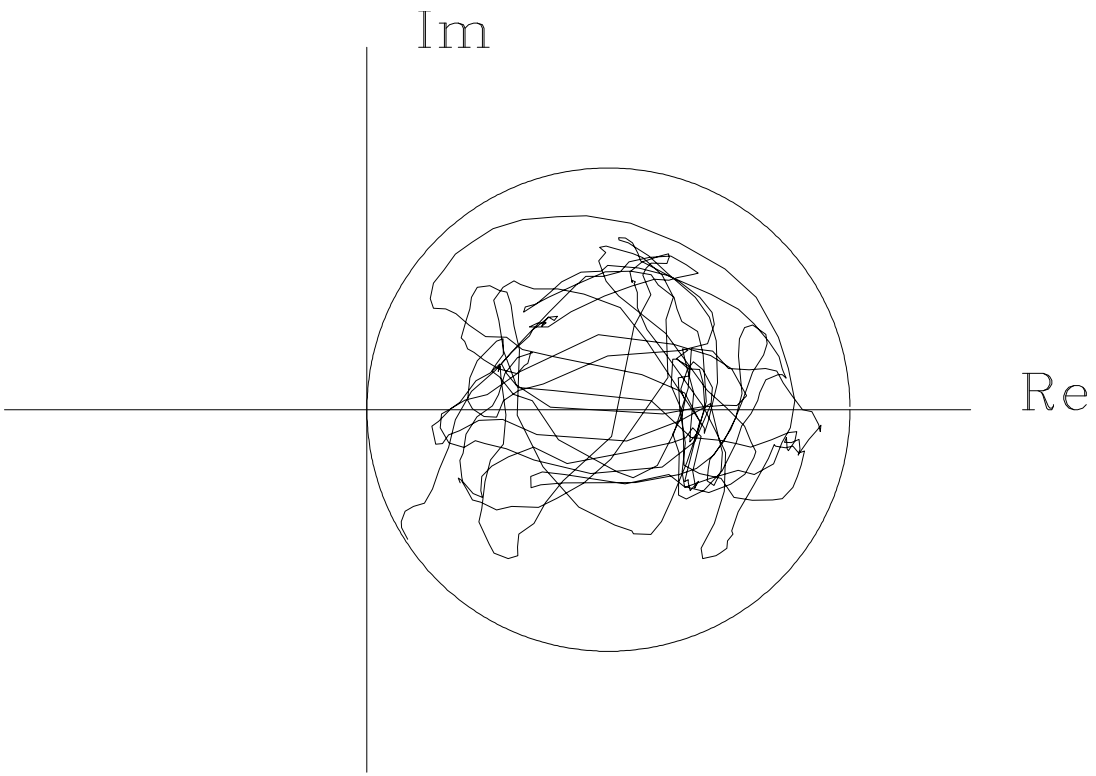


Figure 10.22: A phase trajectory as in Figure 10.18 left, but more complicated. spec-garbage [ER]

You can add garbage to a minimum-phase wavelet if you do not add too much.

This abstract theorem has an immediate physical consequence. Suppose a wave characterized by a minimum-phase $A(Z)$ is emitted from a source and detected at a receiver some time later. At a still later time, an echo bounces off a nearby object and is also detected at the receiver. The receiver sees the signal $Y(Z) = A(Z) + Z^n \alpha A(Z)$, where n measures the delay from the first arrival to the echo, and α represents the amplitude attenuation of the echo. To see that $Y(Z)$ is minimum-phase, we note that the magnitude of Z^n is unity and the reflection coefficient α must be less than unity (to avoid perpetual motion), so that $Z^n \alpha A(Z)$ takes the role of $G(Z)$. Thus, a minimum-phase wave along with its echo is minimum-phase. We will later consider wave propagation with echoes of echoes ad infinitum.

EXERCISES:

1 Find two nonminimum-phase wavelets whose sum is minimum-phase.

2 Let $A(Z)$ be a minimum-phase polynomial of degree N. Let $A'(Z) = Z^N \overline{A}(1/Z)$. Locate in the complex Z plane the roots of $A'(Z)$. $A'(Z)$ is called “maximum phase." (HINT: Work the simple case $A(Z) = a_0 + a_1 Z$ first.)

3 Suppose that $A(Z)$ is maximum-phase and that the degree of $G(Z)$ is less than or equal to the degree of $A(Z)$. Assume $|A| > |G|$. Show that $A(Z) + G(Z)$ is maximum-phase.

4 Let $A(Z)$ be minimum-phase. Where are the roots of $A(Z) + cZ^N \bar{A}(1/Z)$ in the three cases $|c| < 1, |c| > 1, |c| = 1$? (HINT: The roots of a polynomial are continuous functions of the polynomial coefficients.)

Chapter 11

Resolution and random signals

The accuracy of measurements on observed signals is limited not only by practical realities, but also by certain fundamental principles. The most famous example included in this chapter is the time-bandwidth product in Fourier-transformation

theory, called the "**uncertainty principle**."

Observed signals often look random and are often modeled by filtered random numbers. In this chapter we will see many examples of signals built from random numbers and discover how the nomenclature of statistics applies to them. Fundamentally, this chapter characterizes "**resolution**," resolution of frequency and arrival time, and the statistical resolution of signal amplitude and power as functions of time and frequency.

We will see $\sqrt{n}$ popping up everywhere. This $\sqrt{n}$ enters our discussion when we look at spectra of signals built from random numbers. Also, signals that are theoretically uncorrelated generally appear to be weakly correlated at a level of $1/\sqrt{n}$, where n is the number of independent points in the signal.

Measures of resolution (which are variously called **variance**s, **tolerance**s, uncertainties, **bandwidth**s, **duration**s, **spread**s, **rise time**s, spans, etc.) often interact with one another, so that experimental change to reduce one must necessarily increase another or some combination of the others. In this chapter we study basic cases where such conflicting interactions occur.

To avoid confusion I introduce the unusual notation Λ where Δ is commonly used. Notice that the letter Λ resembles the letter Δ, and Λ connotes length with-

out being confused with wavelength. Lengths on the time and frequency axes are defined as follows:

dt, df	mesh intervals in time and frequency
$\Delta t, \Delta f$	mesh intervals in time and frequency
$\Delta T\ \Delta F$	extent of time and frequency axis
$\Lambda T, \Lambda F$	time duration and spectral **bandwidth** of a signal

There is no mathematically tractable and universally acceptable definition for time span ΛT and spectral bandwidth ΛF. A variety of defining equations are easy to write, and many are in general use. The main idea is that the time span ΛT or the frequency span ΛF should be able to include most of the energy but need not contain it all. The time duration of a damped exponential function is infinite if by duration we mean the span of nonzero function values. However, for practical purposes the time span is generally defined as the time required for the amplitude to decay to e^{-1} of its original value. For many functions the span is defined by the span between points on the time or frequency axis where the curve (or its envelope) drops to half of the maximum value. Strange as it may sound, there are certain concepts about the behavior of ΛT and ΛF that seem appropriate for "all" mathematical choices of their definitions, yet these concepts can be proven only for

special choices.

11.1. TIME-FREQUENCY RESOLUTION

A consequence of Fourier transforms being built from $e^{i\omega t}$ is that scaling a function to be narrower in one domain scales it to be wider in the other domain. Scaling ω implies inverse scaling of t to keep the product ωt constant. For example, the FT of a rectangle is a sinc. Making the rectangle narrower broadens the sinc in proportion because ωt is constant. A pure sinusoidal wave has a clearly defined frequency, but it is spread over the infinitely long time axis. At the other extreme is an impulse function (often called a delta function), which is nicely compressed to a point on the time axis but contains a mixture of all frequencies. In this section we examine how the width of a function in one domain relates to that in the other. By the end of the section, we will formalize this into an inequality:

For any signal, the time duration ΛT and the spectral bandwidth ΛF are related by

$$\Lambda F \, \Lambda T \quad \geq \quad 1 \tag{11.1}$$

This **inequality** is the **uncertainty principle**.

Since we are unable to find a precise and convenient analysis for the definitions of ΛF and ΛT, the inequality (11.1) is not strictly true. What is important is that rough equality in (11.1) is observed for many simple functions, but for others the inequality can be extremely slack (far from equal). Strong *inequality* arises from **all-pass filter**s. An all-pass filter leaves the spectrum unchanged, and hence ΛF unchanged, but it can spread out the signal arbitrarily, increasing ΛT arbitrarily. Thus the time-bandwidth maximum is unbounded for all-pass filters. Some people say that the **Gaussian** function has the minimum product in (11.1), but that really depends on a particular method of measuring ΛF and ΛT.

11.1.1. A misinterpretation of the uncertainty principle

It is easy to misunderstand the uncertainty principle. An oversimplification of it is to say that it is "impossible to know the frequency at any particular time." This oversimplification leads us to think about a truncated sinusoid, such as in Figure 11.1. We know the frequency exactly, so ΛF seems zero, whereas ΛT is finite, and this seems to violate (11.1). But what the figure shows is that the truncation of the sinusoid has broadened the frequency band. More particularly, the impulse function in the frequency domain has been convolved by the sinc function that is the Fourier transform of the truncating rectangle function.

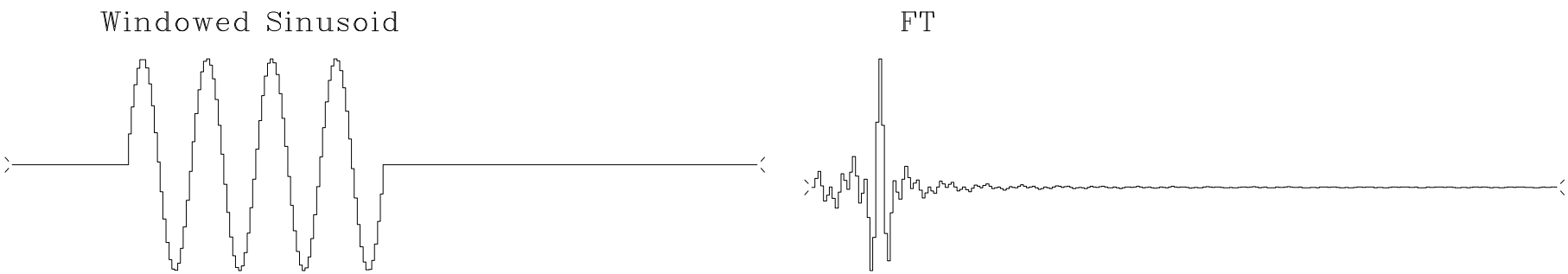


Figure 11.1: Windowed sinusoid and its Fourier transform. rand-windcos [NR]

11.1.2. Measuring the time-bandwidth product

Now examine Figure 11.2, which contains sampled Gaussian functions and their Fourier transforms. The Fourier transform of a Gaussian is well known to be another Gaussian function, as the plot confirms. I adjusted the width of each Gaussian so that the widths would be about equal in both domains. The Gaussians were sampled at various values of n, increasing in steps by a factor of 4. You can measure the width dropping by a factor of 2 at each step. For those of you who have already learned about the uncertainty principle, it may seem paradoxical that the function's width is dropping in both time and frequency domains.

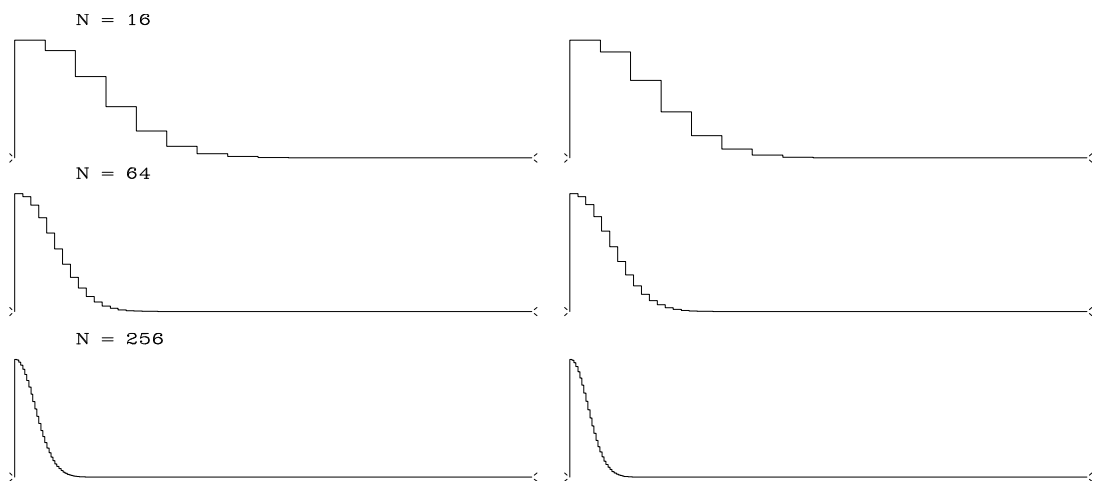


Figure 11.2: Sampled Gaussian functions and their Fourier transforms for vectors of length n = 16, 64, and 256. rand-uncertain [NR]

The resolution of the paradox is that the physical length of the time axis or the frequency axis is varying as we change n (even though the plot length is scaled to a constant on the page). We need to associate a physical mesh with the computational mesh. A method of associating physical and computational meshes was described in chapter 9 on page **??**. In real physical space as well as in Fourier transform space, the object remains a constant size as the mesh is refined.

Let us read from Figure 11.2 values for the widths ΛF and ΛT. On the top row, where $N = 16$, I pick a width of about 4 points, and this seems to include about 90% of the area under the function. For this signal (with the widths roughly equal in both domains) it seems that $\Lambda T = \sqrt{N}dt$ and $\Lambda F = \sqrt{N}df$. Using the relation between dt and df found in equation (**??**), which says that $dt\,df = 1/N$, the product becomes $\Lambda T\,\Lambda F = 1$.

We could also confirm the inequality (11.1) by considering simple functions for which we know the analytic transforms—for example, an impulse function in time. Then $\Lambda T = dt$, and the Fourier transform occupies the entire frequency band from minus to plus the Nyquist frequency $\pm.5/dt$ Hz, i.e., $\Lambda F = 1/dt$. Thus again, the product is $\Lambda T\,\Lambda F = 1$.

11.1.3. The uncertainty principle in physics

The inequality (11.1) derives the name "**uncertainty principle**" from its interpretation in **quantum mechanics**. Observations of subatomic particles show they behave like waves with spatial frequency proportional to particle momentum. The classical laws of mechanics enable prediction of the future of a mechanical system by extrapolation from the currently known position and momentum. But because of the wave nature of matter, with momentum proportional to spatial frequency, such prediction requires simultaneous knowledge of both the location and the spatial frequency of the wave. This is impossible, as we see from (11.1); hence the word "uncertainty."

11.1.4. Gabor's proof of the uncertainty principle

Although it is easy to verify the uncertainty principle in many special cases, it is not easy to deduce it. The difficulty begins from finding a definition of the width of a function that leads to a tractable analysis. One possible definition uses a second moment; that is, ΛT is defined by

$$(\Lambda T)^2 \quad = \quad \frac{\int t^2 b(t)^2 \, dt}{\int b(t)^2 \, dt} \tag{11.2}$$

The spectral bandwidth ΛF is defined likewise. With these definitions, Dennis **Gabor** prepared a widely reproduced proof. I will omit his proof here; it is not an easy proof; it is widely available; and the definition (11.2) seems inappropriate for a function we often use, the **sinc** function, i.e., the FT of a step function. Since the sinc function drops off as t^{-1}, its width ΛT defined with (11.2) is infinity, which is unlike the more human measure of width, the distance to the first axis crossing.

11.1.5. My rise-time proof of the uncertainty principle

In FGDP I came up with a proof of the uncertainty principle that is appropriate for causal functions. That proof is included directly below, but I recommend that the beginning reader skip over it, as it is somewhat lengthy. I include it because this book is oriented toward causal functions, the proof is not well known, and I have improved it since FGDP.

A similar and possibly more basic concept than the product of time and frequency spreads is the relationship between spectral **bandwidth** and the "**rise time**" of a system-response function. The rise time ΛT of a system response is defined as follows: when we kick a physical system with an impulse function, it usually

responds rapidly, rising to some maximum level, and then dropping off more slowly toward zero. The quantitative value of the rise time is generally, and somewhat arbitrarily, taken to be the span between the time of excitation and the time at which the system response is more than halfway up to its maximum.

"Tightness" (nearness to equality) in the inequality (11.1) is associated with minimum phase. "Slackness" (remoteness from equality) in the (11.1) would occur if a filter with an additional all-pass component were used. Slackness could also be caused by a decay time that is more rapid than the rise time, or by other combinations of rises and falls, such as random combinations. Minimum-phase systems generally respond rapidly compared to the rate at which they later decay. Focusing our attention on such systems, we can now seek to derive the inequality (11.1) applied to rise time and bandwidth.

