INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION
INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
I.C.T.P., P.O. BOX 586, 34100 TRIESTE, ITALY, CABLE: CENTRATOM TRIESTE

SMR.693 - 1

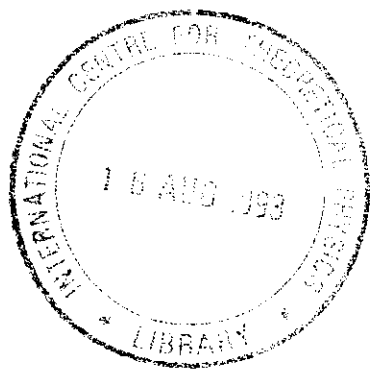
COLLEGE ON COMPUTATIONAL PHYSICS

(17 MAY - 11 JUNE 1993)

BACKGROUND MATERIAL

for lectures on

NUMERICAL TECHNIQUES



W.H. PRESS
Harvard College Observatory
Smithsonian Astrophysical Observatory
60 Garden Street
Cambridge, MA 02138
U.S.A.

These are preliminary lecture notes, intended only for distribution to participants.

Selections from

Numerical Recipes
in FORTRAN

The Art of Scientific Computing

Second Edition

William H. Press

Harvard-Smithsonian Center for Astrophysics

Saul A. Teukolsky

Department of Physics, Cornell University

William T. Vetterling

Polaroid Corporation

Brian P. Flannery

EXXON Research and Engineering Company

Reprinted for the exclusive use of

ICTP College on Computational Physics

17 May – 11 June, 1993

Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
40 West 20th Street, New York, NY 10011-4211, USA
10 Stamford Road, Oakleigh, Victoria 3166, Australia

Some sections of this book were originally published, in different form, in *Computers in Physics* magazine, Copyright © American Institute of Physics, 1988–1992.

First Edition originally published 1986; Second Edition originally published 1992.
This reprinting is corrected to software version 2.02.

Printed in the United States of America
Typeset in TEX

Library of Congress Cataloging in Publication Data

Numerical recipes in FORTRAN : the art of scientific computing / William H. Press ... [et al.]. – 2nd ed.

Includes bibliographical references (p.) and index.

ISBN 0-521-43064-X

1. Numerical analysis–Computer programs. 2. Science–Mathematics–Computer programs. 3. FORTRAN (Computer program language) I. Press, William H.
QA297.N866 1992
519.4'0285'53–dc20 92-8876

A catalog record for this book is available from the British Library.

ISBN 0 521 43064 X Book
ISBN 0 521 43721 0 Example book in FORTRAN
ISBN 0 521 43717 2 FORTRAN diskette (IBM 5.25", 1.2M)
ISBN 0 521 43719 9 FORTRAN diskette (IBM 3.5", 720K)
ISBN 0 521 43716 4 FORTRAN diskette (Mac 3.5", 800K)

Contents

4. Integration of Functions

4.0 Introduction

Numerical integration, which is also called *quadrature*, has a history extending back to the invention of calculus and before. The fact that integrals of elementary functions could not, in general, be computed analytically, while derivatives *could* be, served to give the field a certain panache, and to set it a cut above the arithmetic drudgery of numerical analysis during the whole of the 18th and 19th centuries.

With the invention of automatic computing, quadrature became just one numerical task among many, and not a very interesting one at that. Automatic computing, even the most primitive sort involving desk calculators and rooms full of "computers" (that were, until the 1950s, people rather than machines), opened to feasibility the much richer field of numerical integration of differential equations. Quadrature is merely the simplest special case: The evaluation of the integral

$$I = \int_a^b f(x)dx \tag{4.0.1}$$

is precisely equivalent to solving for the value $I \equiv y(b)$ the differential equation

$$\frac{dy}{dx} = f(x) \tag{4.0.2}$$

with the boundary condition

$$y(a) = 0 \tag{4.0.3}$$

Chapter 16 of this book deals with the numerical integration of differential equations. In that chapter, much emphasis is given to the concept of "variable" or "adaptive" choices of stepsize. We will not, therefore, develop that material here. If the function that you propose to integrate is sharply concentrated in one or more peaks, or if its shape is not readily characterized by a single length-scale, then it is likely that you should cast the problem in the form of (4.0.2)–(4.0.3) and use the methods of Chapter 16.

The quadrature methods in this chapter are based, in one way or another, on the obvious device of adding up the value of the integrand at a sequence of abscissas within the range of integration. The game is to obtain the integral as accurately as possible with the smallest number of function evaluations of the integrand. Just as in the case of interpolation (Chapter 3), one has the freedom to choose methods

of various *orders*, with higher order sometimes, but not always, giving higher accuracy. "Romberg integration," which is discussed in §4.3, is a general formalism for making use of integration methods of a variety of different orders, and we recommend it highly.

Apart from the methods of this chapter and of Chapter 16, there are yet other methods for obtaining integrals. One important class is based on function approximation. We discuss explicitly the integration of functions by Chebyshev approximation ("Clenshaw-Curtis" quadrature) in §5.9. Although not explicitly discussed here, you ought to be able to figure out how to do *cubic spline quadrature* using the output of the routine `spline` in §3.3. (Hint: Integrate equation 3.3.3 over x analytically. See [1].)

Some integrals related to Fourier transforms can be calculated using the fast Fourier transform (FFT) algorithm. This is discussed in §13.9.

Multidimensional integrals are another whole multidimensional bag of worms. Section 4.6 is an introductory discussion in this chapter; the important technique of *Monte-Carlo integration* is treated in Chapter 7.

CITED REFERENCES AND FURTHER READING:

Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley), Chapter 2.

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley), Chapter 7.

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 4.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 3.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), Chapter 4.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §7.4.

Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 5.

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §5.2, p. 89. [1]

Davis, P., and Rabinowitz, P. 1984, *Methods of Numerical Integration*, 2nd ed. (Orlando, FL: Academic Press).

4.1 Classical Formulas for Equally Spaced Abscissas

Where would any book on numerical analysis be without Mr. Simpson and his "rule"? The classical formulas for integrating a function whose value is known at equally spaced steps have a certain elegance about them, and they are redolent with historical association. Through them, the modern numerical analyst communes with the spirits of his or her predecessors back across the centuries, as far as the time of Newton, if not farther. Alas, times *do* change; with the exception of two of the most modest formulas ("extended trapezoidal rule," equation 4.1.11, and "extended

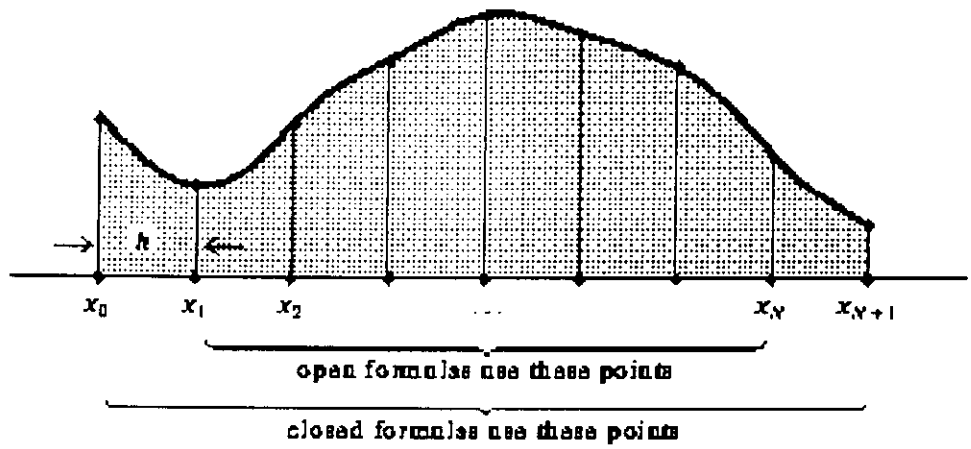


Figure 4.1.1. Quadrature formulas with equally spaced abscissas compute the integral of a function between x_0 and x_{N+1}. Closed formulas evaluate the function on the boundary points, while open formulas refrain from doing so (useful if the evaluation algorithm breaks down on the boundary points).

midpoint rule," equation 4.1.19, sec §4.2), the classical formulas are almost entirely useless. They are museum pieces, but beautiful ones.

Some notation: We have a sequence of abscissas, denoted $x_0, x_1, \ldots, x_N, x_{N+1}$ which are spaced apart by a constant step h,

$$x_i = x_0 + ih \qquad i = 0, 1, \ldots, N+1 \tag{4.1.1}$$

A function $f(x)$ has known values at the x_i's,

$$f(x_i) \equiv f_i \tag{4.1.2}$$

We want to integrate the function $f(x)$ between a lower limit a and an upper limit b, where a and b are each equal to one or the other of the x_i's. An integration formula that uses the value of the function at the endpoints, $f(a)$ or $f(b)$, is called a *closed* formula. Occasionally, we want to integrate a function whose value at one or both endpoints is difficult to compute (e.g., the computation of f goes to a limit of zero over zero there, or worse yet has an integrable singularity there). In this case we want an *open* formula, which estimates the integral using only x_i's strictly *between* a and b (see Figure 4.1.1).

The basic building blocks of the classical formulas are rules for integrating a function over a small number of intervals. As that number increases, we can find rules that are exact for polynomials of increasingly high order. (Keep in mind that higher order does not always imply higher accuracy in real cases.) A sequence of such closed formulas is now given.

Closed Newton-Cotes Formulas

Trapezoidal rule:

$$\int_{x_1}^{x_2} f(x)dx = h\left[\frac{1}{2}f_1 + \frac{1}{2}f_2\right] \quad + O(h^3 f'') \tag{4.1.3}$$

Here the error term $O(\ )$ signifies that the true answer differs from the estimate by an amount that is the product of some numerical coefficient times h^3 times the value

of the function's second derivative somewhere in the interval of integration. The coefficient is knowable, and it can be found in all the standard references on this subject. The point at which the second derivative is to be evaluated is, however, unknowable. If we knew it, we could evaluate the function there and have a higher-order method! Since the product of a knowable and an unknowable is unknowable, we will streamline our formulas and write only $O(\ )$, instead of the coefficient.

Equation (4.1.3) is a two-point formula (x_1 and x_2). It is exact for polynomials up to and including degree 1, i.e., $f(x) = x$. One anticipates that there is a three-point formula exact up to polynomials of degree 2. This is true; moreover, by a cancellation of coefficients due to left-right symmetry of the formula, the three-point formula is exact for polynomials up to and including degree 3, i.e., $f(x) = x^3$:

Simpson's rule:

$$\int_{x_1}^{x_3} f(x)dx = h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{1}{3}f_3\right] \quad + O(h^5 f^{(4)}) \tag{4.1.4}$$

Here $f^{(4)}$ means the fourth derivative of the function f evaluated at an unknown place in the interval. Note also that the formula gives the integral over an interval of size $2h$, so the coefficients add up to 2.

There is no lucky cancellation in the four-point formula, so it is also exact for polynomials up to and including degree 3.

Simpson's $\frac{3}{8}$ rule:

$$\int_{x_1}^{x_4} f(x)dx = h\left[\frac{3}{8}f_1 + \frac{9}{8}f_2 + \frac{9}{8}f_3 + \frac{3}{8}f_4\right] \quad + O(h^5 f^{(4)}) \tag{4.1.5}$$

The five-point formula again benefits from a cancellation:

Bode's rule:

$$\int_{x_1}^{x_5} f(x)dx = h\left[\frac{14}{45}f_1 + \frac{64}{45}f_2 + \frac{24}{45}f_3 + \frac{64}{45}f_4 + \frac{14}{45}f_5\right] \quad + O(h^7 f^{(6)}) \tag{4.1.6}$$

This is exact for polynomials up to and including degree 5.

At this point the formulas stop being named after famous personages, so we will not go any further. Consult [1] for additional formulas in the sequence.

Extrapolative Formulas for a Single Interval

We are going to depart from historical practice for a moment. Many texts would give, at this point, a sequence of "Newton-Cotes Formulas of Open Type." Here is an example:

$$\int_{x_0}^{x_5} f(x)dx = h\left[\frac{55}{24}f_1 + \frac{5}{24}f_2 + \frac{5}{24}f_3 + \frac{55}{24}f_4\right] \quad + O(h^5 f^{(4)})$$

Notice that the integral from $a = x_0$ to $b = x_5$ is estimated, using only the interior points x_1, x_2, x_3, x_4. In our opinion, formulas of this type are not useful for the reasons that (i) they cannot usefully be strung together to get "extended" rules, as we are about to do with the closed formulas, and (ii) for all other possible uses they are dominated by the Gaussian integration formulas which we will introduce in §4.5.

Instead of the Newton-Cotes open formulas, let us set out the formulas for estimating the integral in the single interval from x_0 to x_1, using values of the function f at $x_1, x_2, \ldots$. These will be useful building blocks for the "extended" open formulas.

$$\int_{x_0}^{x_1} f(x)dx = h[f_1] \quad + O(h^2 f') \tag{4.1.7}$$

$$\int_{x_0}^{x_1} f(x)dx = h\left[\frac{3}{2}f_1 - \frac{1}{2}f_2\right] \quad + O(h^3 f'') \tag{4.1.8}$$

$$\int_{x_0}^{x_1} f(x)dx = h\left[\frac{23}{12}f_1 - \frac{16}{12}f_2 + \frac{5}{12}f_3\right] \quad + O(h^4 f^{(3)}) \tag{4.1.9}$$

$$\int_{x_0}^{x_1} f(x)dx = h\left[\frac{55}{24}f_1 - \frac{59}{24}f_2 + \frac{37}{24}f_3 - \frac{9}{24}f_4\right] \quad + O(h^5 f^{(4)}) \tag{4.1.10}$$

Perhaps a word here would be in order about how formulas like the above can be derived. There are elegant ways, but the most straightforward is to write down the basic form of the formula, replacing the numerical coefficients with unknowns, say p, q, r, s. Without loss of generality take $x_0 = 0$ and $x_1 = 1$, so $h = 1$. Substitute in turn for $f(x)$ (and for f_1, f_2, f_3, f_4) the functions $f(x) = 1$, $f(x) = x$, $f(x) = x^2$, and $f(x) = x^3$. Doing the integral in each case reduces the left-hand side to a number, and the right-hand side to a linear equation for the unknowns p, q, r, s. Solving the four equations produced in this way gives the coefficients.

Extended Formulas (Closed)

If we use equation (4.1.3) $N - 1$ times, to do the integration in the intervals $(x_1, x_2), (x_2, x_3), \ldots, (x_{N-1}, x_N)$, and then add the results, we obtain an "extended" or "composite" formula for the integral from x_1 to x_N.

Extended trapezoidal rule:

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{1}{2}f_1 + f_2 + f_3 + \cdots + f_{N-1} + \frac{1}{2}f_N\right] \quad + O\left(\frac{(b-a)^3 f''}{N^2}\right) \tag{4.1.11}$$

Here we have written the error estimate in terms of the interval $b - a$ and the number of points N instead of in terms of h. This is clearer, since one is usually holding a and b fixed and wanting to know (e.g.) how much the error will be decreased

by taking twice as many steps (in this case, it is by a factor of 4). In subsequent equations we will show *only* the scaling of the error term with the number of steps.

For reasons that will not become clear until §4.2, equation (4.1.11) is in fact the most important equation in this section, the basis for most practical quadrature schemes.

The *extended formula of order* $1/N^3$ is:

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{5}{12}f_1 + \frac{13}{12}f_2 + f_3 + f_4 + \cdots + f_{N-2} + \frac{13}{12}f_{N-1} + \frac{5}{12}f_N\right] + O\left(\frac{1}{N^3}\right) \tag{4.1.12}$$

(We will see in a moment where this comes from.)

If we apply equation (4.1.4) to successive, nonoverlapping *pairs* of intervals, we get the *extended Simpson's rule:*

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{2}{3}f_3 + \frac{4}{3}f_4 + \cdots + \frac{2}{3}f_{N-2} + \frac{4}{3}f_{N-1} + \frac{1}{3}f_N\right] + O\left(\frac{1}{N^4}\right) \tag{4.1.13}$$

Notice that the 2/3, 4/3 alternation continues throughout the interior of the evaluation. Many people believe that the wobbling alternation somehow contains deep information about the integral of their function that is not apparent to mortal eyes. In fact, the alternation is an artifact of using the building block (4.1.4). Another extended formula with the same order as Simpson's rule is

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{3}{8}f_1 + \frac{7}{6}f_2 + \frac{23}{24}f_3 + f_4 + f_5 + \cdots + f_{N-4} + f_{N-3} + \frac{23}{24}f_{N-2} + \frac{7}{6}f_{N-1} + \frac{3}{8}f_N\right] + O\left(\frac{1}{N^4}\right) \tag{4.1.14}$$

This equation is constructed by fitting cubic polynomials through successive groups of four points; we defer details to §18.3, where a similar technique is used in the solution of integral equations. We can, however, tell you where equation (4.1.12) came from. It is Simpson's extended rule, averaged with a modified version of itself in which the first and last step are done with the trapezoidal rule (4.1.3). The trapezoidal step is *two* orders lower than Simpson's rule; however, its contribution to the integral goes down as an additional power of N (since it is used only twice, not N times). This makes the resulting formula of degree *one* less than Simpson.