The first step is to choose a definition for rise time. I have found a tractable definition of **rise time** to be

$$\frac{1}{\Lambda T} \quad = \quad \frac{\int_0^\infty \frac{1}{t}\, b(t)^2\, dt}{\int_0^\infty b(t)^2\, dt} \tag{11.3}$$

where $b(t)$ is the response function under consideration. Equation (11.3) defines

ΛT by the first negative moment. Since this is unfamiliar, consider two examples. Taking $b(t)$ to be a step function, recognize that the numerator integral diverges, giving the desired $\Lambda T = 0$ rise time. As a further example, take $b(t)^2$ to grow linearly from zero to t_0 and then vanish. Then the rise time ΛT is $t_0/2$, again reasonable. It is curious that $b(t)$ could grow as $\sqrt{t}$, which rises with infinite slope at $t = 0$, and not cause ΛT to be pushed to zero.

- **Proof by way of the dual problem**

Although the Z-transform method is a great aid in studying cases where divergence (as $1/t$) plays a role, it has the disadvantage that it destroys the formal interchangeability between the time domain and the frequency domain. To take advantage of the analytic simplicity of the Z-transform, we consider instead the dual to the rise-time problem. Instead of a signal whose square vanishes at negative time, we have a spectrum $\overline{B}(1/Z)B(Z)$ that vanishes at negative frequencies. We measure how fast this spectrum can rise after $\omega = 0$. We will find this time interval to be related to the time duration ΛT of the complex-valued signal b_t. More precisely, we now define the lowest significant frequency component ΛF in the spectrum, analogously

to (11.3), as

$$\frac{1}{\Lambda F} \quad = \quad \int_{-\infty}^{\infty} \frac{1}{f}\, \overline{B} B\, df \quad = \quad \int_{-\infty}^{\infty} \overline{B} B\, \frac{d\omega}{\omega} \tag{11.4}$$

where we have assumed the spectrum is normalized, i.e., the zero lag of the autocorrelation of b_t is unity. Now recall the **bilinear transform**, equation (**??**), which represents $1/(-i\omega)$ as the coefficients of $\frac{1}{2}(1+Z)/(1-Z)$, namely, $(\ldots 0, 0, 0, \frac{1}{2}, 1, 1, 1 \ldots$ The pole right on the unit circle at $Z = 1$ causes some nonuniqueness. Because $1/i\omega$ is an imaginary, odd, frequency function, we will want an odd expression (such as on page **??**) to insert into (11.4):

$$\frac{1}{-i\omega} \quad = \quad \frac{(\cdots - Z^{-2} - Z^{-1} + 0 + Z + Z^2 + \cdots)}{2} \tag{11.5}$$

Using limits on the integrals for time-sampled functions and inserting (11.5) into (11.4) gives

$$\frac{1}{\Lambda F} \quad = \quad \frac{-i}{2\pi} \int_{-\pi}^{+\pi} \frac{1}{2}(\cdots - Z^{-2} - Z^{-1} + Z + Z^2 + \cdots)\, \overline{B}\left(\frac{1}{Z}\right) B(Z)\, d\omega \tag{11.6}$$

Let s_t be the autocorrelation of b_t. Since any integral around the unit circle of a Z-transform polynomial selects the coefficient of Z^0 of its integrand, we have

$$\frac{1}{\Lambda F} = \frac{-i}{2}\left[(s_{-1}-s_1)+(s_{-2}-s_2)+(s_{-3}-s_3)+\cdots\right] = \sum_{t=1}^{\infty} -\Im s_t \quad (11.7)$$

$$\frac{1}{\Lambda F} = \sum_{t=1}^{\infty} -\Im s_t \leq \sum_{t=1}^{\infty} |s_t| \quad (11.8)$$

The height of the autocorrelation has been normalized to $s_0 = 1$. The sum in (11.8) is an integral representing area under the $|s_t|$ function. So the area is a measure of the **autocorrelation** width ΛT_{auto}. Thus,

$$\frac{1}{\Lambda F} \leq \sum_{t=1}^{\infty} |s_t| = \Lambda T_{\text{auto}} \quad (11.9)$$

Finally, we must relate the duration of a signal ΛT to the duration of its autocorrelation ΛT_{auto}. Generally speaking, it is easy to find a long signal that has short autocorrelation. Just take an arbitrary short signal and convolve it using a lengthy

all-pass filter. Conversely, we cannot get a long autocorrelation function from a short signal. A good example is the autocorrelation of a rectangle function, which is a triangle. The triangle appears to be twice as long, but considering that the triangle tapers down, it is reasonable to assert that the ΛT's are the same. Thus, we conclude that

$$\Lambda T_{\text{auto}} \quad \leq \quad \Lambda T \tag{11.10}$$

Inserting this inequality into (11.9), we have the uncertainty relation

$$\Lambda T \; \Lambda F \quad \geq \quad 1 \tag{11.11}$$

Looking back over the proof, I feel that the basic time-bandwidth idea is in the *equality* (11.7). I regret that the verbalization of this idea, boxed following, is not especially enlightening. The *inequality* arises from $\Lambda T_{\text{auto}} < \Lambda T$, which is a simple idea.

> The inverse moment of the normalized spectrum of an analytic signal equals the imaginary part of the mean of its autocorrelation.

EXERCISES:

1 Consider $B(Z) = [1-(Z/Z_0)^n]/(1-Z/Z_0)$ as Z_0 goes to the unit circle. Sketch the signal and its squared amplitude. Sketch the frequency function and its squared amplitude. Choose ΛF and ΛT.

2 A time series made up of two frequencies can be written as

$$b_t = A\cos\omega_1 t + B\sin\omega_1 t + C\cos\omega_2 t + D\sin\omega_2 t$$

Given ω_1, ω_2, b_0, b_1, b_2, and b_3, show how to calculate the amplitude and phase angles of the two sinusoidal components.

11.2. FT OF RANDOM NUMBERS

Many real signals are complicated and barely comprehensible. In experimental work, we commonly transform such data. To better understand what this means, it will be worthwhile to examine signals made from **random** numbers.

Figure 11.3 shows discrete Fourier transforms of random numbers. The basic conclusion to be drawn from this figure is that transforms of random numbers look

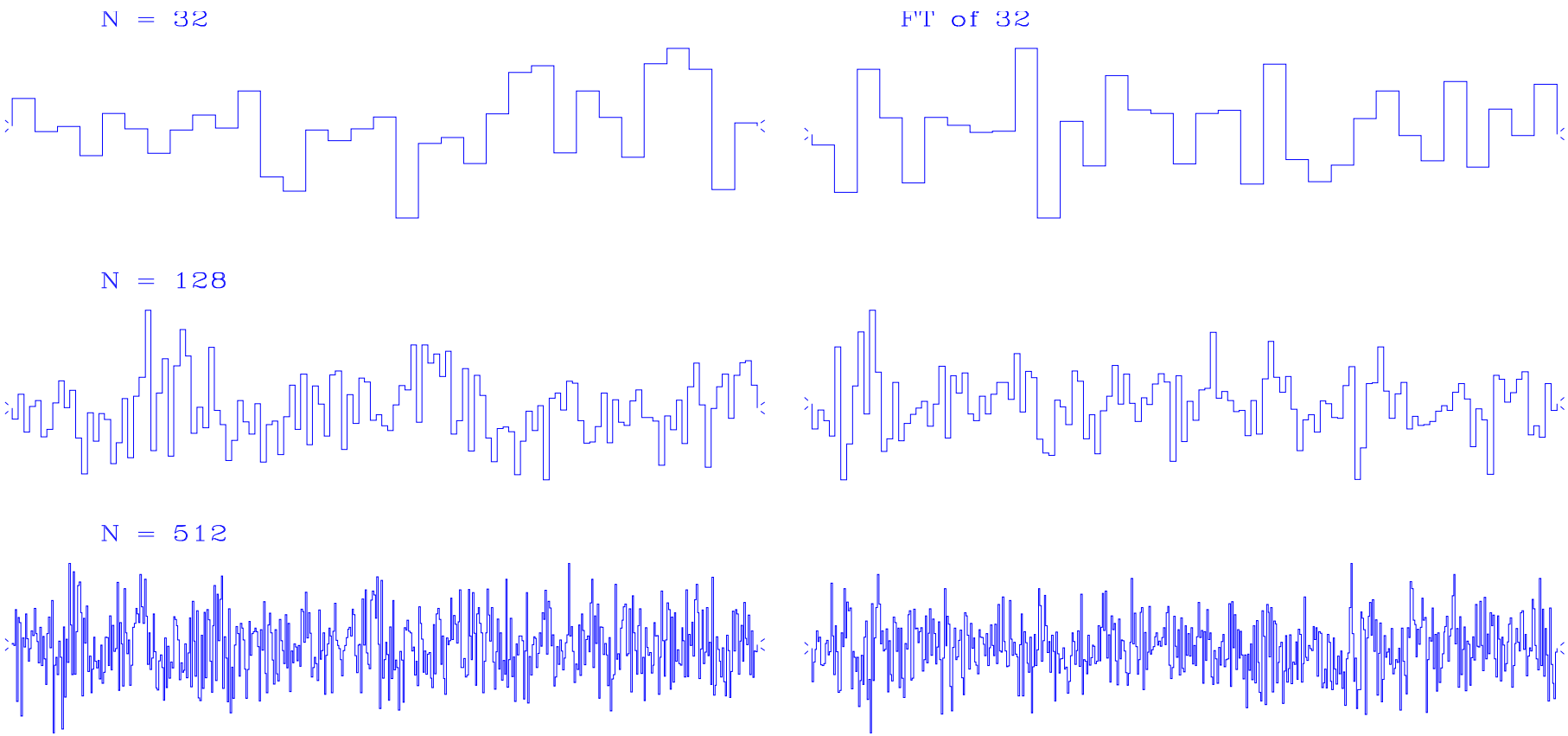


Figure 11.3: Fourier cosine transforms of vectors containing random numbers. N is the number of components in the vector. rand-nrand [NR]

like more random numbers. A random series containing all frequencies is called a “white-noise" series, because the color white is made from roughly equal amounts of all colors. Any series made by independently chosen random numbers is said to be an “independent" series. An independent series must be white, but a white series need not be independent. Figure 11.4 shows Fourier transforms of random numbers surrounded by zeros (or zero padded). Since all the vectors of random numbers are

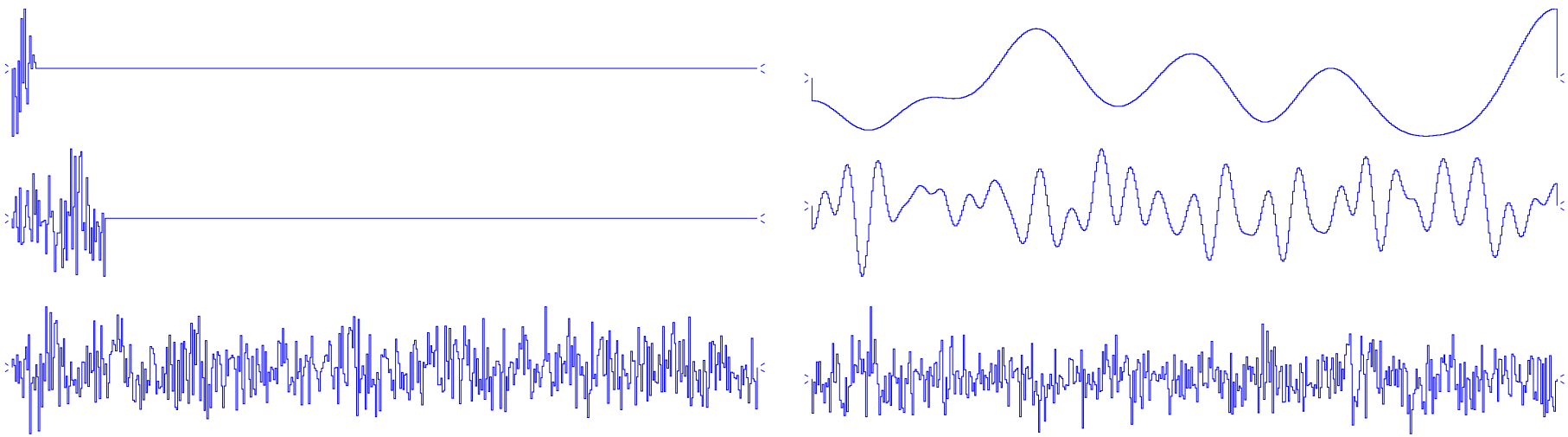


Figure 11.4: Zero-padded random numbers and their FTs. rand-pad [NR]

the same length (each has 1024 points, including both sides of the even function with the even part (513 points) shown), the transforms are also the same length. The top signal has less randomness than the second trace (16 random numbers versus 64). Thus the top FT is smoother than the lower ones. Although I introduced this figure as if the left panel were the time domain and the right panel were frequency, you are free to think of it the opposite way. This is more clear. With the left-hand signal being a frequency function, where higher frequencies are present, the right-hand signal oscillates faster.

11.2.1. Bandlimited noise

Figure 11.5 shows bursts of 25 random numbers at various shifts, and their Fourier transforms. You can think of either side of the figure as the time domain and the other side as the frequency domain. (See page **??** for a description of the different ways of interpreting plots of one side of Fourier-transform pairs of even functions.) I like to think of the left side as the Fourier domain and the right side as the signals. Then the signals seem to be sinusoids of a constant frequency (called the “center" frequency) and of an amplitude that is modulated at a slower rate (called the “**beat**”

frequency). Observe that the center frequency is related to the *location* of the random bursts, and that the beat frequency is related to the *bandwidth* of the noise burst.

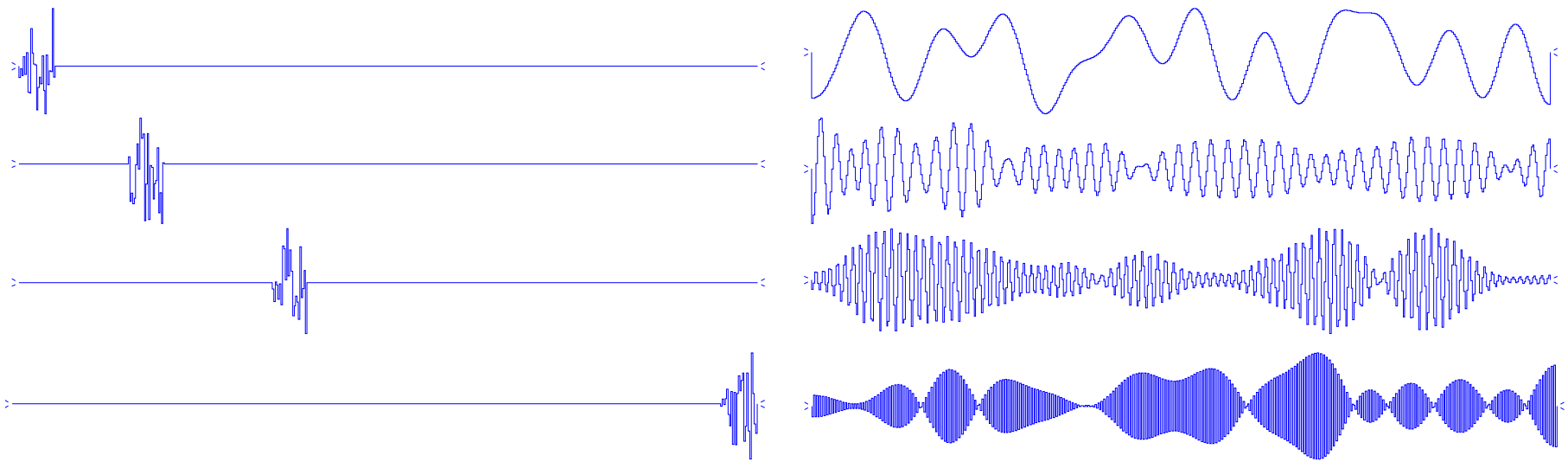


Figure 11.5: Shifted, zero-padded random numbers in bursts of 25 numbers. rand-shift [NR]

You can also think of Figure 11.5 as having one-sided frequency functions on the left, and the right side as being the *real part* of the signal. The real parts are

cosinelike, whereas the imaginary parts (not shown) are sinelike and have the same envelope function as the cosinelike part.

You might have noticed that the bottom plot in Figure 11.5, which has Nyquist-frequency modulated beats, seems to have about twice as many beats as the two plots above it. This can be explained as an end effect. The noise burst near the Nyquist frequency is really twice as wide as shown, because it is mirrored about the Nyquist frequency into negative frequencies. Likewise, the top figure is not modulated at all, but the signal itself has a frequency that matches the beats on the bottom figure.

11.3. TIME-STATISTICAL RESOLUTION

[1] If we flipped a fair coin 1000 times, it is unlikely that we would get exactly 500 heads and 500 tails. More likely the number of heads would lie somewhere between

---

[1] I would like to thank Gilles **Darche** for carefully reading this chapter and pointing out some erroneous assertions in FGDP. If there are any mistakes in the text now, I probably introduced them after his reading.

400 and 600. Or would it lie in another range? The theoretical value, called the "**mean**" or the "**expectation**," is 500. The value from our experiment in actually flipping a fair coin is called the "**sample mean**." How much difference Λm should we expect between the sample mean $\hat{m}$ and the true mean m? Both the coin flips x and our sample mean $\hat{m}$ are *random variables*. Our 1000-flip experiment could be repeated many times and would typically give a different $\hat{m}$ each time. This concept will be formalized in section 11.3.5. as the "**variance of the sample mean**," which is the expected squared difference between the true mean and the mean of our sample.

The problem of estimating the **statistic**al parameters of a time series, such as its mean, also appears in seismic processing. Effectively, we deal with seismic traces of finite duration, extracted from infinite sequences whose parameters can only be estimated from the finite set of values available in these seismic traces. Since the knowledge of these parameters, such as signal-to-noise ratio, can play an important role during the processing, it can be useful not only to estimate them, but also to have an idea of the error made in this estimation.

11.3.1. Ensemble

The "true value" of the mean could be defined as the mean that results when the coin is flipped n times, when n is conceived of as going to infinity. A more convenient definition of true value is that the experiment is imagined as having been done separately under identical conditions by an infinite number of people (an "**ensemble**"). The ensemble may seem a strange construction; nonetheless, much literature in statistics and the natural sciences uses the ensemble idea. Let us say that the ensemble is defined by a probability as a function of time. Then the ensemble idea enables us to define a time-variable mean (the sum of the values found by the ensemble weighted by the probabilities) for, for example, coins that change with time.

11.3.2. Expectation and variance

A conceptual average over the ensemble, or **expectation**, is denoted by the symbol E. The index for summation over the ensemble is never shown explicitly; every random variable is presumed to have one. Thus, the true mean at time t is defined as $m_x(t) = \mathrm{E}(x_t)$. The mean can vary with time:

$$m_x(t) \quad = \quad \mathrm{E}[x(t)] \tag{11.12}$$

The "**variance**" σ^2 is defined to be the power after the mean is removed, i.e.,

$$\sigma_x(t)^2 \quad = \quad \mathrm{E}[(x(t) - m_x(t))^2] \tag{11.13}$$

(Conventionally, σ^2 is referred to as the variance, and σ is called the "**standard deviation**.")

For notational convenience, it is customary to write $m(t)$, $\sigma(t)$, and $x(t)$ simply as m, σ, and x_t, using the verbal context to specify whether m and σ are time-variable or constant. For example, the standard deviation of the seismic amplitudes on a seismic trace before correction of spherical divergence decreases with time, since these amplitudes are expected to be "globally" smaller as time goes on.

When manipulating algebraic expressions, remember that the symbol E behaves like a summation sign, namely,

$$\mathrm{E} \quad \equiv \quad (\lim N \to \infty) \quad \frac{1}{N} \sum_{1}^{N} \tag{11.14}$$

Note that the summation index is not given, since the sum is over the ensemble, not time. To get some practice with the expectation symbol E, we can reduce equa-

tion (11.13):

$$\sigma_x^2 \quad = \quad \mathrm{E}[(x_t - m_x)^2] \quad = \quad \mathrm{E}(x_t^2) \; - \; 2m_x\mathrm{E}(x_t) + m_x^2 \quad = \quad \mathrm{E}(x_t^2) \; - \; m_x^2 \tag{11.15}$$

Equation (11.15) says that the energy is the variance plus the squared mean.

11.3.3. Probability and independence

A random variable x can be described by a **probability** $p(x)$ that the amplitude x will be drawn. In real life we almost never know the probability function, but theoretically, if we do know it, we can compute the **mean** value using

$$m \quad = \quad E(x) \quad = \quad \int x\, p(x)\, dx \tag{11.16}$$

"**Statistical independence**" is a property of two or more random numbers. It means the samples are drawn independently, so they are unrelated to each other. In terms of probability functions, the independence of random variables x and y is expressed by

$$p(x,y) \quad = \quad p(x)\, p(y) \tag{11.17}$$

From these, it is easy to show that

$$\mathrm{E}(xy) \quad = \quad \mathrm{E}(x)\mathrm{E}(y) \tag{11.18}$$

11.3.4. Sample mean

Now let x_t be a time series made up of identically distributed random numbers: m_x and σ_x do not depend on time. Let us also suppose that they are *independently* chosen; this means in particular that for any different times t and s ($t \neq s$):

$$\mathrm{E}(x_t x_s) \quad = \quad \mathrm{E}(x_t)\mathrm{E}(x_s) \tag{11.19}$$

Suppose we have a sample of n points of x_t and are trying to determine the value of m_x. We could make an estimate $\hat{m}_x$ of the mean m_x with the formula

$$\hat{m}_x \quad = \quad \frac{1}{n}\sum_{t=1}^{n} x_t \tag{11.20}$$

A somewhat more elaborate method of estimating the mean would be to take a weighted average. Let w_t define a set of weights normalized so that

$$\sum w_t \quad = \quad 1 \tag{11.21}$$

With these weights, the more elaborate estimate $\hat{m}$ of the mean is

$$\hat{m}_x \quad = \quad \sum w_t \, x_t \tag{11.22}$$

Actually (11.20) is just a special case of (11.22); in (11.20) the weights are $w_t = 1/n$.

Further, the weights could be *convolved* on the random time series, to compute *local* averages of this time series, thus smoothing it. The weights are simply a filter response where the filter coefficients happen to be positive and cluster together. Figure 11.6 shows an example: a random walk function with itself smoothed locally.

11.3.5. Variance of the sample mean

Our objective here is to calculate how far the estimated mean $\hat{m}$ is likely to be from the true mean m for a sample of length n. This difference is the **variance of the sample mean** and is given by $(\Lambda m)^2 = \sigma^2_{\hat{m}}$, where

$$\sigma^2_{\hat{m}} \quad = \quad \mathrm{E}\,[(\hat{m} - m)^2] \tag{11.23}$$

$$\quad = \quad \mathrm{E}\left\{[(\sum w_t x_t) - m]^2\right\} \tag{11.24}$$

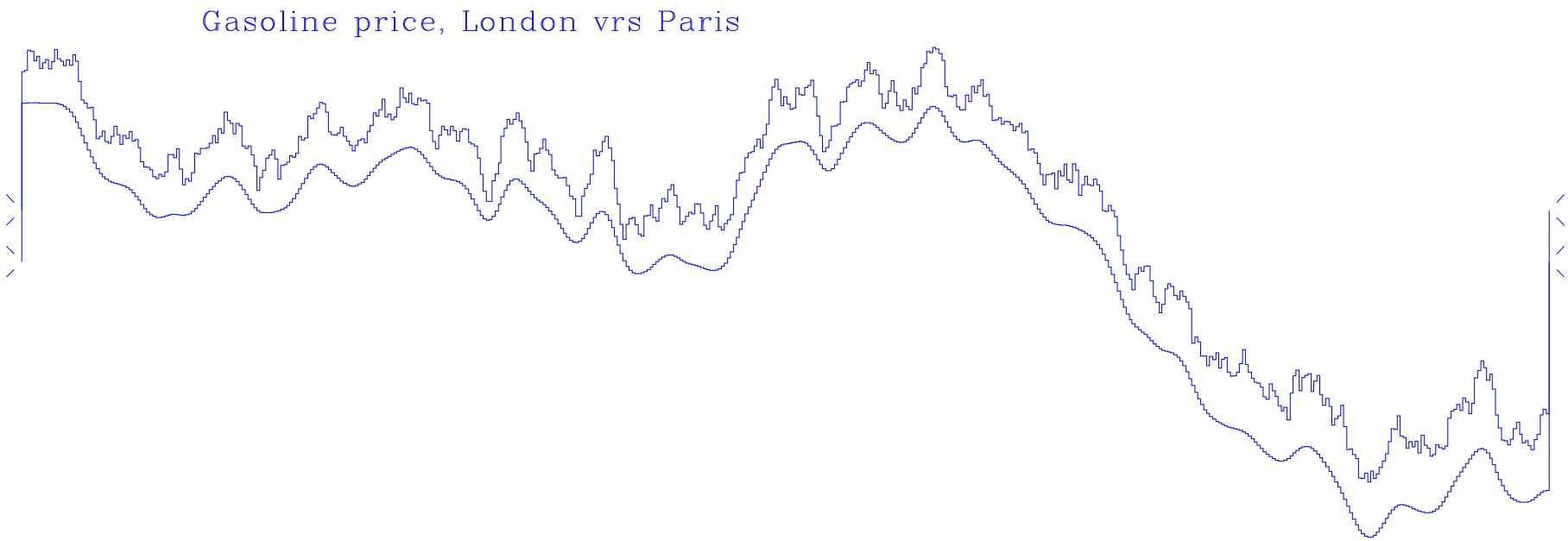


Figure 11.6: Random walk and itself smoothed (and shifted downward). rand-walk [NR]

Now use the fact that $m = m\sum w_t = \sum w_t m$:

$$\sigma^2_{\hat{m}} = \mathrm{E}\left\{\left[\sum_t w_t(x_t - m)\right]^2\right\} \tag{11.25}$$

$$= \mathrm{E}\left\{\left[\sum_t w_t(x_t - m)\right]\left[\sum_s w_s(x_s - m)\right]\right\} \tag{11.26}$$

$$= \mathrm{E}\left[\sum_t\sum_s w_t w_s(x_t - m)(x_s - m)\right] \tag{11.27}$$

The step from (11.26) to (11.27) follows because

$$(a_1+a_2+a_3)(a_1+a_2+a_3) = \text{sum of } \begin{bmatrix} a_1a_1 & a_1a_2 & a_1a_3 \\ a_2a_1 & a_2a_2 & a_2a_3 \\ a_3a_1 & a_3a_2 & a_3a_3 \end{bmatrix} \tag{11.28}$$

The expectation symbol E can be regarded as another summation, which can be done after, as well as before, the sums on t and s, so

$$\sigma^2_{\hat{m}} = \sum_t \sum_s w_t \, w_s \, \mathrm{E}\left[(x_t - m)(x_s - m)\right] \tag{11.29}$$

If $t \neq s$, since x_t and x_s are independent of each other, the expectation $\mathrm{E}[(x_t - m)(x_s - m)]$ will vanish. If $s = t$, then the expectation is the variance defined by (11.13). Expressing the result in terms of the Kronecker delta, δ_{ts} (which equals unity if $t = s$, and vanishes otherwise) gives

$$\sigma^2_{\hat{m}} = \sum_t \sum_s w_t \, w_s \, \sigma_x^2 \delta_{ts} \tag{11.30}$$

$$\sigma^2_{\hat{m}} = \sum_t w_t^2 \, \sigma_x^2 \tag{11.31}$$

$$\sigma_{\hat{m}} = \sigma_x \sqrt{\sum_t w_t^2} \tag{11.32}$$

For n weights, each of size $1/n$, the standard deviation of the sample mean is

$$\Lambda m_x \quad = \quad \sigma_{\hat{m}_x} \quad = \quad \sigma_x \sqrt{\sum_{t=1}^{n} \left(\frac{1}{n}\right)^2} \quad = \quad \frac{\sigma_x}{\sqrt{n}} \qquad (11.33)$$

This is the most important property of random numbers that is not intuitively obvious. Informally, the result (11.33) says this: given a sum y of terms with random polarity, whose theoretical mean is zero, then

$$y \quad = \quad \underbrace{\pm 1 \pm 1 \pm 1 \cdots}_{n \text{ terms}} \qquad (11.34)$$

The sum y is a random variable whose standard deviation is $\sigma_y = \sqrt{n} = \Lambda y$. An experimenter who does not know the mean is zero will report that the mean of y is $\mathrm{E}(y) = \hat{y} \pm \Lambda y$, where $\hat{y}$ is the experimental value.

If we are trying to estimate the mean of a random series that has a time-variable mean, then we face a basic dilemma. Including many numbers in the sum in order to make Λm small conflicts with the possibility of seeing m_t change during the

measurement.

The "**variance of the sample variance**" arises in many contexts. Suppose we want to measure the storminess of the ocean. We measure water level as a function of time and subtract the mean. The storminess is the variance about the mean. We measure the storminess in one minute and call it a sample storminess. We compare it to other minutes and other locations and we find that they are not all the same. To characterize these differences, we need the *variance of the sample variance* $\sigma^2_{\hat{\sigma}^2}$. Some of these quantities can be computed theoretically, but the computations become very cluttered and dependent on assumptions that may not be valid in practice, such as that the random variables are independently drawn and that they have a Gaussian probability function. Since we have such powerful computers, we might be better off ignoring the theory and remembering the basic principle that a function of random numbers is also a random number. We can use simulation to estimate the function's mean and variance. Basically we are always faced with the same dilemma: if we want to have an accurate estimation of the variance, we need a large number of samples, which limits the possibility of measuring a time-varying variance.

EXERCISES:

1 Suppose the mean of a sample of random numbers is estimated by a triangle weighting function, i.e.,

$$\hat{m} \quad = \quad s\sum_{i=0}^{n}(n-i)\,x_i$$

Find the scale factor s so that $\mathrm{E}(\hat{m}) = m$. Calculate Λm. Define a reasonable ΛT. Examine the uncertainty relation.

2 A random series x_t with a possibly time-variable mean may have the mean estimated by the feedback equation

$$\hat{m}_t \quad = \quad (1-\epsilon)\hat{m}_{t-1} + bx_t$$

a. Express $\hat{m}_t$ as a function of $x_t, x_{t-1}, \ldots$, and not $\hat{m}_{t-1}$.

b. What is ΛT, the effective averaging time?

c. Find the scale factor b so that if $m_t = m$, then $\mathrm{E}(\hat{m}_t) = m$.

d. Compute the random error $\Lambda m = \sqrt{\mathrm{E}(\hat{m} - m)^2}$. (HINT: Λm goes to $\sigma\sqrt{\epsilon/2}$ as $\epsilon \to 0$.)

e. What is $(\Lambda m)^2 \Lambda T$ in this case?

11.4. SPECTRAL FLUCTUATIONS

Recall the basic model of time-series analysis, namely, random numbers passing through a filter. A sample of input, filter, and output amplitude spectra is shown in Figure 11.7. From the spectrum of the output we can guess the spectrum of the filter, but the figure shows there are some limitations in our ability to do so. Let us analyze this formally.

Observations of sea level over a long period of time can be summarized in terms of a few statistical averages, such as the mean height m and the variance σ^2. Another important kind of statistical average for use on geophysical time series is the "**power spectrum**." Many mathematical models explain only statistical averages of data and not the data itself. To recognize certain **pitfall**s and understand certain fundamental limitations on work with power spectra, we first consider the idealized example of random numbers.

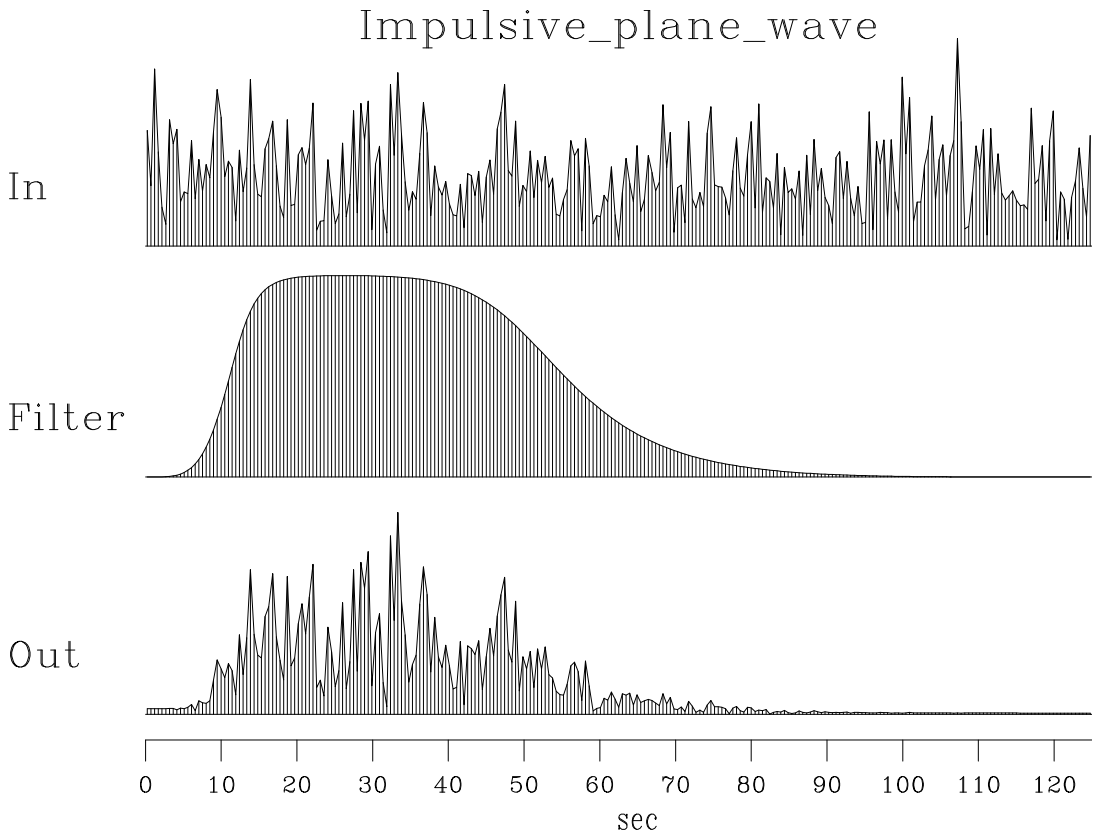


Figure 11.7: Random numbers into a filter. Top is a spectrum of random numbers. Middle is the spectrum of a filter. Bottom is the spectrum of filter output. rand-filter [ER]