Extended Formulas (Open and Semi-open)

We can construct open and semi-open extended formulas by adding the closed formulas (4.1.11)–(4.1.14), evaluated for the second and subsequent steps, to the extrapolative open formulas for the first step, (4.1.7)–(4.1.10). As discussed immediately above, it is consistent to use an end step that is of one order lower than the (repeated) interior step. The resulting formulas for an interval open at both ends are as follows:

Equations (4.1.7) and (4.1.11) give

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{3}{2}f_2 + f_3 + f_4 + \cdots + f_{N-2} + \frac{3}{2}f_{N-1}\right] \quad + O\left(\frac{1}{N^2}\right) \tag{4.1.15}$$

Equations (4.1.8) and (4.1.12) give

$$\begin{aligned}\int_{x_1}^{x_N} f(x)dx = h\Bigg[&\frac{23}{12}f_2 + \frac{7}{12}f_3 + f_4 + f_5 + \\ &\cdots + f_{N-3} + \frac{7}{12}f_{N-2} + \frac{23}{12}f_{N-1}\Bigg] \\ &+ O\left(\frac{1}{N^3}\right)\end{aligned} \tag{4.1.16}$$

Equations (4.1.9) and (4.1.13) give

$$\begin{aligned}\int_{x_1}^{x_N} f(x)dx = h\Bigg[&\frac{27}{12}f_2 + 0 + \frac{13}{12}f_4 + \frac{4}{3}f_5 + \\ &\cdots + \frac{4}{3}f_{N-4} + \frac{13}{12}f_{N-3} + 0 + \frac{27}{12}f_{N-1}\Bigg] \\ &+ O\left(\frac{1}{N^4}\right)\end{aligned} \tag{4.1.17}$$

The interior points alternate 4/3 and 2/3. If we want to avoid this alternation, we can combine equations (4.1.9) and (4.1.14), giving

$$\begin{aligned}\int_{x_1}^{x_N} f(x)dx = h\Bigg[&\frac{55}{24}f_2 - \frac{1}{6}f_3 + \frac{11}{8}f_4 + f_5 + f_6 + f_7 + \\ &\cdots + f_{N-5} + f_{N-4} + \frac{11}{8}f_{N-3} - \frac{1}{6}f_{N-2} + \frac{55}{24}f_{N-1}\Bigg] \\ &+ O\left(\frac{1}{N^4}\right)\end{aligned} \tag{4.1.18}$$

We should mention in passing another extended open formula, for use where the limits of integration are located halfway between tabulated abscissas. This one is known as the *extended midpoint rule*, and is accurate to the same order as (4.1.15):

$$\begin{aligned}\int_{x_1}^{x_N} f(x)dx = h[&f_{3/2} + f_{5/2} + f_{7/2} + \\ &\cdots + f_{N-3/2} + f_{N-1/2}] \quad + O\left(\frac{1}{N^2}\right)\end{aligned} \tag{4.1.19}$$

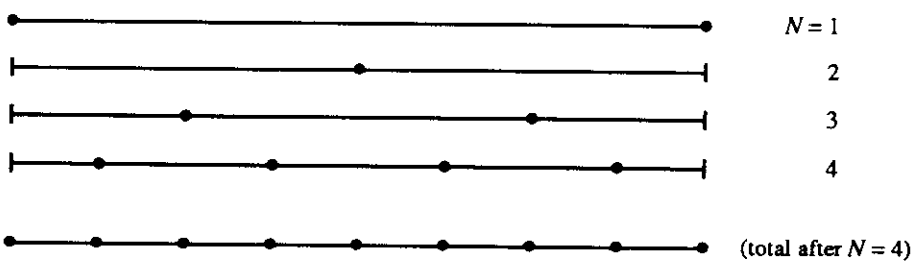


Figure 4.2.1. Sequential calls to the routine `trapzd` incorporate the information from previous calls and evaluate the integrand only at those new points necessary to refine the grid. The bottom line shows the totality of function evaluations after the fourth call. The routine `qsimp`, by weighting the intermediate results, transforms the trapezoid rule into Simpson's rule with essentially no additional overhead.

There are also formulas of higher order for this situation, but we will refrain from giving them.

The *semi-open formulas* are just the obvious combinations of equations (4.1.11)–(4.1.14) with (4.1.15)–(4.1.18), respectively. At the closed end of the integration, use the weights from the former equations; at the open end use the weights from the latter equations. One example should give the idea, the formula with error term decreasing as $1/N^3$ which is closed on the right and open on the left:

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{23}{12}f_2 + \frac{7}{12}f_3 + f_4 + f_5 + \cdots + f_{N-2} + \frac{13}{12}f_{N-1} + \frac{5}{12}f_N\right] + O\left(\frac{1}{N^3}\right) \tag{4.1.20}$$

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, vol. 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.4. [1]

Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley), §7.1.

4.2 Elementary Algorithms

Our starting point is equation (4.1.11), the extended trapezoidal rule. There are two facts about the trapezoidal rule which make it the starting point for a variety of algorithms. One fact is rather obvious, while the second is rather "deep."

The obvious fact is that, for a fixed function $f(x)$ to be integrated between fixed limits a and b, one can double the number of intervals in the extended trapezoidal rule without losing the benefit of previous work. The coarsest implementation of the trapezoidal rule is to average the function at its endpoints a and b. The first stage of refinement is to add to this average the value of the function at the halfway point. The second stage of refinement is to add the values at the 1/4 and 3/4 points. And so on (see Figure 4.2.1).

Without further ado we can write a routine with this kind of logic to it:

```
      SUBROUTINE trapzd(func,a,b,s,n)
      INTEGER n
      REAL a,b,s,func
      EXTERNAL func
          This routine computes the nth stage of refinement of an extended trapezoidal rule. func is
          input as the name of the function to be integrated between limits a and b, also input. When
          called with n=1, the routine returns as s the crudest estimate of ∫_a^b f(x)dx. Subsequent
          calls with n=2,3,... (in that sequential order) will improve the accuracy of s by adding 2^(n-2)
          additional interior points. s should not be modified between sequential calls.
      INTEGER it,j
      REAL del,sum,tnm,x
      if (n.eq.1) then
          s=0.5*(b-a)*(func(a)+func(b))
      else
          it=2**(n-2)
          tnm=it
          del=(b-a)/tnm                  This is the spacing of the points to be added.
          x=a+0.5*del
          sum=0.
          do 11 j=1,it
              sum=sum+func(x)
              x=x+del
          enddo 11
          s=0.5*(s+(b-a)*sum/tnm)        This replaces s by its refined value.
      endif
      return
      END
```

The above routine (`trapzd`) is a workhorse that can be harnessed in several ways. The simplest and crudest is to integrate a function by the extended trapezoidal rule where you know in advance (we can't imagine how!) the number of steps you want. If you want $2^M + 1$, you can accomplish this by the fragment

```
      do 11 j=1,m+1
          call trapzd(func,a,b,s,j)
      enddo 11
```

with the answer returned as `s`.

Much better, of course, is to refine the trapezoidal rule until some specified degree of accuracy has been achieved:

```
      SUBROUTINE qtrap(func,a,b,s)
      INTEGER JMAX
      REAL a,b,func,s,EPS
      EXTERNAL func
      PARAMETER (EPS=1.e-6, JMAX=20)
C     USES trapzd
          Returns as s the integral of the function func from a to b. The parameters EPS can be set
          to the desired fractional accuracy and JMAX so that 2 to the power JMAX-1 is the maximum
          allowed number of steps. Integration is performed by the trapezoidal rule.
      INTEGER j
      REAL olds
      olds=-1.e30                        Any number that is unlikely to be the average of the function
      do 11 j=1,JMAX                        at its endpoints will do here.
          call trapzd(func,a,b,s,j)
          if (abs(s-olds).lt.EPS*abs(olds)) return
          olds=s
      enddo 11
      pause 'too many steps in qtrap'
      END
```

Unsophisticated as it is, routine `qtrap` is in fact a fairly robust way of doing integrals of functions that are not very smooth. Increased sophistication will usually translate into a higher-order method whose efficiency will be greater only for sufficiently smooth integrands. `qtrap` is the method of choice, e.g., for an integrand which is a function of a variable that is linearly interpolated between measured data points. Be sure that you do not require too stringent an EPS, however: If `qtrap` takes too many steps in trying to achieve your required accuracy, accumulated roundoff errors may start increasing, and the routine may never converge. A value 10^{-6} is just on the edge of trouble for most 32-bit machines; it is achievable when the convergence is moderately rapid, but not otherwise.