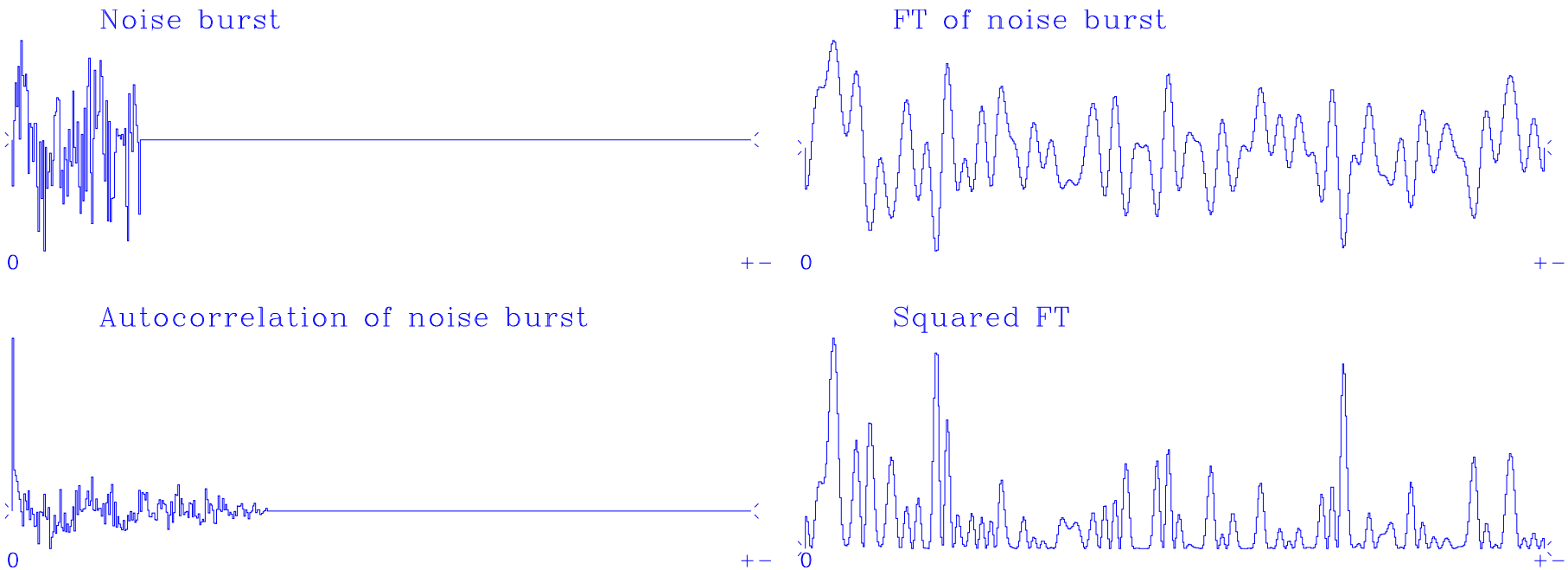


Figure 11.8: Autocorrelation and spectra of random numbers. rand-auto [NR]

Figure 11.8 shows a signal that is a burst of noise; its Fourier transform, and the transform squared; and its inverse transform, the autocorrelation. Here the FT squared is the same as the more usual FT times its complex conjugate—because the noise-burst signal is even, its FT is real.

Notice that the **autocorrelation** has a big spike at zero lag. This spike represents the correlation of the random numbers with themselves. The other lags are much smaller. They represent the correlation of the noise burst with itself shifted. Theoretically, the noise burst is *not* correlated with itself shifted: these small fluctuations result from the finite extent of the noise sample.

Imagine many copies of Figure 11.8. Ensemble averaging would amount to adding these other autocorrelations or, equivalently, adding these other spectra. The fluctuations aside the central lobe of the autocorrelation would be destroyed by ensemble averaging, and the fluctuations in the spectrum would be smoothed out. The **expectation of the autocorrelation** is that it is an impulse at zero lag. The **expectation of the spectrum** is that it is a constant, namely,

$$\mathrm{E}[\hat{S}(Z)] \quad = \quad S(Z) \quad = \quad \text{const} \tag{11.35}$$

11.4.1. Paradox: large n vs. the ensemble average

Now for the paradox. Imagine $n \to \infty$ in Figure 11.8. Will we see the same limit as results from the ensemble average? Here are two contradictory points of view:

- For increasing n, the fluctuations on the nonzero autocorrelation lags get smaller, so the autocorrelation should tend to an impulse function. Its Fourier transform, the spectrum, should tend to a constant.
- On the other hand, for increasing n, as in Figure 11.3, the spectrum does not get any smoother, because the FTs should still look like random noise.

We will see that the first idea contains a false assumption. The autocorrelation does tend to an impulse, but the fuzz around the sides cannot be ignored—although the fuzz tends to zero amplitude, it also tends to infinite extent, and the product of zero with infinity here tends to have the same energy as the central impulse.

To examine this issue further, let us discover how these autocorrelations decrease to zero with n (the number of samples). Figure 11.9 shows the autocorrelation samples as a function of n in steps of n increasing by factors of four. Thus $\sqrt{n}$ increases by factors of two. Each autocorrelation in the figure was normalized at zero lag. We see the sample variance for nonzero lags of the autocorrelation

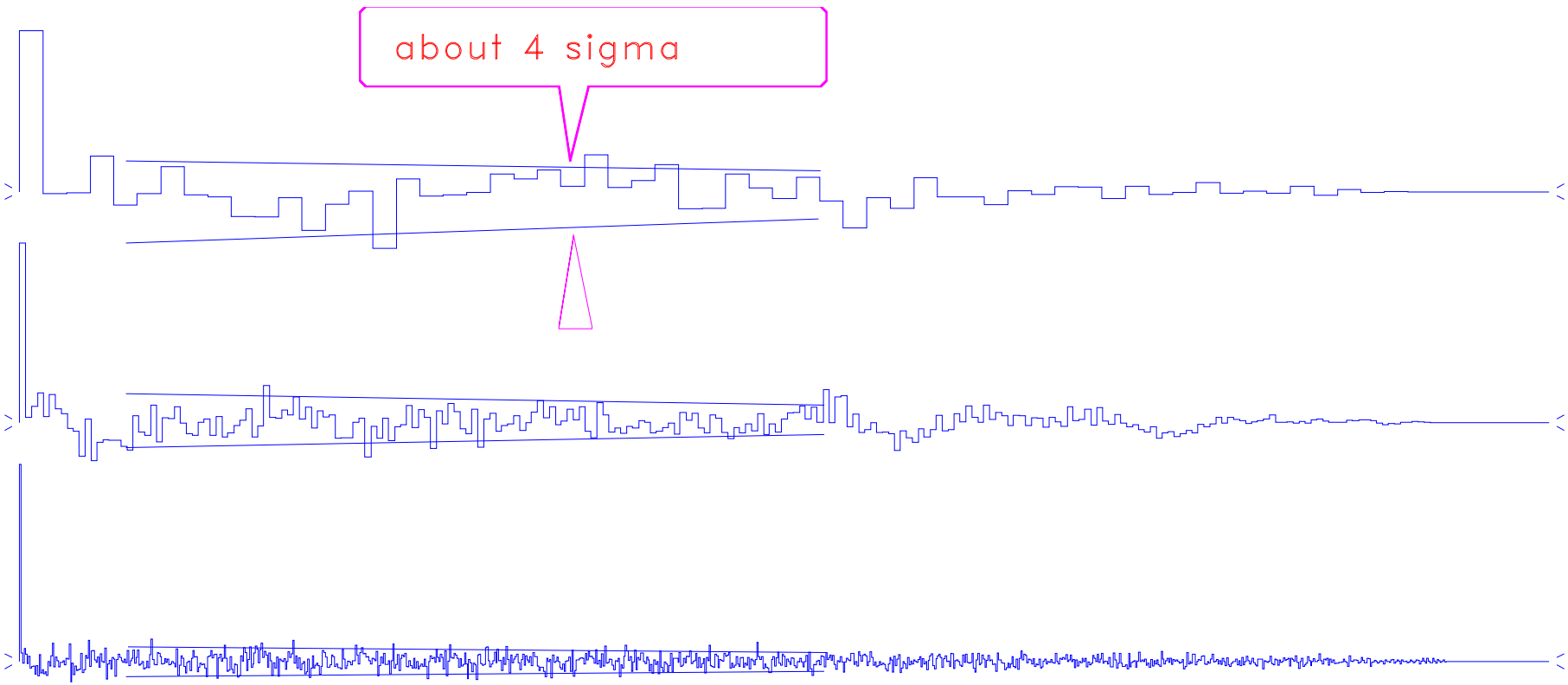


Figure 11.9: Autocorrelation as a function of number of data points. The random-noise-series (even) lengths are 60, 240, 960. rand-fluct [NR]

dropping off as $\sqrt{n}$. We also observe that the ratios between the values for the first nonzero lags and the value at lag zero roughly fit $1/\sqrt{n}$. Notice also that the fluctuations drop off with lag. The drop-off goes to zero at a lag equal to the sample length, because the number of terms in the autocorrelation diminishes to zero at that lag. A first impression is that the autocorrelation fits a triangular envelope. More careful inspection, however, shows that the triangle bulges upward at wide offsets, or large values of k (this is slightly clearer in Figure 11.8). Let us explain all these observations. Each lag of the autocorrelation is defined as

$$s_k = \sum_{t=1}^{n-k} x_t x_{t+k} \tag{11.36}$$

where (x_t) is a sequence of zero-mean *independent* random variables. Thus, the expectations of the autocorrelations can be easily computed:

$$\mathrm{E}(s_0) = \sum_{1}^{n} \mathrm{E}(x_t^2) = n\sigma_x^2 \tag{11.37}$$

$$\mathrm{E}(s_k) \quad = \quad \sum_{1}^{n-k} \mathrm{E}(x_t)\mathrm{E}(x_{t+k}) \quad = \quad 0 \quad (\text{for } k \geq 1) \qquad (11.38)$$

In Figure 11.9, the value at lag zero is more or less $n\sigma_x^2$ (before normalization), the deviation being more or less the standard deviation (square root of the variance) of s_0. On the other hand, for $k \geq 1$, as $\mathrm{E}(s_k) = 0$, the value of the autocorrelation is directly the deviation of s_k, i.e., something close to its standard deviation. We now have to compute the variances of the s_k. Let us write

$$s_k \quad = \quad \sum_{t=1}^{n-k} y_k(t) \qquad (\text{where } y_k(t) = x_t x_{t+k}) \qquad (11.39)$$

So: $s_k = (n-k)\hat{m}_{y_k}$, where $\hat{m}_{y_k}$ is the sample mean of y_k with $n-k$ terms. If $k \neq 0$, $\mathrm{E}(y_k) = 0$, and we apply (11.33) to $\hat{m}_{y_k}$:

$$\mathrm{E}(\hat{m}_{y_k}^2) \quad = \quad \frac{\sigma_{y_k}^2}{n-k} \qquad (11.40)$$

The computation of $\sigma^2_{y_k}$ is straightforward:

$$\sigma^2_{y_k} \quad = \quad \mathrm{E}(x_t^2 x_{t+k}^2) \quad = \quad \mathrm{E}(x_t^2)\mathrm{E}(x_{t+k}^2) \quad = \quad \sigma_x^4\,, \tag{11.41}$$

Thus, for the autocorrelation s_k:

$$\mathrm{E}(s_k^2) \quad = \quad (n-k)\sigma^2_{y_k} \quad = \quad (n-k)\sigma_x^4 \quad = \quad \frac{n-k}{n^2}(\mathrm{E}(s_0))^2 \tag{11.42}$$

Finally, as $\mathrm{E}(s_k) = 0$, we get

$$\sigma_{s_k} \quad = \quad \sqrt{E(s_k^2)} \quad = \quad \mathrm{E}(s_0)\,\frac{\sqrt{n-k}}{n} \tag{11.43}$$

This result explains the properties observed in Figure 11.9. As $n \to \infty$, all the nonzero lags tend to zero compared to the zero lag, since $\sqrt{n-k}/n$ tends to zero. Then, the first lags ($k << n$) yield the ratio $1/\sqrt{n}$ between the autocorrelations and the value at lag zero. Finally, the autocorrelations do not decrease linearly with k, because of $\sqrt{n-k}$.

We can now explain the paradox. The energy of the nonzero lags will be

$$\mathcal{E} \quad = \quad \sum_{k\neq 0} \mathrm{E}(s_k^2) \quad = \quad \frac{(\mathrm{E}(s_0))^2}{n^2} \sum_{k=1}^{n} (n-k) \quad = \quad (\mathrm{E}(s_0))^2 \frac{n(n-1)}{n^2} \qquad (11.44)$$

Hence there is a conflict between the decrease to zero of the autocorrelations and the increasing number of nonzero lags, which themselves prevent the energy from decreasing to zero. The autocorrelation does not *globally* tend to an impulse function. In the frequency domain, the spectrum $S(\omega)$ is now

$$S(\omega) \quad = \quad \frac{1}{n}(s_0 + s_1 \cos\omega + s_2 \cos 2\omega + \cdots) \qquad (11.45)$$

So $\mathrm{E}[S(\omega)] = (1/n)\mathrm{E}[s_0] = \sigma_x^2$, and the *average* spectrum is a constant, independent of the frequency. However, as the s_k fluctuate more or less like $\mathrm{E}[s_0]/\sqrt{n}$, and as their count in $S(\omega)$ is increasing with n, we will observe that $S(\omega)$ will also fluctuate, and indeed,

$$S(\omega) \quad = \quad \frac{1}{n}\mathrm{E}[s_0] \pm \frac{1}{n}\mathrm{E}[s_0] \quad = \quad \sigma_x^2 \pm \sigma_x^2 \qquad (11.46)$$

This explains why the spectrum remains fuzzy: the fluctuation is independent of the number of samples, whereas the autocorrelation seems to tend to an impulse. In conclusion, the expectation (ensemble average) of the spectrum is not properly estimated by letting $n \to \infty$ in a sample.

11.4.2. An example of the bandwidth/reliability tradeoff

Letting n go to infinity does not take us to the expectation $\hat{S} = \sigma^2$. The problem is, as we increase n, we increase the frequency resolution but not the statistical resolution (i.e., the fluctuation around $\hat{S}$). To increase the statistical resolution, we need to simulate ensemble averaging. There are two ways to do this:

1. Take the sample of n points and break it into k equal-length segments of n/k points each. Compute an $S(\omega)$ for each segment and then average all k of the $S(\omega)$ together. The variance of the average spectrum is equal to the variance of each spectrum (σ_x^2) *divided* by the number of segments, and so the fluctuation is substantially reduced.

2. Form $S(\omega)$ from the n-point sample. Replace each of the $n/2$ independent amplitudes by an average over its k nearest neighbors. This could also be

done by tapering the autocorrelation.

The second method is illustrated in Figure 11.10. This figure shows a noise burst of 240 points. Since the signal is even, the burst is effectively 480 points wide, so the autocorrelation is 480 points from center to end: the number of samples will be the same for all cases. The spectrum is very rough. Multiplying the autocorrelation by a triangle function effectively smooths the spectrum by a sinc-squared function, thus reducing the spectral resolution ($1/\Lambda F$). Notice that ΛF is equal here to the width of the sinc-squared function, which is inversely proportional to the length of the triangle (ΛT_{auto}).

However, the first taper takes the autocorrelation width from 480 lags to 120 lags. Thus the spectral fluctuations ΛS should drop by a factor of 2, since the count of terms s_k in $S(\omega)$ is reduced to 120 lags. The width of the next weighted autocorrelation width is dropped from 480 to 30 lags. Spectral roughness should consequently drop by another factor of 2. In all cases, the *average* spectrum is unchanged, since the first lag of the autocorrelations is unchanged. This implies a reduction in the relative spectral fluctuation proportional to the square root of the length of the triangle ($\sqrt{\Lambda T_{\text{auto}}}$).

Our conclusion follows:

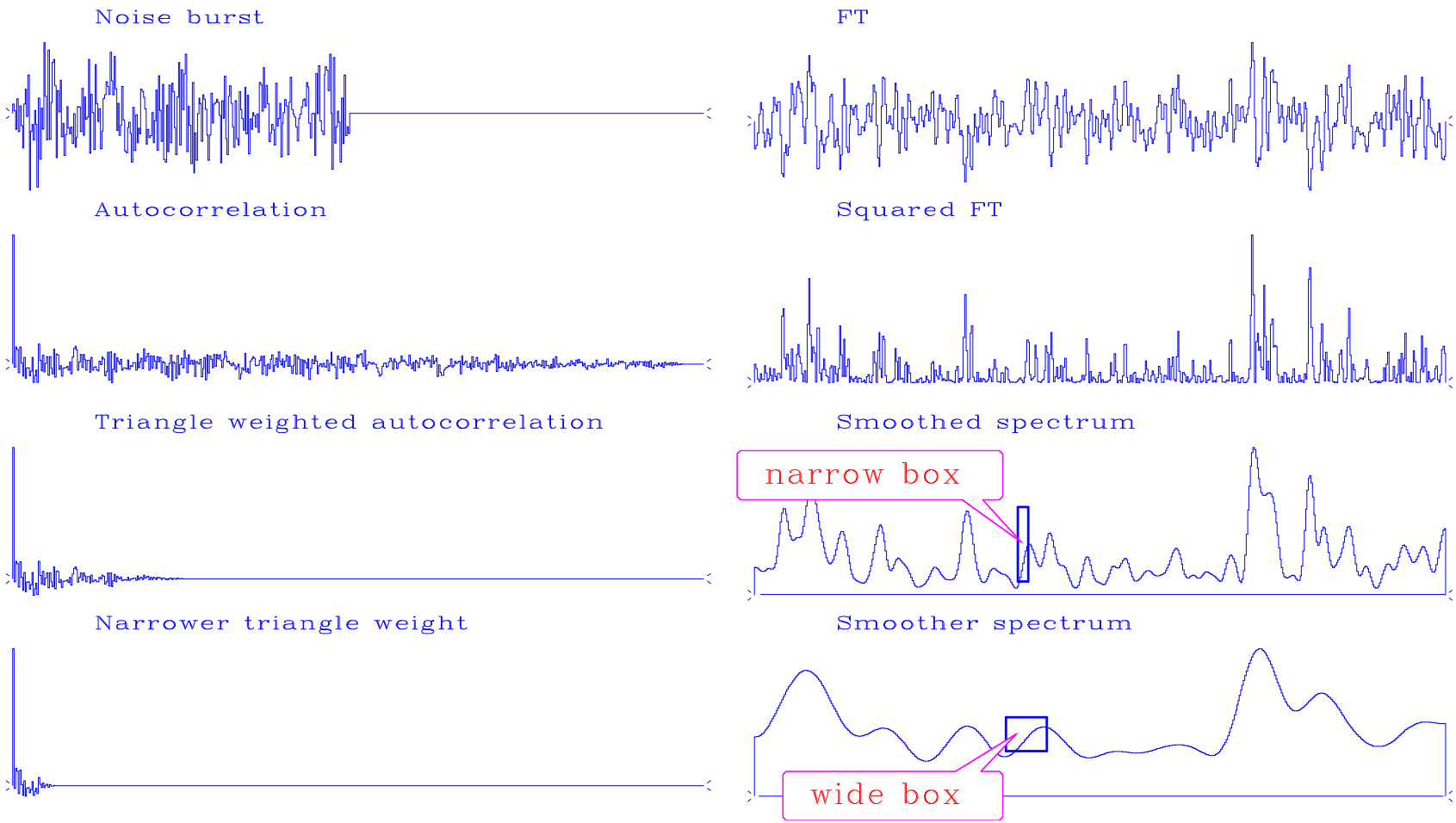


Figure 11.10: Spectral smoothing by tapering the autocorrelation. ΛT is constant and specified on the top row. Successive rows show ΛF increasing while ΛS decreases. The width of a superimposed box roughly gives ΛF, and its height roughly gives ΛS. rand-taper [NR]

The trade-off among **resolution**s of time, frequency, and spectral amplitude is

$$\Lambda F \, \Lambda T \left(\frac{\Lambda S}{S} \right)^2 \quad > \quad 1 \qquad (11.47)$$

11.4.3. Spectral estimation

In Figure 11.10 we did not care about spectral resolution, since we knew theoretically that the spectrum was white. But in practice we do not have such foreknowledge. Indeed, the random factors we deal with in nature rarely are white. A widely used model for naturally occurring random functions, such as microseism, or sometimes reflection seismograms, is white noise put into a filter. The spectra for an example of this type are shown in Figure 11.7. We can see that smoothing the envelope of the power spectrum of the output gives an estimate of the spectrum of the filter. But we also see that the estimate may need even more smoothing.

11.5. CROSSCORRELATION AND COHERENCY

With two time series we can see how crosscorrelation and coherency are related.

11.5.1. Correlation

"**Correlation**" is a concept similar to cosine. A cosine measures the angle between two vectors. It is given by the dot product of the two vectors divided by their magnitudes:

$$c = \frac{(\mathbf{x} \cdot \mathbf{y})}{\sqrt{(\mathbf{x} \cdot \mathbf{x})(\mathbf{y} \cdot \mathbf{y})}} \tag{11.48}$$

This is the **sample normalized correlation** we first encountered on page **??** as a quality measure of fitting one image to another.

Formally, the **normalized correlation** is defined using x and y as zero-mean, scalar, random variables instead of sample vectors. The summation is thus an expectation instead of a dot product:

$$c = \frac{\mathrm{E}(xy)}{\sqrt{\mathrm{E}(x^2)\,\mathrm{E}(y^2)}} \tag{11.49}$$

A practical difficulty arises when the ensemble averaging is simulated over a sample. The problem occurs with small samples and is most dramatically illustrated when we deal with a sample of only one element. Then the sample correlation is

$$\hat{c} \quad = \quad \frac{xy}{|x||y|} \quad = \quad \pm 1 \tag{11.50}$$

regardless of what value the random number x or the random number y should take. For any n, the sample correlation $\hat{c}$ scatters away from zero. Such scatter is called "bias." The topic of bias and variance of coherency estimates is a complicated one, but a rule of thumb seems to be to expect bias and variance of $\hat{c}$ of about $1/\sqrt{n}$ for samples of size n. Bias, no doubt, accounts for many false "discoveries," since cause-and-effect is often inferred from correlation.

11.5.2. Coherency

The concept of "**coherency**" in time-series analysis is analogous to correlation. Taking x_t and y_t to be time series, we find that they may have a mutual relationship which could depend on time delay, scaling, or even filtering. For example, perhaps $Y(Z) = F(Z)X(Z) + N(Z)$, where $F(Z)$ is a filter and n_t is unrelated noise. The

generalization of the correlation concept is to define coherency by

$$C \quad = \quad \frac{\mathrm{E}\left[X\left(\frac{1}{Z}\right)Y(Z)\right]}{\sqrt{\mathrm{E}(\overline{X}X)\mathrm{E}(\overline{Y}Y)}} \tag{11.51}$$

Correlation is a real scalar. *Coherency* is a complex function of frequency; it expresses the frequency dependence of correlation. In forming an estimate of coherency, it is always essential to simulate ensemble averaging. Note that if the ensemble averaging were to be omitted, the coherency (squared) calculation would give

$$|C|^2 \quad = \quad \overline{C}C \quad = \quad \frac{(\overline{\overline{X}Y})(\overline{X}Y)}{(\overline{X}X)(\overline{Y}Y)} \quad = \quad 1 \tag{11.52}$$

which states that the coherency squared is unity, independent of the data. Because correlation scatters away from zero, we find that coherency squared is biased away from zero.

11.5.3. The covariance matrix of multiple signals

A useful model of *single*-channel time-series analysis is that random numbers x_t enter a filter f_t and come out as a signal y_t. A useful model of *multiple*-channel **time-series analysis**—with two channels, for example—is to start with independent random numbers in both the $x_1(t)$ channel and the $x_2(t)$ channel. Then we need *four* filters, $f_{11}(t)$, $f_{12}(t)$, $f_{21}(t)$, and $f_{22}(t)$, which produce two output signals defined by the Z-transforms

$$Y_1(Z) = B_{11}(Z)X_1(Z) + B_{12}(Z)X_2(Z) \tag{11.53}$$

$$Y_2(Z) = B_{21}(Z)X_1(Z) + B_{22}(Z)X_2(Z) \tag{11.54}$$

These signals have realistic characteristics. Each has its own spectral color. Each has a partial relationship to the other which is characterized by a spectral amplitude and phase. Typically we begin by examining the **covariance matrix**. For example, consider two time series, $y_1(t)$ and $y_2(t)$. Their Z-transforms are $Y_1(Z)$ and $Y_2(Z)$. Their covariance matrix is

$$\begin{bmatrix} \mathrm{E}[\overline{Y_1}(1/Z)Y_1(Z)] & \mathrm{E}[\overline{Y_1}(1/Z)Y_2(Z)] \\ \mathrm{E}[\overline{Y_2}(1/Z)Y_1(Z)] & \mathrm{E}[\overline{Y_2}(1/Z)Y_2(Z)] \end{bmatrix} = \mathrm{E}\left(\begin{bmatrix} \overline{Y_1}(1/Z) \\ \overline{Y_2}(1/Z) \end{bmatrix} \begin{bmatrix} Y_1(Z) & Y_2(Z) \end{bmatrix} \right) \tag{11.55}$$

Here Z-transforms represent the components of the matrix in the frequency domain. In the time domain, each of the four elements in the matrix of (11.55) becomes a **Toeplitz** matrix, a matrix of correlation functions (see page **??**).

The expectations in equation (11.55) are specified by theoretical assertions or estimated by sample averages or some combination of the two. Analogously to spectral **factorization**, the covariance matrix can be factored into two parts, $\mathbf{U}'\mathbf{U}$, where $\mathbf{U}$ is an upper triangular matrix. The factorization might be done by the well known **Cholesky** method. The factorization is a multichannel generalization of spectral factorization and raises interesting questions about minimum-phase that are partly addressed in FGDP.

11.5.4. Bispectrum

The "bispectrum" is another statistic that is used to search for nonlinear interactions. For a Fourier transform $F(\omega)$, it is defined by

$$B(\omega_1,\omega_2) \quad = \quad \mathrm{E}[F(\omega_1)F(\omega_2)\overline{F(\omega_1+\omega_2)}] \tag{11.56}$$

A statistic defined analogously is the "bispectral coherency." In seismology, signals rarely have adequate duration for making sensible bispectral estimates from time

averages.

11.6. SMOOTHING IN TWO DIMENSIONS

In previous sections we assumed that we were using one-dimensional models, and smoothing was easy. Working in two dimensions is nominally much more costly, but some tricks are available to make things easier. Here I tell you my favorite trick for **smoothing** in two dimensions. You can convolve with a two-dimensional (almost) Gaussian weighting function *of any area* for a cost of only sixteen additions per output point. (You might expect instead a cost proportional to the area.)

11.6.1. Tent smoothing

First recall triangular smoothing in one dimension with subroutine `triangle()` /prog:triangle. This routine is easily adapted to two dimensions. First we smooth in the direction of the 1-axis for all values of the 2-axis. Then we do the reverse, convolve on the 2-axis for all values of the 1-axis. Now recall that smoothing with a rectangle is especially fast, because we do not need to add all the points within

the rectangle. We merely adapt a shifted rectangle by adding a point at one end and subtracting a point at the other end. In other words, the cost of smoothing is independent of the width of the rectangle. And no multiplies are required. To get a triangle, we smooth twice with rectangles.

Figure 11.11 shows the application of triangle smoothers on two pulses in a plane. The plane was first convolved with a triangle on the 1-axis and then with another triangle on the 2-axis. This takes each impulse and smooths it into an interesting pyramid that I call a tent. The expected side-boundary effect is visible on the foreground tent. In the contour plot (of the same 120 by 40 mesh), we see that the cross section of the tent is rectangular near the base and diamond shaped near the top. The altitude of the jth tent face is $z = a(x - x_j)(y - y_j)$, where (x_j, y_j) is the location of a corner and a is a scale. The tent surface is parabolic (like $z = x^2$) along $x = y$ but linear along lines parallel to the axes. A contour of constant z is the (hyperbolic) curve $y = a + b/(x + c)$ (where a, b, and c are different constants on each of the four faces).

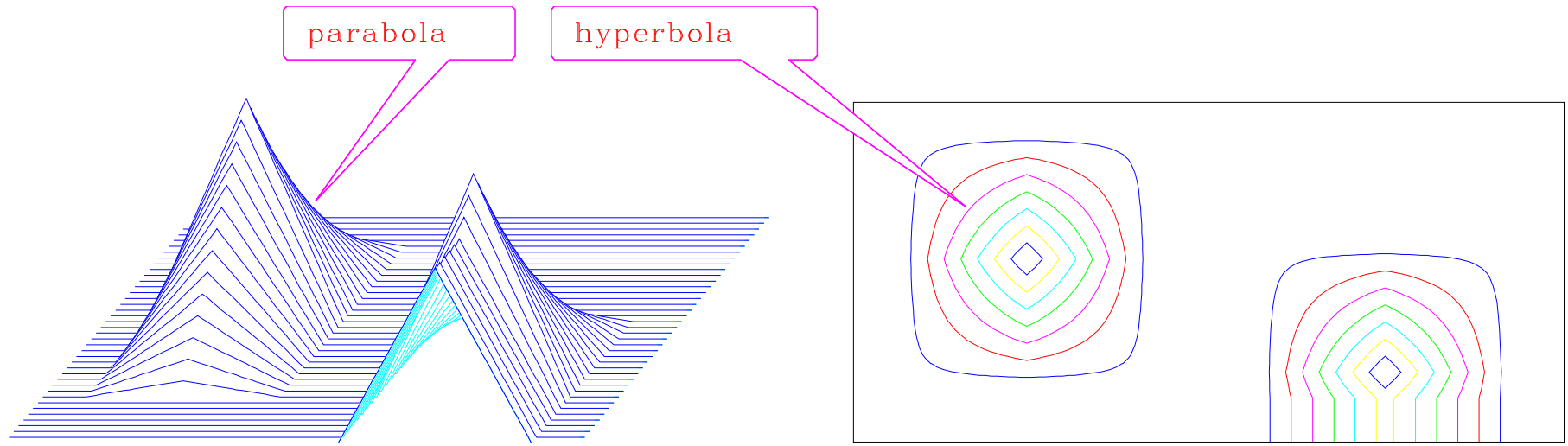


Figure 11.11: Two impulses in two dimensions filtered with a triangle function along each spatial axis. Left: bird's-eye view. Right: contours of constant altitude z. rand-pyram [NR]

11.6.2. Gaussian mounds

In Figure 11.12 we see the result of applying tent smoothing twice. Notice that

Impulsive_plane_wave

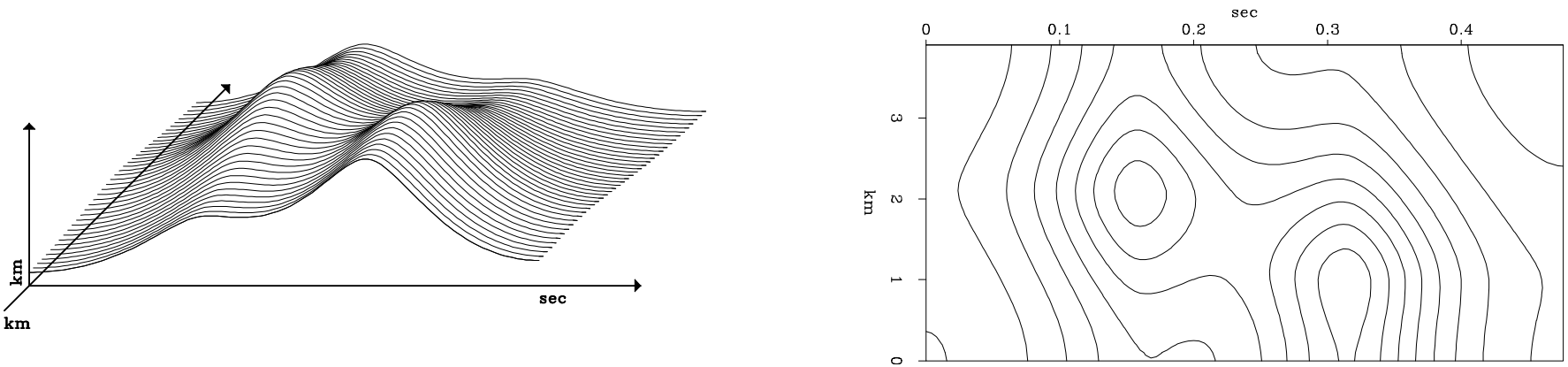


Figure 11.12: Two impulses in two dimensions filtered twice on each axis with a triangle function. Left: bird's-eye view. Right: contours of constant altitude z. rand-mound [ER]

the contours, instead of being diamonds and rectangles, have become much more circular. The reason for this is briefly as follows: convolution of a rectangle with itself many times approachs the limit of a **Gaussian** function. (This is a well-known result called the "**central-limit theorem**." It is explained in section 11.7.) It happens that the convolution of a triangle with itself is already a good approximation to the Gaussian function $z(x) = e^{-x^2}$. The convolution in y gives $z(x,y) = e^{-x^2-y^2} = e^{-r^2}$, where r is the radius of the circle. When the triangle on the 1-axis differs in width from the triangle on the 2-axis, then the circles become ellipses.

11.6.3. Speed of 2-D Gaussian smoothing

This approximate Gaussian smoothing in two dimensions is very fast. Only eight add-subtract pairs are required per output point, and no multiplies at all are required except for final scaling. The compute time is independent of the widths of the Gaussian(!). (You should understand this if you understood that one-dimensional convolution with a rectangle requires just one add-subtract pair per output point.) Thus this technique should be useful in two-dimensional slant stack.