We come now to the "deep" fact about the extended trapezoidal rule, equation (4.1.11). It is this: The error of the approximation, which begins with a term of order $1/N^2$, is in fact *entirely even* when expressed in powers of $1/N$. This follows directly from the *Euler-Maclaurin Summation Formula*,

$$\int_{x_1}^{x_N} f(x)dx = h\left[\frac{1}{2}f_1 + f_2 + f_3 + \cdots + f_{N-1} + \frac{1}{2}f_N\right] - \frac{B_2h^2}{2!}(f'_N - f'_1) - \cdots - \frac{B_{2k}h^{2k}}{(2k)!}(f_N^{(2k-1)} - f_1^{(2k-1)}) - \cdots \tag{4.2.1}$$

Here B_{2k} is a *Bernoulli number*, defined by the generating function

$$\frac{t}{e^t - 1} = \sum_{n=0}^{\infty} B_n \frac{t^n}{n!} \tag{4.2.2}$$

with the first few even values (odd values vanish except for $B_1 = -1/2$)

$$B_0 = 1 \quad B_2 = \frac{1}{6} \quad B_4 = -\frac{1}{30} \quad B_6 = \frac{1}{42}$$
$$B_8 = -\frac{1}{30} \quad B_{10} = \frac{5}{66} \quad B_{12} = -\frac{691}{2730} \tag{4.2.3}$$

Equation (4.2.1) is not a convergent expansion, but rather only an asymptotic expansion whose error when truncated at any point is always less than twice the magnitude of the first neglected term. The reason that it is not convergent is that the Bernoulli numbers become very large, e.g.,

$$B_{50} = \frac{495057205241079648212477525}{66}$$

The key point is that only even powers of h occur in the error series of (4.2.1). This fact is not, in general, shared by the higher-order quadrature rules in §4.1. For example, equation (4.1.12) has an error series beginning with $O(1/N^3)$, but continuing with all subsequent powers of N: $1/N^4$, $1/N^5$, etc.

Suppose we evaluate (4.1.11) with N steps, getting a result S_N, and then again with $2N$ steps, getting a result S_{2N}. (This is done by any two consecutive calls of

trapzd.) The leading error term in the second evaluation will be 1/4 the size of the error in the first evaluation. Therefore the combination

$$S = \frac{4}{3}S_{2N} - \frac{1}{3}S_N \tag{4.2.4}$$

will cancel out the leading order error term. But there *is* no error term of order $1/N^3$, by (4.2.1). The surviving error is of order $1/N^4$, the same as Simpson's rule. In fact, it should not take long for you to see that (4.2.4) is *exactly* Simpson's rule (4.1.13), alternating 2/3's, 4/3's, and all. This is the preferred method for evaluating that rule, and we can write it as a routine exactly analogous to qtrap above:

```
      SUBROUTINE qsimp(func,a,b,s)
      INTEGER JMAX
      REAL a,b,func,s,EPS
      EXTERNAL func
      PARAMETER (EPS=1.e-6, JMAX=20)
C     USES trapzd
        Returns as s the integral of the function func from a to b. The parameters EPS can be set
        to the desired fractional accuracy and JMAX so that 2 to the power JMAX-1 is the maximum
        allowed number of steps. Integration is performed by Simpson's rule.
      INTEGER j
      REAL os,ost,st
      ost=-1.e30
      os= -1.e30
      do 11 j=1,JMAX
        call trapzd(func,a,b,st,j)
        s=(4.*st-ost)/3.                    Compare equation (4.2.4), above.
        if (abs(s-os).lt.EPS*abs(os)) return
        os=s
        ost=st
      enddo 11
      pause 'too many steps in qsimp'
      END
```

The routine qsimp will in general be more efficient than qtrap (i.e., require fewer function evaluations) when the function to be integrated has a finite 4th derivative (i.e., a continuous 3rd derivative). The combination of qsimp and its necessary workhorse trapzd is a good one for light-duty work.

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §3.3.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §§7.4.1–7.4.2.

Forsythe, G.E., Malcolm, M.A., and Moler, C.B. 1977, *Computer Methods for Mathematical Computations* (Englewood Cliffs, NJ: Prentice-Hall), §5.3.

4.3 Romberg Integration

We can view Romberg's method as the natural generalization of the routine `qsimp` in the last section to integration schemes that are of higher order than Simpson's rule. The basic idea is to use the results from k successive refinements of the extended trapezoidal rule (implemented in `trapzd`) to remove all terms in the error series up to but not including $O(1/N^{2k})$. The routine `qsimp` is the case of $k = 2$. This is one example of a very general idea that goes by the name of *Richardson's deferred approach to the limit*: Perform some numerical algorithm for various values of a parameter h, and then extrapolate the result to the continuum limit $h = 0$.

Equation (4.2.4), which subtracts off the leading error term, is a special case of polynomial extrapolation. In the more general Romberg case, we can use Neville's algorithm (see §3.1) to extrapolate the successive refinements to zero stepsize. Neville's algorithm can in fact be coded very concisely within a Romberg integration routine. For clarity of the program, however, it seems better to do the extrapolation by subroutine call to `polint`, already given in §3.1.

```
      SUBROUTINE qromb(func,a,b,ss)
      INTEGER JMAX,JMAXP,K,KM
      REAL a,b,func,ss,EPS
      EXTERNAL func
      PARAMETER (EPS=1.e-6, JMAX=20, JMAXP=JMAX+1, K=5, KM=K-1)
C     USES polint,trapzd
        Returns as ss the integral of the function func from a to b. Integration is performed by
        Romberg's method of order 2K, where, e.g., K=2 is Simpson's rule.
        Parameters: EPS is the fractional accuracy desired, as determined by the extrapolation
        error estimate; JMAX limits the total number of steps; K is the number of points used in
        the extrapolation.
      INTEGER j
      REAL dss,h(JMAXP),s(JMAXP)          These store the successive trapezoidal approximations
      h(1)=1.                               and their relative stepsizes.
      do 11 j=1,JMAX
        call trapzd(func,a,b,s(j),j)
        if (j.ge.K) then
          call polint(h(j-KM),s(j-KM),K,0.,ss,dss)
          if (abs(dss).le.EPS*abs(ss)) return
        endif
        s(j+1)=s(j)
        h(j+1)=0.25*h(j)                  This is a key step: The factor is 0.25 even though
      enddo 11                              the stepsize is decreased by only 0.5. This makes
      pause 'too many steps in qromb'       the extrapolation a polynomial in h^2 as allowed
      END                                   by equation (4.2.1), not just a polynomial in h.
```

The routine `qromb`, along with its required `trapzd` and `polint`, is quite powerful for sufficiently smooth (e.g., analytic) integrands, integrated over intervals which contain no singularities, and where the endpoints are also nonsingular. `qromb`, in such circumstances, takes many, *many* fewer function evaluations than either of the routines in §4.2. For example, the integral

$$\int_0^2 x^4 \log(x + \sqrt{x^2 + 1})dx$$

converges (with parameters as shown above) on the very first extrapolation, after just 5 calls to `trapzd`, while `qsimp` requires 8 calls (8 times as many evaluations of the integrand) and `qtrap` requires 13 calls (making 256 times as many evaluations of the integrand).

CITED REFERENCES AND FURTHER READING:

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §§3.4–3.5.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §§7.4.1–7.4.2.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §4.10–2.

4.4 Improper Integrals

For our present purposes, an integral will be "improper" if it has any of the following problems:

- its integrand goes to a finite limiting value at finite upper and lower limits, but cannot be evaluated *right on* one of those limits (e.g., $\sin x/x$ at $x = 0$)
- its upper limit is ∞ , or its lower limit is $-\infty$
- it has an integrable singularity at either limit (e.g., $x^{-1/2}$ at $x = 0$)
- it has an integrable singularity at a known place between its upper and lower limits
- it has an integrable singularity at an unknown place between its upper and lower limits

If an integral is infinite (e.g., $\int_1^\infty x^{-1}dx$), or does not exist in a limiting sense (e.g., $\int_{-\infty}^{\infty} \cos x dx$), we do not call it improper; we call it impossible. No amount of clever algorithmics will return a meaningful answer to an ill-posed problem.

In this section we will generalize the techniques of the preceding two sections to cover the first four problems on the above list. A more advanced discussion of quadrature with integrable singularities occurs in Chapter 18, notably §18.3. The fifth problem, singularity at unknown location, can really only be handled by the use of a variable stepsize differential equation integration routine, as will be given in Chapter 16.

We need a workhorse like the extended trapezoidal rule (equation 4.1.11), but one which is an *open* formula in the sense of §4.1, i.e., does not require the integrand to be evaluated at the endpoints. Equation (4.1.19), the extended midpoint rule, is the best choice. The reason is that (4.1.19) shares with (4.1.11) the "deep" property of having an error series that is entirely even in h. Indeed there is a formula, not as well known as it ought to be, called the *Second Euler-Maclaurin summation formula*,

$$\begin{aligned}\int_{x_1}^{x_N} f(x)dx = h[f_{3/2} + f_{5/2} + f_{7/2} + \cdots + f_{N-3/2} + f_{N-1/2}] \\ + \frac{B_2 h^2}{4}(f'_N - f'_1) + \cdots \\ + \frac{B_{2k} h^{2k}}{(2k)!}(1 - 2^{-2k+1})(f_N^{(2k-1)} - f_1^{(2k-1)}) + \cdots\end{aligned} \tag{4.4.1}$$

This equation can be derived by writing out (4.2.1) with stepsize h, then writing it out again with stepsize $h/2$, then subtracting the first from twice the second.