EXERCISES:

1 Deduce that a 2-D filter based on the subroutine `triangle()` /prog:triangle which produces the 2-D quasi-Gaussian mound in Figure 11.12 has a gain of unity at zero (two-dimensional) frequency (also known as $(k_x, k_y) = 0$).

2 Let the 2-D quasi-Gaussian filter be known as F. Sketch the spectral response of F.

3 Sketch the spectral response of $1 - F$ and suggest a use for it.

4 The tent filter can be implemented by smoothing first on the 1-axis and then on the 2-axis. The conjugate operator smooths first on the 2-axis and then on the 1-axis. The tent-filter operator should be self-adjoint (equal to its conjugate), unless some complication arises at the sides or corners. How can a dot-product test be used to see if a tent-filter program is self-adjoint?

11.7. PROBABILITY AND CONVOLUTION

One way to obtain **random** integers from a known **probability** function is to write integers on slips of paper and place them in a hat. Draw one slip at a time. After

each drawing, replace the slip in the hat. The probability of drawing the integer i is given by the ratio a_i of the number of slips containing the integer i divided by the total number of slips. Obviously the sum over i of a_i must be unity. Another way to get random integers is to throw one of a pair of dice. Then all a_i equal zero except $a_1 = a_2 = a_3 = a_4 = a_5 = a_6 = \frac{1}{6}$. The probability that the integer i will occur on the first drawing and the integer j will occur on the second drawing is $a_i a_j$. If we draw two slips or throw a pair of dice, then the probability that the sum of i and j equals k is the sum of all the possible ways this can happen:

$$c_k \quad = \quad \sum_i a_i a_{k-i} \tag{11.57}$$

Since this equation is a **convolution**, we may look into the meaning of the Z-transform

$$A(Z) \quad = \quad \cdots a_{-1} Z^{-1} + a_0 + a_1 Z + a_2 Z^2 + \cdots \tag{11.58}$$

In terms of Z-transforms, the probability that i plus j equals k is simply the coefficient of Z^k in

$$C(Z) \quad = \quad A(Z)\, A(Z) \tag{11.59}$$

The probability density of a *sum* of random numbers is the *convolution* of their probability density functions.

EXERCISES:

1 A random-number generator provides random integers 2, 3, and 6 with probabilities $p(2) = 1/2$, $p(3) = 1/3$, and $p(6) = 1/6$. What is the probability that any given integer n is the sum of three of these random numbers? (HINT: Leave the result in the form of coefficients of a complicated polynomial.)

11.8. THE CENTRAL-LIMIT THEOREM

The **central-limit theorem** of probability and statistics is perhaps the most important theorem in these fields of study. A derivation of the theorem explains why the **Gaussian** probability function is so frequently encountered in nature; not just in physics but also in the biological and social sciences. No experimental scientist should be unaware of the basic ideas behind this theorem. Although the central-

limit theorem is deep and is even today the topic of active research, we can quickly go to the basic idea.

From equation (11.59), if we add n random numbers, the probability that the sum of them equals k is given by the coefficient of Z^k in

$$G(Z) \quad = \quad A(Z)^n \tag{11.60}$$

The central-limit theorem says that as n goes to infinity, the polynomial $G(Z)$ goes to a special form, almost regardless of the specific polynomial $A(Z)$. The specific form is such that a graph of the coefficients of $G(Z)$ comes closer and closer to fitting under the envelope of the bell-shaped Gaussian function. This happens because, if we raise any function to a high enough power, eventually all we can see is the highest value of the function and its immediate environment, i.e., the second derivative there. We already saw an example of this in Figure **??**. Exceptions to the central-limit theorem arise (1) when there are multiple maxima of the same height, and (2) where the second derivative vanishes at the maximum.

Although the central-limit theorem tells us that a Gaussian function is the limit as $n \to \infty$, it does not say anything about how fast the limit is attained. To test this, I plotted the coefficients of $(\frac{1}{4Z} + \frac{1}{2} + \frac{1}{4}Z)^n$ for large values of n. This signal

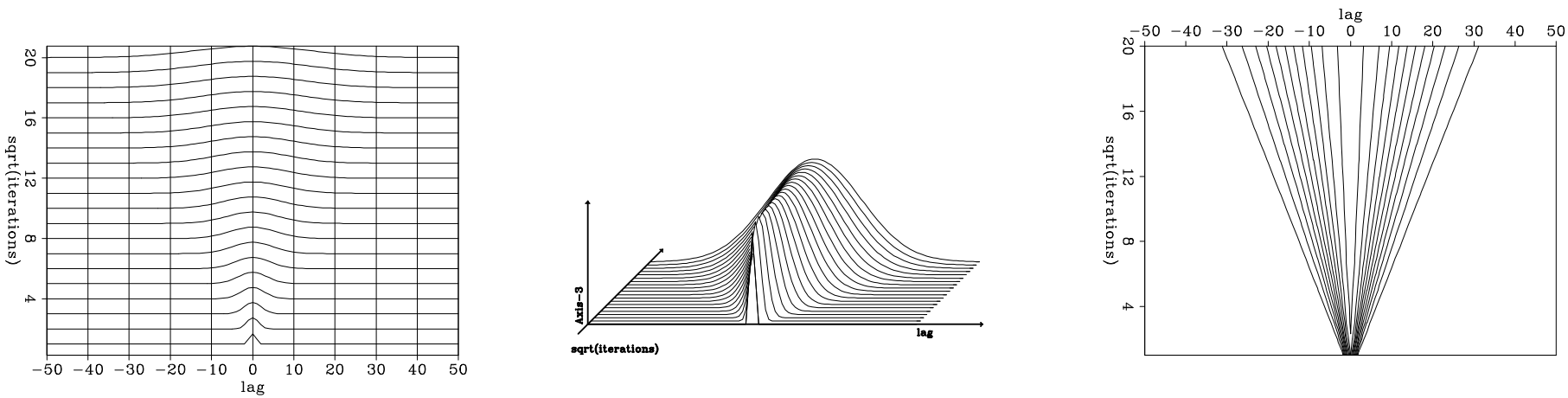


Figure 11.13: Left: wiggle plot style. Middle: perspective. Right: contour. rand-clim [ER]

is made up of scaled **binomial coefficients**. To keep signals in a suitable amplitude scale, I multiplied them by $\sqrt{n}$. Figure 11.13 shows views of the coefficients of $\sqrt{n}(\frac{1}{4Z} + \frac{1}{2} + \frac{1}{4}Z)^n$ (horizontal axis) versus $\sqrt{n}$ (vertical axis). We see that scaling by $\sqrt{n}$ has kept signal peak amplitudes constant. We see also that the width of the signal increases linearly with $\sqrt{n}$. The contours of constant amplitude show that the various orders are self-similar with the width stretching.

Sums of independently chosen random variables tend to have Gaussian probability density functions.

Chapter 12

Entropy and Jensen inequality

Jensen inequality is my favorite theory-that-never-quite-made-it-into-practice, but there is still hope!

In this book we have solved many problems by minimizing a weighted sum of

squares. We understand vaguely that the weights should somehow be the inverse to the expected value of the object they weight. We really cannot justify the *square*, however, except to say that it makes residuals positive, and positive residuals lead to ready methods of analysis. Here we will think about a more general approach, more clumsy in computation, but potentially more powerful in principle. As we begin with some mathematical abstractions, you should think of applications where populations such as envelopes, spectra, or magnitudes of residuals are adjustable by adjusting model parameters. What you will see here is a wide variety of ways that equilibrium can be defined.

12.1. THE JENSEN INEQUALITY

Let f be a function with a positive second derivative. Such a function is called “convex" and satisfies the **inequality**

$$\frac{f(a) + f(b)}{2} - f\left(\frac{a+b}{2}\right) \geq 0 \qquad (12.1)$$

Equation (12.1) relates a function of an average to an average of the function. The average can be weighted, for example,

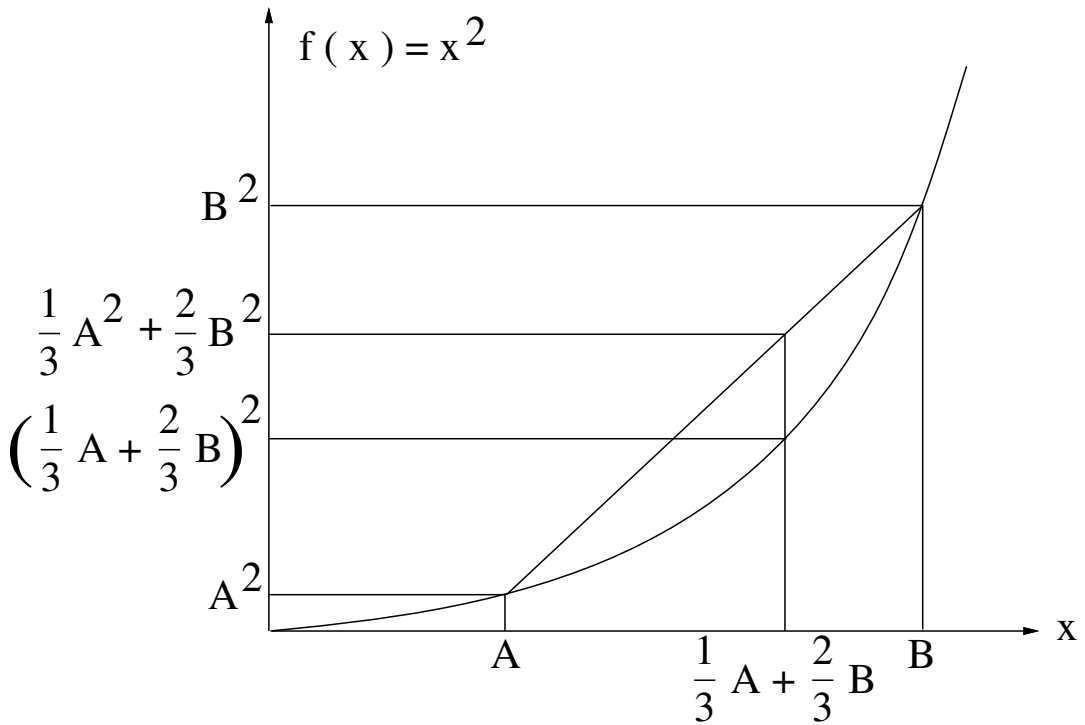
$$\frac{1}{3}\,f(a) + \frac{2}{3}\,f(b) - f\left(\frac{1}{3}a + \frac{2}{3}b\right) \quad \geq \quad 0 \tag{12.2}$$

Figure 12.1 is a graphical interpretation of equation (12.2) for the function $f = x^2$. There is nothing special about $f = x^2$, except that it is convex. Given three numbers a, b, and c, the inequality (12.2) can first be applied to a and b, and then to c and the average of a and b. Thus, recursively, an inequality like (12.2) can be built for a weighted average of three or more numbers. Define weights $w_j \geq 0$ that are normalized ($\sum_j w_j = 1$). The general result is

$$S(p_j) \quad = \quad \sum_{j=1}^{N} w_j\, f(p_j) - f\left(\sum_{j=1}^{N} w_j\, p_j\right) \quad \geq \quad 0 \tag{12.3}$$

If all the p_j are the same, then both of the two terms in S are the same, and S vanishes. Hence, minimizing S is like urging all the p_j to be identical. Equilibrium is when S is reduced to the smallest possible value which satisfies any constraints

Figure 12.1: Sketch of $y = x^2$ for interpreting equation ((12.2)). jen-jen [NR]



that may be applicable. The function S defined by (12.3) is like the **entropy** defined in **thermodynamics**.

12.1.1. Examples of Jensen inequalities

The most familiar example of a Jensen inequality occurs when the weights are all equal to $1/N$ and the convex function is $f(x) = x^2$. In this case the Jensen inequality gives the familiar result that the mean square exceeds the square of the mean:

$$Q \quad = \quad \frac{1}{N}\sum_{i=1}^{N} x_i^2 - \left(\frac{1}{N}\sum_{i=1}^{N} x_i\right)^2 \quad \geq \quad 0 \tag{12.4}$$

In the other applications we will consider, the population consists of positive members, so the function $f(p)$ need have a positive second derivative only for positive values of p. The function $f(p) = 1/p$ yields a Jensen inequality for the **harmonic mean**:

$$H \quad = \quad \sum \frac{w_i}{p_i} - \frac{1}{\sum w_i\, p_i} \quad \geq \quad 0 \tag{12.5}$$

A more important case is the **geometric inequality**. Here $f(p) = -\ln(p)$, and

$$G \quad = \quad -\sum w_i \ln p_i \; + \; \ln \sum w_i \, p_i \quad \geq \quad 0 \tag{12.6}$$

The more familiar form of the geometric inequality results from exponentiation and a choice of weights equal to 1/N:

$$\frac{1}{N} \sum_{i=1}^{N} p_i \quad \geq \quad \prod_{i=1}^{N} p_i^{1/N} \tag{12.7}$$

In other words, the product of square roots of two values is smaller than half the sum of the values. A Jensen inequality with an adjustable parameter is suggested by $f(p) = p^{\gamma}$:

$$\Gamma_{\gamma} \quad = \quad \sum_{i=1}^{N} w_i \, p_i^{\gamma} \; - \; \left(\sum_{i=1}^{N} w_i \, p_i \right)^{\gamma} \tag{12.8}$$

Whether Γ is always positive or always negative depends upon the numerical value of γ. In practice we may see the dimensionless form, in which the ratio instead of the difference of the two terms is used. A most important inequality in information theory and thermodynamics is the one based on $f(p) = p^{1+\epsilon}$, where ϵ is a

small positive number tending to zero. I call this the “weak" inequality. With some calculation we will quickly arrive at the limit:

$$\sum w_i\, p_i^{1+\epsilon} \quad \geq \quad \left(\sum w_i\, p_i\right)^{1+\epsilon} \tag{12.9}$$

Take logarithms

$$\ln \sum w_i\, p_i^{1+\epsilon} \quad \geq \quad (1+\epsilon) \ln \sum w_i\, p_i \tag{12.10}$$

Expand both sides in a Taylor series in powers of ϵ using

$$\frac{d}{d\epsilon} a^u \quad = \quad \frac{du}{d\epsilon} a^u \ln a \tag{12.11}$$

The leading term is identical on both sides and can be canceled. Divide both sides by ϵ and go to the limit $\epsilon = 0$, obtaining

$$\frac{\sum w_i\, p_i \ln p_i}{\sum w_i\, p_i} \quad \geq \quad \ln \sum w_i\, p_i \tag{12.12}$$

We can now define a positive variable S' with or without a positive scaling factor $\sum wp$:

$$S'_{\text{intensive}} = \frac{\sum w_i\, p_i \ln p_i}{\sum w_i\, p_i} - \ln \sum w_i\, p_i \quad \geq \quad 0 \tag{12.13}$$

$$S'_{\text{extensive}} = \sum w_i\, p_i \ln p_i - \left(\sum w_i\, p_i\right) \ln\left(\sum w_i\, p_i\right) \quad \geq \quad 0 \tag{12.14}$$

Seismograms often contain zeros and gaps. Notice that a single zero p_i can upset the harmonic H or geometric G inequality, but a single zero has no horrible effect on S or Γ.

12.2. RELATED CONCEPTS

In practice we may wonder which Jensen inequality to use.

12.2.1. Prior and posterior distributions

Random variables have a **prior distribution** and a **posterior distribution**. Denote the prior by b_i (for “before") and posterior by a_i (for “after"). Define $p_i = a_i / b_i$, and insert p_i in any of the inequalities above. Now suppose we have an adjustable model parameter upon which the a_i all depend. Suppose we adjust that model parameter to try to make some Jensen inequality into an equality. Thus we will be adjusting it to get all the p_i equal to each other, that is, to make all the posteriors equal to their priors. It is nice to have so many ways to do this, one for each Jensen inequality. The next question is, which Jensen inequality should we use? I cannot answer this directly, but we can learn more about the various inequalities.