It is not possible to double the number of steps in the extended midpoint rule and still have the benefit of previous function evaluations (try it!). However, it is possible to *triple* the number of steps and do so. Shall we do this, or double and accept the loss? On the average, tripling does a factor $\sqrt{3}$ of unnecessary work, since the "right" number of steps for a desired accuracy criterion may in fact fall anywhere in the logarithmic interval implied by tripling. For doubling, the factor is only $\sqrt{2}$, but we lose an extra factor of 2 in being unable to use all the previous evaluations. Since $1.732 < 2 \times 1.414$, it is better to triple.

Here is the resulting routine, which is directly comparable to `trapzd`.

```
      SUBROUTINE midpnt(func,a,b,s,n)
      INTEGER n
      REAL a,b,s,func
      EXTERNAL func
        This routine computes the nth stage of refinement of an extended midpoint rule. func is
        input as the name of the function to be integrated between limits a and b, also input. When
        called with n=1, the routine returns as s the crudest estimate of ∫_a^b f(x)dx. Subsequent
        calls with n=2,3,... (in that sequential order) will improve the accuracy of s by adding
        (2/3) × 3^(n-1) additional interior points. s should not be modified between sequential calls.
      INTEGER it,j
      REAL ddel,del,sum,tnm,x
      if (n.eq.1) then
        s=(b-a)*func(0.5*(a+b))
      else
        it=3**(n-2)
        tnm=it
        del=(b-a)/(3.*tnm)
        ddel=del+del                    The added points alternate in spacing between del and ddel.
        x=a+0.5*del
        sum=0.
        do 11 j=1,it
          sum=sum+func(x)
          x=x+ddel
          sum=sum+func(x)
          x=x+del
        enddo 11
        s=(s+(b-a)*sum/tnm)/3.          The new sum is combined with the old integral to give a
      endif                                 refined integral.
      return
      END
```

The routine `midpnt` can exactly replace `trapzd` in a driver routine like `qtrap` (§4.2); one simply changes `call trapzd` to `call midpnt`, and perhaps also decreases the parameter `JMAX` since $3^{\mathtt{JMAX}-1}$ (from step tripling) is a much larger number than $2^{\mathtt{JMAX}-1}$ (step doubling).

The open formula implementation analogous to Simpson's rule (`qsimp` in §4.2) substitutes `midpnt` for `trapzd` and decreases `JMAX` as above, but now also changes the extrapolation step to be

```
        s=(9.*st-ost)/8.
```

since, when the number of steps is tripled, the error decreases to 1/9th its size, not 1/4th as with step doubling.

Either the modified `qtrap` or the modified `qsimp` will fix the first problem on the list at the beginning of this section. Yet more sophisticated is to generalize Romberg integration in like manner:

```
      SUBROUTINE qromo(func,a,b,ss,choose)
      INTEGER JMAX,JMAXP,K,KM
      REAL a,b,func,ss,EPS
      EXTERNAL func,choose
      PARAMETER (EPS=1.e-6, JMAX=14, JMAXP=JMAX+1, K=5, KM=K-1)
C     USES polint
        Romberg integration on an open interval. Returns as ss the integral of the function func
        from a to b, using any specified integrating subroutine choose and Romberg's method.
        Normally choose will be an open formula, not evaluating the function at the endpoints. It
        is assumed that choose triples the number of steps on each call, and that its error series
        contains only even powers of the number of steps. The routines midpnt, midinf, midsql,
        midsqu, are possible choices for choose. The parameters have the same meaning as in
        qromb.
      INTEGER j
      REAL dss,h(JMAXP),s(JMAXP)
      h(1)=1.
      do 11 j=1,JMAX
        call choose(func,a,b,s(j),j)
        if (j.ge.K) then
          call polint(h(j-KM),s(j-KM),K,0.,ss,dss)
          if (abs(dss).le.EPS*abs(ss)) return
        endif
        s(j+1)=s(j)
        h(j+1)=h(j)/9.          This is where the assumption of step tripling and an even
      enddo 11                    error series is used.
      pause 'too many steps in qromo'
      END
```

The differences between `qromo` and `qromb` (§4.3) are so slight that it is perhaps gratuitous to list `qromo` in full. It, however, is an excellent driver routine for solving all the other problems of improper integrals in our first list (except the intractable fifth), as we shall now see.

The basic trick for improper integrals is to make a change of variables to eliminate the singularity, or to map an infinite range of integration to a finite one. For example, the identity

$$\int_a^b f(x)dx = \int_{1/b}^{1/a} \frac{1}{t^2} f\left(\frac{1}{t}\right) dt \qquad ab > 0 \tag{4.4.2}$$

can be used with *either* $b \to \infty$ and a positive, *or* with $a \to -\infty$ and b negative, and works for any function which decreases towards infinity faster than $1/x^2$.

You can make the change of variable implied by (4.4.2) either analytically and then use (e.g.) `qromo` and `midpnt` to do the numerical evaluation, *or* you can let the numerical algorithm make the change of variable for you. We prefer the latter method as being more transparent to the user. To implement equation (4.4.2) we simply write a modified version of `midpnt`, called `midinf`, which allows b to be infinite (or, more precisely, a very large number on your particular machine, such as 1×10^{30}), or a to be negative and infinite.

```
SUBROUTINE midinf(funk,aa,bb,s,n)
INTEGER n
REAL aa,bb,s,funk
EXTERNAL funk
    This routine is an exact replacement for midpnt, i.e., returns as s the nth stage of refinement
    of the integral of funk from aa to bb, except that the function is evaluated at evenly spaced
    points in 1/x rather than in x. This allows the upper limit bb to be as large and positive
    as the computer allows, or the lower limit aa to be as large and negative, but not both.
    aa and bb must have the same sign.
INTEGER it,j
REAL a,b,ddel,del,sum,tnm,func,x
func(x)=funk(1./x)/x**2          This statement function effects the change of variable.
b=1./aa                          These two statements change the limits of integration ac-
a=1./bb                              cordingly.
if (n.eq.1) then                 From this point on, the routine is exactly identical to midpnt.
    s=(b-a)*func(0.5*(a+b))
else
    it=3**(n-2)
    tnm=it
    del=(b-a)/(3.*tnm)
    ddel=del+del
    x=a+0.5*del
    sum=0.
    do 11 j=1,it
        sum=sum+func(x)
        x=x+ddel
        sum=sum+func(x)
        x=x+del
    enddo 11
    s=(s+(b-a)*sum/tnm)/3.
endif
return
END
```

If you need to integrate from a negative lower limit to positive infinity, you do this by breaking the integral into two pieces at some positive value, for example,

```
call qromo(funk,-5.,2.,s1,midpnt)
call qromo(funk,2.,1.e30,s2,midinf)
answer=s1+s2
```

Where should you choose the breakpoint? At a sufficiently large positive value so that the function `funk` is at least beginning to approach its asymptotic decrease to zero value at infinity. The polynomial extrapolation implicit in the second call to `qromo` deals with a polynomial in $1/x$, not in x.

To deal with an integral that has an integrable power-law singularity at its lower limit, one also makes a change of variable. If the integrand diverges as $(x-a)^\gamma$, $0 \le \gamma < 1$, near $x = a$, use the identity

$$\int_a^b f(x)dx = \frac{1}{1-\gamma}\int_0^{(b-a)^{1-\gamma}} t^{\frac{\gamma}{1-\gamma}} f(t^{\frac{1}{1-\gamma}} + a)dt \qquad (b > a) \qquad (4.4.3)$$

If the singularity is at the upper limit, use the identity

$$\int_a^b f(x)dx = \frac{1}{1-\gamma}\int_0^{(b-a)^{1-\gamma}} t^{\frac{\gamma}{1-\gamma}} f(b - t^{\frac{1}{1-\gamma}})dt \qquad (b > a) \qquad (4.4.4)$$

If there is a singularity at both limits, divide the integral at an interior breakpoint as in the example above.

Equations (4.4.3) and (4.4.4) are particularly simple in the case of inverse square-root singularities, a case that occurs frequently in practice:

$$\int_a^b f(x)dx = \int_0^{\sqrt{b-a}} 2tf(a+t^2)dt \qquad (b>a) \tag{4.4.5}$$

for a singularity at a, and

$$\int_a^b f(x)dx = \int_0^{\sqrt{b-a}} 2tf(b-t^2)dt \qquad (b>a) \tag{4.4.6}$$

for a singularity at b. Once again, we can implement these changes of variable transparently to the user by defining substitute routines for `midpnt` which make the change of variable automatically:

```
SUBROUTINE midsql(funk,aa,bb,s,n)
INTEGER n
REAL aa,bb,s,funk
EXTERNAL funk
    This routine is an exact replacement for midpnt, except that it allows for an inverse square-
    root singularity in the integrand at the lower limit aa.
INTEGER it,j
REAL ddel,del,sum,tnm,x,func,a,b
func(x)=2.*x*funk(aa+x**2)
b=sqrt(bb-aa)
a=0.
if (n.eq.1) then
    The rest of the routine is exactly like midpnt and is omitted.
```

Similarly,

```
SUBROUTINE midsqu(funk,aa,bb,s,n)
INTEGER n
REAL aa,bb,s,funk
EXTERNAL funk
    This routine is an exact replacement for midpnt, except that it allows for an inverse square-
    root singularity in the integrand at the upper limit bb.
INTEGER it,j
REAL ddel,del,sum,tnm,x,func,a,b
func(x)=2.*x*funk(bb-x**2)
b=sqrt(bb-aa)
a=0.
if (n.eq.1) then
    The rest of the routine is exactly like midpnt and is omitted.
```

One last example should suffice to show how these formulas are derived in general. Suppose the upper limit of integration is infinite, and the integrand falls off exponentially. Then we want a change of variable that maps $e^{-x}dx$ into $(\pm)dt$ (with the sign chosen to keep the upper limit of the new variable larger than the lower limit). Doing the integration gives by inspection

$$t = e^{-x} \qquad \text{or} \qquad x = -\log t \tag{4.4.7}$$

so that

$$\int_{x=a}^{x=\infty} f(x)dx = \int_{t=0}^{t=e^{-a}} f(-\log t)\frac{dt}{t} \tag{4.4.8}$$

The user-transparent implementation would be

```
SUBROUTINE midexp(funk,aa,bb,s,n)
INTEGER n
REAL aa,bb,s,funk
EXTERNAL funk
    This routine is an exact replacement for midpnt, except that bb is assumed to be infinite
    (value passed not actually used). It is assumed that the function funk decreases exponen-
    tially rapidly at infinity.
INTEGER it,j
REAL ddel,del,sum,tnm,x,func,a,b
func(x)=funk(-log(x))/x
b=exp(-aa)
a=0.
if (n.eq.1) then
    The rest of the routine is exactly like midpnt and is omitted.
```

CITED REFERENCES AND FURTHER READING:

Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 4.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §7.4.3, p. 294.

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §3.7, p. 152.

4.5 Gaussian Quadratures and Orthogonal Polynomials

In the formulas of §4.1, the integral of a function was approximated by the sum of its functional values at a set of equally spaced points, multiplied by certain aptly chosen weighting coefficients. We saw that as we allowed ourselves more freedom in choosing the coefficients, we could achieve integration formulas of higher and higher order. The idea of *Gaussian quadratures* is to give ourselves the freedom to choose not only the weighting coefficients, but also the location of the abscissas at which the function is to be evaluated: They will no longer be equally spaced. Thus, we will have *twice* the number of degrees of freedom at our disposal; it will turn out that we can achieve Gaussian quadrature formulas whose order is, essentially, twice that of the Newton-Cotes formula with the same number of function evaluations.

Does this sound too good to be true? Well, in a sense it is. The catch is a familiar one, which cannot be overemphasized: High order is not the same as high accuracy. High order translates to high accuracy only when the integrand is very smooth, in the sense of being "well-approximated by a polynomial."

There is, however, one additional feature of Gaussian quadrature formulas that adds to their usefulness: We can arrange the choice of weights and abscissas to make the integral exact for a class of integrands "polynomials times some known function $W(x)$" rather than for the usual class of integrands "polynomials." The function $W(x)$ can then be chosen to remove integrable singularities from the desired integral. Given $W(x)$, in other words, and given an integer N, we can find a set of weights w_j and abscissas x_j such that the approximation

$$\int_a^b W(x)f(x)dx \approx \sum_{j=1}^{N} w_j f(x_j) \tag{4.5.1}$$

is exact if $f(x)$ is a polynomial. For example, to do the integral

$$\int_{-1}^{1} \frac{\exp(-\cos^2 x)}{\sqrt{1-x^2}}dx \tag{4.5.2}$$

(not a very natural looking integral, it must be admitted), we might well be interested in a Gaussian quadrature formula based on the choice

$$W(x) = \frac{1}{\sqrt{1-x^2}} \tag{4.5.3}$$

in the interval $(-1, 1)$. (This particular choice is called *Gauss-Chebyshev integration*, for reasons that will become clear shortly.)

Notice that the integration formula (4.5.1) can also be written with the weight function $W(x)$ not overtly visible: Define $g(x) \equiv W(x)f(x)$ and $v_j \equiv w_j/W(x_j)$. Then (4.5.1) becomes

$$\int_a^b g(x)dx \approx \sum_{j=1}^{N} v_j g(x_j) \tag{4.5.4}$$

Where did the function $W(x)$ go? It is lurking there, ready to give high-order accuracy to integrands of the form polynomials times $W(x)$, and ready to *deny* high-order accuracy to integrands that are otherwise perfectly smooth and well-behaved. When you find tabulations of the weights and abscissas for a given $W(x)$, you have to determine carefully whether they are to be used with a formula in the form of (4.5.1), or like (4.5.4).

Here is an example of a quadrature routine that contains the tabulated abscissas and weights for the case $W(x) = 1$ and $N = 10$. Since the weights and abscissas are, in this case, symmetric around the midpoint of the range of integration, there are actually only five distinct values of each:

```
SUBROUTINE qgaus(func,a,b,ss)
REAL a,b,ss,func
EXTERNAL func
    Returns as ss the integral of the function func between a and b, by ten-point Gauss-
    Legendre integration: the function is evaluated exactly ten times at interior points in the
    range of integration.
INTEGER j
```

```
REAL dx,xm,xr,w(5),x(5)        The abscissas and weights.
SAVE w,x
DATA w/.2955242247,.2692667193,.2190863625,.1494513491,.0666713443/
DATA x/.1488743389,.4333953941,.6794095682,.8650633666,.9739065285/
xm=0.5*(b+a)
xr=0.5*(b-a)
ss=0                           Will be twice the average value of the function, since the ten
do 11 j=1,5                        weights (five numbers above each used twice) sum to 2.
    dx=xr*x(j)
    ss=ss+w(j)*(func(xm+dx)+func(xm-dx))
enddo 11
ss=xr*ss                       Scale the answer to the range of integration.
return
END
```

The above routine illustrates that one can use Gaussian quadratures without necessarily understanding the theory behind them: One just locates tabulated weights and abscissas in a book (e.g., [1] or [2]). However, the theory is very pretty, and it will come in handy if you ever need to construct your own tabulation of weights and abscissas for an unusual choice of $W(x)$. We will therefore give, without any proofs, some useful results that will enable you to do this. Several of the results assume that $W(x)$ does not change sign inside (a, b), which is usually the case in practice.

The theory behind Gaussian quadratures goes back to Gauss in 1814, who used continued fractions to develop the subject. In 1826 Jacobi rederived Gauss's results by means of orthogonal polynomials. The systematic treatment of arbitrary weight functions $W(x)$ using orthogonal polynomials is largely due to Christoffel in 1877. To introduce these orthogonal polynomials, let us fix the interval of interest to be (a, b). We can define the "scalar product of two functions f and g over a weight function W" as

$$\langle f|g\rangle \equiv \int_a^b W(x)f(x)g(x)dx \tag{4.5.5}$$

The scalar product is a number, not a function of x. Two functions are said to be *orthogonal* if their scalar product is zero. A function is said to be *normalized* if its scalar product with itself is unity. A set of functions that are all mutually orthogonal and also all individually normalized is called an *orthonormal* set.

We can find a set of polynomials (i) that includes exactly one polynomial of order j, called $p_j(x)$, for each $j = 0, 1, 2, \ldots$, and (ii) all of which are mutually orthogonal over the specified weight function $W(x)$. A constructive procedure for finding such a set is the recurrence relation

$$\begin{aligned} p_{-1}(x) &\equiv 0 \\ p_0(x) &\equiv 1 \\ p_{j+1}(x) &= (x - a_j)p_j(x) - b_j p_{j-1}(x) \qquad j = 0, 1, 2, \ldots \end{aligned} \tag{4.5.6}$$

where

$$\begin{aligned} a_j &= \frac{\langle xp_j|p_j\rangle}{\langle p_j|p_j\rangle} \qquad j = 0, 1, \ldots \\ b_j &= \frac{\langle p_j|p_j\rangle}{\langle p_{j-1}|p_{j-1}\rangle} \qquad j = 1, 2, \ldots \end{aligned} \tag{4.5.7}$$

The coefficient b_0 is arbitrary; we can take it to be zero.