12.2.2. Jensen average

Physicists speak of maximizing **entropy**, which, if we change the polarity, is like minimizing the various Jensen inequalities. As we minimize a Jensen inequality, the small values tend to get larger while the large values tend to get smaller. For each population of values there is an average value, i.e., a value that tends to get neither larger nor smaller. The average depends not only on the population, but also on

the definition of entropy. Commonly, the p_j are positive and $\sum w_j\, p_j$ is an energy. Typically the total energy, which will be fixed, can be included as a constraint, or we can find some other function to minimize. For example, divide both terms in (12.3) by the second term and get an expression which is scale invariant; i.e., scaling p leaves (12.15) unchanged:

$$\frac{\sum_{j=1}^{N} w_j\, f(p_j)}{f\left(\sum_{j=1}^{N} w_j\, p_j\right)} \quad \geq \quad 1 \tag{12.15}$$

Because the expression exceeds unity, we are tempted to take a logarithm and make a new function for minimization:

$$J \quad = \quad \ln\left(\sum_j w_j\, f(p_j)\right) \; - \; \ln\left[\, f\left(\sum_j w_j\, p_j\right)\right] \quad \geq \quad 0 \tag{12.16}$$

Given a population p_j of positive variants, and an inequality like (12.16), I am now prepared to define the "**Jensen average**" $\overline{p}$. Suppose there is one element, say p_J, of the population p_j that can be given a first-order perturbation, and only a second-order perturbation in J will result. Such an element is in equilibrium and is the

Jensen average $\overline{p}$:

$$0 \quad = \quad \frac{\partial J}{\partial p_J}\Bigg]_{p_J=\overline{p}} \tag{12.17}$$

Let f_p denote the derivative of f with respect to its argument. Inserting (12.16) into (12.17) gives

$$0 \quad = \quad \frac{\partial J}{\partial p_J} \quad = \quad \frac{w_J \ f_p(p_J)}{\sum w_j \ f(p_j)} - \frac{f_p(\sum_{j=1}^{N} w_j p_j) w_J}{f\left(\sum w_j p_j\right)} \tag{12.18}$$

Solving,

$$\overline{p} \quad = \quad p_J \quad = \quad f_p^{-1}\left(e^J \ f_p(\sum_{j=1}^{N} w_j p_j)\right) \tag{12.19}$$

But where do we get the function f, and what do we say about the equilibrium value? Maybe we can somehow derive f from the population. If we cannot work out a general theory, perhaps we can at least find the constant γ, assuming the functional form to be $f = p^\gamma$.

12.2.3. Additivity of envelope entropy to spectral entropy

In some of my efforts to fill in missing data with **entropy** criteria, I have often based the entropy on the **spectrum** and then found that the **envelope** would misbehave. I have come to believe that the definition of entropy should involve both the spectrum and the envelope. To get started, let us assume that the power of a seismic signal is the product of an envelope function times a spectral function, say

$$u(\omega,t) \quad = \quad p(\omega)e(t) \tag{12.20}$$

Notice that this separability assumption resembles the **stationarity** concept. I am not defending the assumption (12.20), only suggesting that it is an improvement over each term separately. Let us examine some of the algebraic consequences. First evaluate the intensive entropy:

$$S'_{\text{intensive}} \quad = \quad \frac{\sum_t \sum_\omega u \ln u}{\sum_t \sum_\omega u} - \ln \frac{1}{N^2} \sum_t \sum_\omega u \ \geq \ 0 \tag{12.21}$$

$$= \quad \frac{\sum \sum pe(\ln p + \ln e)}{(\sum p)(\sum e)} - \ln \left(\frac{1}{N} \sum_\omega p \ \frac{1}{N} \sum_t e \right) \tag{12.22}$$

$$= \frac{\sum e \sum p \ln p + \sum p \sum e \ln e}{(\sum p)(\sum e)} - \ln \frac{1}{N} \sum p - \ln \frac{1}{N} \sum e \quad (12.23)$$

$$= \left(\frac{\sum p \ln p}{\sum p} - \ln \frac{1}{N} \sum p \right) + \left(\frac{\sum e \ln e}{\sum e} - \ln \frac{1}{N} \sum e \right) \quad (12.24)$$

$$= S(p) + S(e) \geq 0 \quad (12.25)$$

It is remarkable that all the cross terms have disappeared and that the resulting entropy is the sum of the two parts. Now we will tackle the same calculation with the geometric inequality:

$$G = \ln \frac{1}{N^2} \sum \sum u - \frac{1}{N^2} \sum \sum \ln u \quad (12.26)$$

$$= \ln \left[\left(\frac{1}{N} \sum_t e \right) \left(\frac{1}{N} \sum_\omega p \right) \right] - \frac{1}{N^2} \sum_t \sum_\omega (\ln p_\omega + \ln e_t) \quad (12.27)$$

$$= \ln \overline{e} + \ln \overline{p} - \frac{1}{N^2} \sum_t 1_t \sum_\omega \ln p_\omega - \frac{1}{N^2} \sum_\omega 1_\omega \sum_t \ln e_t \quad (12.28)$$

$$= \quad \ln\bar{e} + \ln\bar{p} - \frac{1}{N}\sum_{\omega}\ln p - \frac{1}{N}\sum_{t}\ln e \tag{12.29}$$

$$= \quad G(t) + G(\omega) \tag{12.30}$$

Again all the cross terms disappear, and the resulting entropy is the sum of the two parts. I wonder if this result applies for the other Jensen inequalities.

In conclusion, although this book is dominated by model building using the method of least squares, Jensen inequalities suggest many interesting alternatives.

Chapter 13

RATional FORtran == Ratfor

Bare-bones **Fortran** is our most universal computer language for computational physics. For general programming, however, it has been surpassed by **C**. "**Ratfor**" is Fortran with C-like syntax. I believe Ratfor is the best available expository language

for mathematical algorithms. Ratfor was invented by the people who invented C. Ratfor programs are converted to Fortran with the Ratfor **preprocessor**. Since the preprocessor is publicly available, Ratfor is practically as universal as Fortran.[1]

You will not really need the Ratfor preprocessor or any precise definitions if you already know Fortran or almost any other computer language, because then the Ratfor language will be easy to understand. Statements on a line may be separated by ";." Statements may be grouped together with braces { }. Do loops do not require statement numbers because { } defines the range. Given that `if( )` is true, the statements in the following { } are done. `else{ }` does what you expect. We may *not* contract `else if` to `elseif`. We may always omit the braces { } when they contain only one statement. `break` will cause premature termination of the enclosing { }. `break 2` escapes from {{ }}. `while( ) { }` repeats the statements in { }

---

[1]Kernighan, B.W. and Plauger, P.J., 1976, Software Tools: Addison-Wesley. Ratfor was invented at AT&T, which makes it available directly or through many computer vendors. The original Ratfor transforms Ratfor code to **Fortran** 66. See `http://sepwww.stanford.edu/sep/prof` for a public-domain Ratfor translator to **Fortran** 77.

while the condition ( ) is true. `repeat { ... } until( )` is a loop that tests at the bottom. A looping statement more general than `do` is `for(`initialize`;` condition`;` reinitialize`) { }`. `next` causes skipping to the end of any loop and a retrial of the test condition. `next` is rarely used, but when it is, we must beware of an inconsistancy between Fortran and C-language. Where Ratfor uses `next`, the C-language uses `continue` (which in Ratfor and Fortran is merely a place holder for labels). The Fortran relational operators `.gt.`, `.ge.`, `.ne.`, etc. may be written >, >=, !=, etc. The logical operators `.and.` and `.or.` may be written && and ||. Anything from a # to the end of the line is a comment. Anything that does not make sense to the Ratfor preprocessor, such as Fortran input-output, is passed through without change. (Ratfor has a `switch` statement but we never use it because it conflicts with the `implicit undefined` declaration. Anybody want to help us fix the switch in public domain ratfor?)

Indentation in `Ratfor` is used for readability. It is not part of the `Ratfor` language. Choose your own style. I have overcondensed. There are two **pitfall**s associated with indentation. The beginner's pitfall is to assume that a `do` loop ends where the indentation ends. The loop ends after the first statement. A larger scope for the `do` loop is made by enclosing multiple statements in braces. The other pitfall

arises in any construction like `if() ... if() ... else`. The `else` goes with the last `if()` regardless of indentation. If you want the `else` with the earlier `if()`, you must use braces like `if() { if() ... } else ...`.

The most serious limitation of **Fortran-**77 is its lack of ability to allocate temporary memory. I have written a **preprocessor** to **Ratfor** or **Fortran** to overcome this memory-allocation limitation. This program, named **`sat`**, allows subroutines to include the declaration `temporary real data(n1,n2)`, so that memory is allocated during execution of the subroutine where the declaration is written. Fortran-77 forces us to accomplish something like this only with predetermined constants or parameters. If the **`sat`** preprocessor is not available on your system, you can modify the subroutines in this book by putting the appropriate numerical constants into the memory arrays being allocated, or adapt the programs here to Fortran 90 (although students at Stanford seem to prefer the **`sat`** approach).

Below are simple Ratfor subroutines for erasing an array (`zero()`), (`null()`), for copying one array to another (`copy()`), for vector scaling (`scaleit()`), for the signum function $\mathrm{sgn}(x) = x/|x|$ (`signum()`), for nearest-neighbor interpolation. In the interpolation programs the mathematical concept $x = x_0 + n\Delta x$ is expressed as `x = x0 +(ix-1)*dx`. The idea of "nearest neighbor" arises when backsolving for

the integer `ix`: a half is added to the floating-point value before rounding down to an integer, i.e., `ix = .5 + 1 + (x-x0)/dx`. The file `quantile()` /prog:quantile contains two quantile-finding utilities. The method is the well-known one developed by Hoare. zero null copy scaleit signum quantile rand01

```
subroutine zero( n, xx)
integer i, n;   real xx(n)
do i= 1, n
        xx(i) = 0.
return; end
```

Back

```
subroutine null(     xx, n)
integer i, n;   real xx( n)
do i= 1, n
        xx(i) = 0.
return; end
```

Back

```
subroutine copy( n, xx, yy)
integer i, n;   real xx(n), yy(n)
do i= 1, n
        yy(i) = xx(i)
return; end
```

Back

```
subroutine scaleit( factor, n, data)
integer i,                 n
real factor, data(n)
do i= 1, n
        data(i) = factor * data(i)
return; end
```

Back

```
real function signum( x)
real x
        if      ( x > 0 )  { signum =  1. }
        else if ( x < 0 )  { signum = -1. }
        else               { signum =  0. }
return; end
```

Back

```
# Two quantile utilities.  Changed since formally tested.
#
# value = value of bb(k) if bb(1...n) were sorted into increasing order.
subroutine quantile( k, n, bb, value)
        integer   i, k, n;      real bb(n), value
        temporary real aa(n)
        do i= 1, n
                aa(i) = bb(i)
        call quantinternal( k, n, aa)
        value = aa(k)
        return; end

# value = value of abs(bb(k)) if abs(bb(1...n)) were sorted to increasing order.
subroutine quantabs( k, n, bb, value)
        integer   i, k, n;      real bb(n), value
        temporary real aa(n)
        do i= 1, n
                aa(i) = abs( bb(i))
        call quantinternal( k, n, aa)
        value = aa(k)
        return; end

subroutine quantinternal( k, n, a)
integer k, n;                   real a(n)
integer i, j, low, hi;          real ak, aa
if( k>n || k<1)  call erexit("quant: inputs not in range  1 <= k <= n ")
low = 1;        hi = n
while( low < hi) {
        ak = a(k);      i = low;        j = hi
        repeat {
                if( a(i) < ak)
                        i = i+1
                else {
                        while( a(j) > ak)       j = j-1
                        if( i > j)  break
                        aa = a(i);      a(i) = a(j);    a(j) = aa
                        i = i+1;        j = j-1
                        if( i > j)  break
```

```
real function rand01( iseed)
integer ia, im,       iseed
parameter(ia = 727,im = 524287)
iseed = mod(iseed*ia,im)
rand01 =(float(iseed) - 0.5)/float(im - 1)
return; end
```

Back

Chapter 14

Seplib and SEP software

At the time of writing, this book can be run on a variety of computers. You will have noticed that each figure caption contains a box enclosing a label. In the electronic book, this box is a pushbutton that generally activates a rebuilding of the figure,

sometimes after program or parameter changes and sometimes interactively. The label in the box points to the location of the underlying software. My associates and I have worked through complete cycles of "**burn**ing" and building all the figures on various computers. To enable you to do the same, and to further enable you to rapidly build on my work, I intend to release an electronic copy of the book soon. This short appendix describes the utility software that is used extensively in the electronic book.

Most of the seismic utility software at the **Stanford Exploration Project**[1] (**SEP**) handles seismic data as a rectangular lattice or "cube" of numbers. Each cube-processing program appends to the history file for the cube. Preprocessors extend **Fortran** (or **Ratfor**) to enable it to allocate memory at run time, to facilitate input and output of data cubes, and to facilitate self-documenting programs.

At the SEP a library of subroutines known as **`seplib`** evolved for routine operations. These subroutines mostly handle data in the form of cubes, planes, and vectors. A cube is defined by 14 parameters with standard names and two files: one

---

[1] Old reports of the Stanford Exploration Project can be found in the library of the Society of Exploration Geophysicists in Tulsa, Oklahoma.

the data cube itself, and the other containing the 14 parameters and a history of the life of the cube as it passed through a sequence of cube-processing programs. Most of these cube-processing programs have been written by researchers, but several nonscientific cube programs have become highly developed and are widely shared. Altogether there are (1) a library of subroutines, (2) a library of main programs, (3) some naming conventions, and (4) a graphics library called **`vplot`**. The subroutine library has good manual pages. The main programs rarely have manual pages, their documentation being supplied by the on-line self-documentation that is extracted from the comments at the beginning of the source file. Following is a list of the names of popular main programs:

`Byte`	Scale floats to brightness bytes for raster display.
`Cat`	Concatenate conforming cubes along the 3-axis.
`Contour`	Contour plot a plane.
`Cp`	Copy a cube.
`Dd`	Convert between ASCI, floats, complex, bytes, etc.
`Dots`	Plot a plane of floats.
`Ft3d`	Do three-dimensional Fourier transform.
`Graph`	Plot a line of floats.

`In`	Check the validity of a data cube.
`Merge`	Merge conforming cubes side by side on any axis.
`Movie`	View a cube with Rick Ottolini's cube viewer.
`Noise`	Add noise to data.
`Reverse`	Reverse a cube axis.
`Spike`	Make a plane wave of synthetic data.
`Ta2vplot`	Convert a byte format to raster display with `vplot`.
`Tpow`	Scale data by a power of time *t* (1-axis).
`Thplot`	Make a hidden line plot.
`Transpose`	Transpose cube axes.
`Tube`	View a `vplot` file on a screen.
`Wiggle`	Plot a plane of floats as "wiggle traces."
`Window`	Find a subcube by truncation or subsampling.

To use the cube-processing programs, read this document, and then for each command, read its on-line self-documentation. To write cube-processing programs, read the manual page for `seplib` and the subroutines mentioned there and here. To write `vplot` programs, see the references on `vplot`.

14.1. THE DATA CUBE

The data cube itself is like a Fortran three-dimensional matrix. Its location in the computer file system is denoted by `in=PATHNAME`, where `in=` is the literal occurrence of those three characters, and `PATHNAME` is a directory tree location like `/sep/professor/pvi/data/western73.F`. Like the Fortran cube, the data cube can be real, complex, double precision, or byte, and these cases are distinguished by the element size in bytes. Thus the history file contains one of `esize=4`, `esize=8`, or `esize=1`, respectively. Embedded blanks around the "=" are always forbidden. The cube values are binary information; they cannot be printed or edited (without the intervention of something like a Fortran "format"). To read and write cubes, see the manual pages for such routines as `reed, sreed, rite, srite, snap`.

A cube has three axes. The number of points on the 1-axis is `n1`. A Fortran declaration of a cube could be `real mydata(n1,n2,n3)`. For a plane, `n3=1`, and for a line, `n2=1`. In addition, many programs take "1" as the default for an undefined value of `n2` or `n3`. The physical location of the single data value `mydata(1,1,1)`, like a mathematical origin (o_1, o_2, o_3), is denoted by the three real variables `o1`, `o2`, and `o3`. The data-cube values are presumed to be uniformly spaced along these axes like the mathematical increments $(\Delta_1, \Delta_2, \Delta_3)$, which may be negative and

are denoted by the three real variables `d1`, `d2`, and `d3`. Each axis has a label, and naturally these labels are called `label1`, `label2`, and `label3`. Examples of labels are `kilometers`, `sec`, `Hz`, and `"offset, km"`. Most often, `label1="time, sec"`. Altogether that is $2+3\times4$ parameters, and there is an optional title parameter that is interpreted by most of the plot programs. An example is `title="Yilmaz and Cumro Canada profile 25"`. We reserve the names `n4,o4,d4, and label4` (a few programs support them already), and please do not use `n5` etc. for anything but a five-dimensional cubic lattice.

14.2. THE HISTORY FILE

The 15 parameters above, and many more parameters defined by authors of cube-processing programs, are part of the "**history file**" (which is ASCI, so we can print it). A great many cube-processing programs are simple filters—i.e., one cube goes in and one cube comes out—and that is the case I will describe in detail here. For other cases, such as where two go in and one comes out, or none go in and one comes out (synthetic data), or one goes in and none come out (plotting program), I refer you to the manual pages, particularly to subroutine names beginning with `aux`

(as in “auxiliary").

Let us dissect an example of a simple cube-processing program and its use. Suppose we have a seismogram in a data cube and we want only the first 500 points on it, i.e., the first 500 points on the 1-axis. A utility cube filter named `Window` will do the job. Our command line looks like `< mygiven.H Window n1=500 > myshort.H` On this command line, `mygiven.H` is the name of the history file of the data we are given, and `myshort.H` is the history file we will create. The moment `Window`, or any other `seplib` program, begins, it copies `mygiven.H` to `myshort.H`; from then on, information can only be appended to `myshort.H`. When `Window` learns that we want the 1-axis on our output cube to be 500, it does `call putch('n1','i',500)`, which appends `n1=500` to `myshort.H`. But before this, some other things happen. First, `seplib`’s internals will get our log-in name, the date, the name of the computer we are using, and `Window`’s name (which is `Window`), and append these to `myshort.H`. The internals will scan `mygiven.H` for `in=somewhere` to find the input data cube itself, and will then figure out where we want to keep the output cube. `Seplib` will guess that someone named professor wants to keep his data cube at some place like `/scr/professor/_Window.H@`. You should read the manual page for `datapath` to see how you can set up the default location for your

datasets. The reason `datapath` exists is to facilitate isolating data from text, which is usually helpful for archiving.

When a cube-processing filter wonders what the value is of `n1` for the cube coming in, it makes a subroutine call like `call hetch("n1","i",n1)`. The value returned for `n1` will be the *last* value of `n1` found on the history file. `Window` also needs to find a different `n1`, the one we put on the command line. For this it will invoke something like `call getch("n1","i",n1out)`. Then, so the next user will know how big the output cube is, it will `call putch("n1","i",n1out)`. For more details, see the manual pages.

If we want to take input parameters from a file instead of from the command line, we type something like `<in.H Window par=myparfile.p > out.H`. The `.p` is my naming convention and is wholly optional, as is the `.H` notation for a history file.

`Sepcube` programs are self-documenting. When you type the name of the program with no input cube and no command-line arguments, you should see the self-documentation (which comes from the initial comment lines in the program).

SEP software supports “pipelining.” For example, we can slice a plane out of a data cube, make a contour plot, and display the plot, all with the command

line `<in.H Window n3=1 | Contour | Tube` where, as in UNIX pipes, the "`|`" denotes the passage of information from one program to the next. Pipelining is a convenience for the user because it saves defining a location for necessary intermediate files. The history files do flow down UNIX pipes. You may not have noticed that some location had to be assigned to the data at the intermediate stages, and when you typed the pipeline above, you were spared that clutter. To write `seplib` programs that allow pipelining, you need to read the manual page on `hclose()` to keep the history file from intermingling with the data cube itself.

A sample history file follows: this was an old one, so I removed a few anachronisms manually.

```
# Texaco Subduction Trench: read from tape by Bill Harlan
n1=1900 n2=2274
o1=2.4 it0=600 d1=.004 d2=50.  in=/d5/alaska
Window:   bill   Wed Apr 13 14:27:57 1983
                input() :    in ="/d5/alaska"
                output() : sets next in="/q2/data/Dalw"
                Input:  float   Fortran (1900,2274,1)
                Output: float   Fortran (512,128,1)
```

```
          n1=512 n2=128 n3=1
Swab:    root@mazama   Mon Feb 17 03:23:08 1986
#        input history file /r3/q2/data/Halw
                 input() :    in ="/q2/data/Dalw"
                 output() : sets next in="/q2/data/Dalw_002870_Rcp"
                 #ibs=8192 #obs=8192
Rcp: paul Mon Feb 17 03:23:15 PST 1986
         Copying from mazama:/r3/q2/data/Halw
         to hanauma:/q2/data/Halw
         in="/q2/data/Dalw"
Cp:    jon@hanauma   Wed Apr  3 23:18:13 1991
                 input() :    in ="/q2/data/Dalw"
                 output() : sets next in="/scr/jon/_junk.H@"
```

14.3. MEMORY ALLOCATION

Sepcube programs can be written in Fortran, Ratfor, or C. A serious problem with **Fortran-**77 (and hence Ratfor) is that memory cannot be allocated for arrays whose

size is determined at run time. We have worked around this limitation by using two home-grown preprocessors, one called **saw** (Stanford Auto Writer) for main programs, and one called **sat** (Stanford Auto Temporaries) for subroutines. Both preprocessors transform either Fortran or Ratfor.

14.3.1. Memory allocation in subroutines with sat

The **sat** preprocessor allows us to declare temporary arrays of arbitrary dimension, such as `temporary real*4 data(n1,n2,n3), convolution(j+k-1)` These declarations must follow other declarations and precede the executable statements.

14.3.2. The main program environment with saw

The **saw** preprocessor also calls an essential initialization routine `initpar()`, organizes the self-doc, and simplifies data-cube input. See the on-line self-documentation or the manual pages for full details. Following is a complete **saw** program for a simple task:

```
# <in.H  Scale scaleval=1. > out.H
```

```
#
#       Copy input to output and scale by scaleval
# keyword generic scale
#%
integer n1, n2, n3, esize
from history:   integer n1, n2, n3, esize
if (esize !=4)  call erexit('esize != 4')
allocate:       real x(n1,n2)
subroutine scaleit( n1,n2, x)
integer i1,i2, n1,n2
real    x(n1,n2), scaleval
from par:       real scaleval=1.
call hclose()                   # no more parameter handling.
call sreed('in', x, 4*n1*n2)
do i1=1,n1
        do i2=1,n2
                x(i1,i2) = x(i1,i2) * scaleval
call srite( 'out', x, 4*n1*n2)
```

```
return; end
```

14.4. References

Claerbout, J., 1990, Introduction to `seplib` and SEP utility software: **SEP–70**, 413–436.

Claerbout, J., 1986, A canonical program library: **SEP–50**, 281–290.

Cole, S., and Dellinger, J., Vplot: SEP's plot language: SEP**-60**, 349–389.

Dellinger, J., 1989, Why does SEP still use Vplot?: SEP–**61**, 327–335.

14.5. Acknowledgments

Robert Clayton introduced the original parameter-fetching method. I introduced history files. Stew Levin got pipes to work and brought the code to a high standard. Dave Nichols generalized it to support many computer architectures and networks of machines.

Chapter 15

Notation

The following **notation** is consistent throughout this book. Other notation defined locally as applying to an exercise or a subsection is not given here. A few symbols have several meanings (separated by semicolons), but never more than one meaning

per chapter.

15.1. OPERATORS

$\Re z$	real part of complex number z
$\Im z$	imaginary part of complex number z
E	expectation; sum over ensemble

15.2. SCALARS

n, m, N	number of components in a vector
x, y, z	Cartesian coordinates
r	radius
ϕ	phase angle
$z = x + iy = re^{i\phi}$	complex number
$\bar{z}$	complex conjugate of z

t	time; transmission coefficient
j, k	index on discrete time
f	generic function; frequency in cycles
$\omega = 2\pi f$	angular frequency (common)
$Z = e^{i\omega\Delta t}$	Z-transform variable
$*$	convolution; multiplication (in programs)
$\Delta t, dt$	sampling time interval
$\Delta f, df$	frequency sampling interval
ΔT	extent of time axis
ΔF	extent of frequency axis
ΛT	signal duration
ΛF	spectral bandwidth
σ^2	variance
c	reflection coefficient

15.3. FILTERS, SIGNALS, AND THEIR TRANSFORMS

The example $x(t)$, x_t, X_k, $X(Z)$, $X(\omega)$ can be understood as follows. A lower-case letter with a function argument (t) denotes a continuous time function (rare). Lower case with a subscript denotes a signal or filter as a function of discrete time (common). Upper case with subscript denotes a discrete Fourier transform. Z-transforms are denoted by the function argument (Z). Where a function argument ω is occasionally seen, such as in $A(\omega)$, it is generally a shorthand for $A(Z = e^{i\omega})$. For a definition of the complex conjugate of filters, see page **??**.

$a\,A$	feedback filter (autoregression)
$b\,B$	convolution filter
$c\,C$	causal filter; reflected wave; cross-spectrum
$d\,D$	downgoing wave
E	escaping wave
$f\,F$	component of layer matrix; force; generic function
$g\,G$	component of layer matrix; analytic signal; causal garbage filter
$h\,H$	admittance

I	causal integration operator
$JKLMO$	unused
N	noise
pP	phase shift; pressure; all-pass filter; generic input space
qQ	quadrature filter; generic output space
rR	impedance; reflection seismogram
sS	S is spectrum; s_t is autocorrelation
T	transmitted wave
uU	upcoming wave; logarithm of S
vV	velocity
W	weighting function; vertical component of flow
xX	generic input signal
yY	generic output signal
$\phi\Phi$	phase

15.4. MATRICES AND VECTORS

Matrices are universally denoted by upper-case boldface. Vectors are lower-case boldface everywhere except in the conjugate-gradient section of chapter 5, where vectors are capitalized when in transform space.

x	generic model space, often unknown
y	generic data space
d	data, given
A	generic matrix
B	generic matrix
$\mathbf{B}'$	conjugate transpose of generic matrix
I	identity matrix
U	unitary or pseudounitary matrix
W	weighting diagonal matrix
D	diagonal matrix
N	NMO (normal-moveout) matrix
T	tridiagonal matrix; matrix with time t on diagonal
Q	quadratic form

15.5. CHANGES FROM FGDP

In **FGDP** I used $R(Z)$ to denote a reflection seismogram, an impedance function, and a spectrum with autocorrelation coefficients r_t. I liked this classic notation, which was used by the mathematicians Wiener and Levinson. It is confusing, however, to use in one equation r_t both for the causal, one-sided, reflection seismogram and for the two-sided autocorrelation. Thus I have introduced S, which is a natural notation for spectrum, although s is admittedly less natural for autocorrelation.

Chapter 16

Interactive, 1-D, seismology program ed1D

The `ed1D` program made 23 figures for this paper book, many of them in chapters 9 and 11. In the electronic book, the caption for each of those 23 figures contains

a pushbutton that activates `ed1D` and initializes it to that figure. `ed1D` has a built-in tutorial that will enable you to operate it without this document.

`ed1D` is an interactive program that shows two one-dimensional signals related by various selectable mathematical transforms. Using a pointer, you can edit either signal and see the effect on the other one. The signals can be **Fourier-transform** pairs, or a wide variety of other pairs created by transformations such as **Hilbert transform**s, spectral **factorization**, autocorrelations, reflection coefficients, and impedance. Some of these transformations are not described in this book, but are described in chapter 8 of FGDP.

When you enter the program, you should move the pointer around the screen until you find the "Tutor" button and then click pointer buttons on it, all the while watching the message window for instructions.

You will see that there are several ways of editing a signal. First, you can use the pointer simply to draw the signal you wish. Second, you can draw a weighting function to multiply any signal that you have previously prepared. Third, there are a variety of preexisting analytic signals that you can use as weights. These mathematical weighting functions have two adjustable parameters, the shift and the **bandwidth**, which you select with the pointer. Watch the message window for

instructions for selecting these parameters.

As long as the number of ordinates is less than about 256, edited changes in one domain show up immediately in both domains. That is good for learning. With more ordinates (more computational work), you see the changes only in the domain you are editing, until later, when you move the cursor into the other domain.

The number of options in this program proved confusing to beginners, so I commented out a few in the source code. See Claerbout (1988) for more details. For example, there is a parabolic-shaped editing tool that can be pushed against any signal to deform it. The curvature of the parabola is adjustable. You can reinstall the parabolic pushing tool by uncommenting a few lines in the control panel. Another example is the huge number of transformations that can be studied with this program: I hid these, since they have no obvious interest and proved confusing to beginners.

16.1. References

Claerbout, J., 1988, Interaction with 1-D seismology: **SEP–57**, 513–522.

Chapter 17

The Zplane program

The `Zplane` program made 16 figures for this paper book. In the electronic book, each of those 16 figure captions contains a pushbutton that activates `Zplane` and initializes it to that figure. `Zplane` has a built-in tutorial that enables you to operate it

without this document. Huge gaps between the abstract and the concrete are bridged by `Zplane`. First is the conceptual gap from the time-domain representation of a filter to its poles and zeros in the complex frequency plane (as described in chapter 3). Second is a gap from the appearance of a filter to the appearance of field data after applying it. `Zplane` gives you *hands-on experience* with all these relationships. Z-plane theory conveniently incorporates causality and relates time and frequency domains. With `Zplane`, you create and move poles and zeros in the complex Z-plane. You immediately see the filter **impulse response** and its **spectrum** as you readjust the **pole**s and **zero**s. If you choose to touch a plane of seismograms, it is filtered by your chosen filter and redisplayed after a few seconds.

Choice of a display filter is important for both field data and synthetic data. Goals for filter design that are expressed in the frequency domain generally conflict with other goals in the time domain. For example, when a filter is specified by frequency cutoffs and rolloffs, then the time-domain behavior, i.e., filter length, phase shift, and energy delay, are left to fall where they may.

17.1. THE SCREEN

The program displays four planes: (1) an impulse-response graph, (2) a frequency-response graph, (3) a complex frequency plane for roots, and (4) a seismic data plane (such as a gather or section). Planes (1), (2), and (3) are line drawings or "vector plots," and they update immediately, whereas plane (4) is a variable brightness plane that updates only on command and after a delay of several seconds.

17.1.1. Complex frequency plane

A frequency-response graph displays the amplitude spectra of the current filter. On the same axes, the amplitude spectrum of a portion of data can be displayed. Further, since the horizontal axis of these spectra is the real ω-axis, it is convenient to superpose the complex ω-plane with $\Re\omega$ horizontal and scaled $\Im\omega$ vertical. The location of the pointer in the complex frequency plane is printed in the message window as the pointer moves. Theory suggests a display of the complex Z-plane. Instead I selected a complex ω-plane, because its Cartesian axes are well suited to the superposition of the amplitude spectra of filters and data.

The letters "z" and "p" are plotted in the complex ω-plane to show the locations

of poles and zeros. The location of these roots is under the exact center of the letter. You may put one letter exactly on top of another, but that only disguises the multiplicity of the root.

Recall from Z-plane theory that to keep the filter response real, any pole or zero on the positive ω-axis must have a twin on the negative ω-axis. To save screen space, I do not plot the negative axis, so you do not see the twin. Thus you need to be careful to distinguish between a root exactly at zero frequency (or at Nyquist frequency) with no twin, and a root slightly away from zero (or Nyquist) that has a twin at negative frequency (not displayed).

Let the complex frequency be decomposed into its real and imaginary parts, i.e., $\omega = \Re\omega + i\Im\omega$. All filters are required to be causal and minimum-phase—that is, all poles and zeros must be outside the unit circle in the Z-plane. Since $Z = e^{i\omega}$, the roots must all have negative values of $\Im\omega$. Any attempt to push a root to positive values of $\Im\omega$ simply leaves the root stranded on the axis of $\Im\omega = 0$. Likewise, roots can easily be placed along the edges $\Re\omega = 0$ and $\Re\omega = \pi$.

Although mathematics suggests plotting $\Im\omega$ along the vertical axis, I found it more practical to plot something like the logarithm of $\Im\omega$, because we frequently need to put poles close to the real axis. The logarithm is not exactly what we

want either, because zeros may be exactly on the unit circle. I could not devise an ideal theory for scaling $\Im\omega$. After some experimentation, I settled on $\Im\omega = -(1+y^3)/(1-y^3)$, where y is the vertical position in a window of vertical range $0 < y < 1$, but you do not need to know this since the value of ρ can be read from the message window as you move the pointer on the Z-plane.

17.1.2. The seismic data plane

The seismic data plane is displayed as wiggle traces or as raster information, i.e., gray levels, with clipped values shown in a dull red.

The “clip" value is defined as that above which a signal cannot be displayed, because the screen cannot be made brighter. To replot the filtered data with a different clip value, you touch the data in a different place. The clip is taken as 1% more than the maximum of the 30 time points surrounding the pointer.

There are no numbered axes on the data plane because none are needed. As you move the pointer across the data plane, the values of time and space are written near the ends of the axes. These values are more accurate than you could read from numbered axes.

17.1.3. Burg spectra

The **Burg** spectral method is described in FGDP. A theoretical description is not repeated in this book. The main feature of the Burg spectrum is its insensitivity to the edges of the data window of estimation.

In building the `Zplane` program, several interesting practical aspects arose. First, the program allows us to put a box on the data, and the Burg spectrum of the data in that box is computed and displayed. Thus the Burg computation of the reflection coefficients is a ratio of a numerator to a denominator, each of which is averaged in your selected box. Second, some traditional literature suggests that the only parameter you choose with the Burg spectrum is the filter length. After experimenting a while, I decided to keep the filter length at a constant 25, and instead let the variable be the corners of the estimation box that we draw on the data plane. Third, I found it necessary to bias the reflection coefficients downward as the lag approaches the data length.

17.2. References

Claerbout, J., 1987, Interactive filter design in the Z-plane: **SEP–56**, 263–271.

Index

Jon F. Claerbout (M.I.T., B.S. physics, 1960; M.S. 1963; Ph.D. geophysics, 1967), professor at Stanford University, 1967. Best Presentation Award from the Society of Exploration Geophysicists (SEG) for his paper, *Extrapolation of Wave Fields.* Honorary member and SEG Fessenden Award "in recognition of his outstanding and original pioneering work in seismic wave analysis." Founded the Stanford Exploration Project (SEP) in 1973. Elected Fellow of the American Geophysical Union. Authored three published books and five internet books. Elected to the National Academy of Engineering. Maurice Ewing Medal, SEG's highest award. Honorary Member of the European Assn. of Geoscientists & Engineers (EAGE). EAGE's highest recognition, the Erasmus Award.