The polynomials defined by (4.5.6) are *monic*, i.e., the coefficient of their leading term [x^j for $p_j(x)$] is unity. If we divide each $p_j(x)$ by the constant $[\langle p_j|p_j\rangle]^{1/2}$ we can render the set of polynomials orthonormal. One also encounters orthogonal polynomials with various other normalizations. You can convert from a given normalization to monic polynomials if you know that the coefficient of x^j in p_j is λ_j, say; then the monic polynomials are obtained by dividing each p_j by λ_j. Note that the coefficients in the recurrence relation (4.5.6) depend on the adopted normalization.

The polynomial $p_j(x)$ can be shown to have exactly j distinct roots in the interval (a, b). Moreover, it can be shown that the roots of $p_j(x)$ "interleave" the $j-1$ roots of $p_{j-1}(x)$, i.e., there is exactly one root of the former in between each two adjacent roots of the latter. This fact comes in handy if you need to find all the roots: You can start with the one root of $p_1(x)$ and then, in turn, bracket the roots of each higher j, pinning them down at each stage more precisely by Newton's rule or some other root-finding scheme (see Chapter 9).

Why would you ever want to find all the roots of an orthogonal polynomial $p_j(x)$? Because the abscissas of the N-point Gaussian quadrature formulas (4.5.1) and (4.5.4) with weighting function $W(x)$ in the interval (a, b) are precisely the roots of the orthogonal polynomial $p_N(x)$ for the same interval and weighting function. This is the fundamental theorem of Gaussian quadratures, and lets you find the abscissas for any particular case.

Once you know the abscissas $x_1, \ldots, x_N$, you need to find the weights w_j, $j = 1, \ldots, N$. One way to do this (not the most efficient) is to solve the set of linear equations

$$\begin{bmatrix} p_0(x_1) & \cdots & p_0(x_N) \\ p_1(x_1) & \cdots & p_1(x_N) \\ \vdots & & \vdots \\ p_{N-1}(x_1) & \cdots & p_{N-1}(x_N) \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_N \end{bmatrix} = \begin{bmatrix} \int_a^b W(x)p_0(x)dx \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{4.5.8}$$

Equation (4.5.8) simply solves for those weights such that the quadrature (4.5.1) gives the correct answer for the integral of the first N orthogonal polynomials. Note that the zeros on the right-hand side of (4.5.8) appear because $p_1(x), \ldots, p_{N-1}(x)$ are all orthogonal to $p_0(x)$, which is a constant. It can be shown that, with those weights, the integral of the *next* $N-1$ polynomials is also exact, so that the quadrature is exact for all polynomials of degree $2N-1$ or less. Another way to evaluate the weights (though one whose proof is beyond our scope) is by the formula

$$w_j = \frac{\langle p_{N-1}|p_{N-1}\rangle}{p_{N-1}(x_j)p'_N(x_j)} \tag{4.5.9}$$

where $p'_N(x_j)$ is the derivative of the orthogonal polynomial at its zero x_j.

The computation of Gaussian quadrature rules thus involves two distinct phases: (i) the generation of the orthogonal polynomials $p_0, \ldots, p_N$, i.e., the computation of the coefficients a_j, b_j in (4.5.6); (ii) the determination of the zeros of $p_N(x)$, and the computation of the associated weights. For the case of the "classical" orthogonal polynomials, the coefficients a_j and b_j are explicitly known (equations 4.5.10 –

4.5.14 below) and phase (i) can be omitted. However, if you are confronted with a "nonclassical" weight function $W(x)$, and you don't know the coefficients a_j and b_j, the construction of the associated set of orthogonal polynomials is not trivial. We discuss it at the end of this section.

Computation of the Abscissas and Weights

This task can range from easy to difficult, depending on how much you already know about your weight function and its associated polynomials. In the case of classical, well-studied, orthogonal polynomials, practically everything is known, including good approximations for their zeros. These can be used as starting guesses, enabling Newton's method (to be discussed in §9.4) to converge very rapidly. Newton's method requires the derivative $p_N'(x)$, which is evaluated by standard relations in terms of p_N and p_{N-1}. The weights are then conveniently evaluated by equation (4.5.9). For the following named cases, this direct root-finding is faster, by a factor of 3 to 5, than any other method.

Here are the weight functions, intervals, and recurrence relations that generate the most commonly used orthogonal polynomials and their corresponding Gaussian quadrature formulas.

Gauss-Legendre:

$$W(x) = 1 \qquad -1 < x < 1$$

$$(j+1)P_{j+1} = (2j+1)xP_j - jP_{j-1} \tag{4.5.10}$$

Gauss-Chebyshev:

$$W(x) = (1-x^2)^{-1/2} \qquad -1 < x < 1$$

$$T_{j+1} = 2xT_j - T_{j-1} \tag{4.5.11}$$

Gauss-Laguerre:

$$W(x) = x^\alpha e^{-x} \qquad 0 < x < \infty$$

$$(j+1)L_{j+1}^\alpha = (-x+2j+\alpha+1)L_j^\alpha - (j+\alpha)L_{j-1}^\alpha \tag{4.5.12}$$

Gauss-Hermite:

$$W(x) = e^{-x^2} \qquad -\infty < x < \infty$$

$$H_{j+1} = 2xH_j - 2jH_{j-1} \tag{4.5.13}$$

Gauss-Jacobi:

$$W(x) = (1-x)^\alpha(1+x)^\beta \qquad -1 < x < 1$$

$$c_j P_{j+1}^{(\alpha,\beta)} = (d_j + e_j x) P_j^{(\alpha,\beta)} - f_j P_{j-1}^{(\alpha,\beta)} \tag{4.5.14}$$

where the coefficients c_j, d_j, e_j, and f_j are given by

$$\begin{aligned} c_j &= 2(j+1)(j+\alpha+\beta+1)(2j+\alpha+\beta) \\ d_j &= (2j+\alpha+\beta+1)(\alpha^2-\beta^2) \\ e_j &= (2j+\alpha+\beta)(2j+\alpha+\beta+1)(2j+\alpha+\beta+2) \\ f_j &= 2(j+\alpha)(j+\beta)(2j+\alpha+\beta+2) \end{aligned} \tag{4.5.15}$$

We now give individual routines that calculate the abscissas and weights for these cases. First comes the most common set of abscissas and weights, those of Gauss-Legendre. The routine, due to G.B. Rybicki, uses equation (4.5.9) in the special form for the Gauss-Legendre case,

$$w_j = \frac{2}{(1-x_j^2)[P_N'(x_j)]^2} \tag{4.5.16}$$

The routine also scales the range of integration from (x_1, x_2) to $(-1, 1)$, and provides abscissas x_j and weights w_j for the Gaussian formula

$$\int_{x_1}^{x_2} f(x)dx = \sum_{j=1}^{N} w_j f(x_j) \tag{4.5.17}$$

```
      SUBROUTINE gauleg(x1,x2,x,w,n)
      INTEGER n
      REAL x1,x2,x(n),w(n)
      DOUBLE PRECISION EPS
      PARAMETER (EPS=3.d-14)                    EPS is the relative precision.
        Given the lower and upper limits of integration x1 and x2, and given n, this routine returns
        arrays x(1:n) and w(1:n) of length n, containing the abscissas and weights of the Gauss-
        Legendre n-point quadrature formula.
      INTEGER i,j,m
      DOUBLE PRECISION p1,p2,p3,pp,xl,xm,z,z1
        High precision is a good idea for this routine.
      m=(n+1)/2                                 The roots are symmetric in the interval, so we
      xm=0.5d0*(x2+x1)                            only have to find half of them.
      xl=0.5d0*(x2-x1)
      do 12 i=1,m                               Loop over the desired roots.
        z=cos(3.141592654d0*(i-.25d0)/(n+.5d0))
          Starting with the above approximation to the ith root, we enter the main loop of re-
          finement by Newton's method.
1       continue
          p1=1.d0
          p2=0.d0
          do 11 j=1,n                           Loop up the recurrence relation to get the Leg-
            p3=p2                                 endre polynomial evaluated at z.
            p2=p1
            p1=((2.d0*j-1.d0)*z*p2-(j-1.d0)*p3)/j
          enddo 11
            p1 is now the desired Legendre polynomial. We next compute pp, its derivative, by
            a standard relation involving also p2, the polynomial of one lower order.
          pp=n*(z*p1-p2)/(z*z-1.d0)
```

```
        z1=z
        z=z1-p1/pp                          Newton's method.
      if(abs(z-z1).gt.EPS)goto 1
      x(i)=xm-xl*z                          Scale the root to the desired interval,
      x(n+1-i)=xm+xl*z                      and put in its symmetric counterpart.
      w(i)=2.d0*xl/((1.d0-z*z)*pp*pp)       Compute the weight
      w(n+1-i)=w(i)                         and its symmetric counterpart.
    enddo 12
    return
    END
```

Next we give three routines that use initial approximations for the roots given by Stroud and Secrest [2]. The first is for Gauss-Laguerre abscissas and weights, to be used with the integration formula

$$\int_0^\infty x^\alpha e^{-x} f(x)dx = \sum_{j=1}^{N} w_j f(x_j) \tag{4.5.18}$$

```
      SUBROUTINE gaulag(x,w,n,alf)
      INTEGER n,MAXIT
      REAL alf,w(n),x(n)
      DOUBLE PRECISION EPS
      PARAMETER (EPS=3.D-14,MAXIT=10)     Increase EPS if you don't have this precision.
C     USES gammln
        Given alf, the parameter α of the Laguerre polynomials, this routine returns arrays x(1:n)
        and w(1:n) containing the abscissas and weights of the n-point Gauss-Laguerre quadrature
        formula. The smallest abscissa is returned in x(1), the largest in x(n).
      INTEGER i,its,j
      REAL ai,gammln
      DOUBLE PRECISION p1,p2,p3,pp,z,z1
        High precision is a good idea for this routine.
      do 13 i=1,n                         Loop over the desired roots.
        if(i.eq.1)then                    Initial guess for the smallest root.
          z=(1.+alf)*(3.+.92*alf)/(1.+2.4*n+1.8*alf)
        else if(i.eq.2)then               Initial guess for the second root.
          z=z+(15.+6.25*alf)/(1.+.9*alf+2.5*n)
        else                              Initial guess for the other roots.
          ai=i-2
          z=z+((1.+2.55*ai)/(1.9*ai)+1.26*ai*alf/
     *         (1.+3.5*ai))*(z-x(i-2))/(1.+.3*alf)
        endif
        do 12 its=1,MAXIT                 Refinement by Newton's method.
          p1=1.d0
          p2=0.d0
          do 11 j=1,n                     Loop up the recurrence relation to get the Laguerre
            p3=p2                             polynomial evaluated at z.
            p2=p1
            p1=((2*j-1+alf-z)*p2-(j-1+alf)*p3)/j
          enddo 11
            p1 is now the desired Laguerre polynomial. We next compute pp, its derivative, by
            a standard relation involving also p2, the polynomial of one lower order.
          pp=(n*p1-(n+alf)*p2)/z
          z1=z
          z=z1-p1/pp                      Newton's formula.
          if(abs(z-z1).le.EPS)goto 1
        enddo 12
        pause 'too many iterations in gaulag'
1       x(i)=z                            Store the root and the weight.
```

```
        w(i)=-exp(gammln(alf+n)-gammln(float(n)))/(pp*n*p2)
      enddo 13
      return
      END
```

Next is a routine for Gauss-Hermite abscissas and weights. If we use the "standard" normalization of these functions, as given in equation (4.5.13), we find that the computations overflow for large N because of various factorials that occur. We can avoid this by using instead the orthonomal set of polynomials $\tilde{H}_j$. They are generated by the recurrence

$$\tilde{H}_{-1} = 0, \quad \tilde{H}_0 = \frac{1}{\pi^{1/4}}, \quad \tilde{H}_{j+1} = x\sqrt{\frac{2}{j+1}}\tilde{H}_j - \sqrt{\frac{j}{j+1}}\tilde{H}_{j-1} \tag{4.5.19}$$

The formula for the weights becomes

$$w_j = \frac{2}{(\tilde{H}_j')^2} \tag{4.5.20}$$

while the formula for the derivative with this normalization is

$$\tilde{H}_j' = \sqrt{2j}\tilde{H}_{j-1} \tag{4.5.21}$$

The abscissas and weights returned by `gauher` are used with the integration formula

$$\int_{-\infty}^{\infty} e^{-x^2} f(x)dx = \sum_{j=1}^{N} w_j f(x_j) \tag{4.5.22}$$

```
      SUBROUTINE gauher(x,w,n)
      INTEGER n,MAXIT
      REAL w(n),x(n)
      DOUBLE PRECISION EPS,PIM4
      PARAMETER (EPS=3.D-14,PIM4=.75112554446494250,MAXIT=10)
        Given n, this routine returns arrays x(1:n) and w(1:n) containing the abscissas and
        weights of the n-point Gauss-Hermite quadrature formula. The largest abscissa is returned
        in x(1), the most negative in x(n).
        Parameters: EPS is the relative precision, PIM4 = 1/π^1/4, MAXIT = maximum iterations.
      INTEGER i,its,j,m
      DOUBLE PRECISION p1,p2,p3,pp,z,z1
        High precision is a good idea for this routine.
      m=(n+1)/2
        The roots are symmetric about the origin, so we have to find only half of them.
      do 13 i=1,m                       Loop over the desired roots.
        if(i.eq.1)then                  Initial guess for the largest root.
          z=sqrt(float(2*n+1))-1.85575*(2*n+1)**(-.16667)
        else if(i.eq.2)then             Initial guess for the second largest root.
          z=z-1.14*n**.426/z
        else if (i.eq.3)then            Initial guess for the third largest root.
          z=1.86*z-.86*x(1)
        else if (i.eq.4)then            Initial guess for the fourth largest root.
          z=1.91*z-.91*x(2)
        else                            Initial guess for the other roots.
          z=2.*z-x(i-2)
```

```
      endif
      do 12 its=1,MAXIT                Refinement by Newton's method.
        p1=PIM4
        p2=0.d0
        do 11 j=1,n                    Loop up the recurrence relation to get the Hermite poly-
          p3=p2                            nomial evaluated at z.
          p2=p1
          p1=z*sqrt(2.d0/j)*p2-sqrt(dble(j-1)/dble(j))*p3
        enddo 11
        p1 is now the desired Hermite polynomial. We next compute pp, its derivative, by
        the relation (4.5.21) using p2, the polynomial of one lower order.
        pp=sqrt(2.d0*n)*p2
        z1=z
        z=z1-p1/pp                     Newton's formula.
        if(abs(z-z1).le.EPS)goto 1
      enddo 12
      pause 'too many iterations in gauher'
1     x(i)=z                           Store the root
      x(n+1-i)=-z                      and its symmetric counterpart.
      w(i)=2.d0/(pp*pp)                Compute the weight
      w(n+1-i)=w(i)                    and its symmetric counterpart.
    enddo 13
    return
    END
```

Finally, here is a routine for Gauss-Jacobi abscissas and weights, which implement the integration formula

$$\int_{-1}^{1}(1-x)^{\alpha}(1+x)^{\beta}f(x)dx=\sum_{j=1}^{N}w_j f(x_j) \tag{4.5.23}$$

```
      SUBROUTINE gaujac(x,w,n,alf,bet)
      INTEGER n,MAXIT
      REAL alf,bet,w(n),x(n)
      DOUBLE PRECISION EPS
      PARAMETER (EPS=3.D-14,MAXIT=10)   Increase EPS if you don't have this precision.
C     USES gammln
        Given alf and bet, the parameters α and β of the Jacobi polynomials, this routine returns
        arrays x(1:n) and w(1:n) containing the abscissas and weights of the n-point Gauss-Jacobi
        quadrature formula. The largest abscissa is returned in x(1), the smallest in x(n).
      INTEGER i,its,j
      REAL alfbet,an,bn,r1,r2,r3,gammln
      DOUBLE PRECISION a,b,c,p1,p2,p3,pp,temp,z,z1
        High precision is a good idea for this routine.
      do 13 i=1,n                       Loop over the desired roots.
        if(i.eq.1)then                  Initial guess for the largest root.
          an=alf/n
          bn=bet/n
          r1=(1.+alf)*(2.78/(4.+n*n)+.768*an/n)
          r2=1.+1.48*an+.96*bn+.452*an*an+.83*an*bn
          z=1.-r1/r2
        else if(i.eq.2)then             Initial guess for the second largest root.
          r1=(4.1+alf)/((1.+alf)*(1.+.156*alf))
          r2=1.+.06*(n-8.)*(1.+.12*alf)/n
          r3=1.+.012*bet*(1.+.25*abs(alf))/n
          z=z-(1.-z)*r1*r2*r3
        else if(i.eq.3)then             Initial guess for the third largest root.
          r1=(1.67+.28*alf)/(1.+.37*alf)
          r2=1.+.22*(n-8.)/n
```

```
        r3=1.+8.*bet/((6.28+bet)*n*n)
        z=z-(x(i)-z)*r1*r2*r3
      else if(i.eq.n-1)then          Initial guess for the second smallest root.
        r1=(1.+.235*bet)/(.766+.119*bet)
        r2=1./(1.+.639*(n-4.)/(1.+.71*(n-4.)))
        r3=1./(1.+20.*alf/((7.5+alf)*n*n))
        z=z+(z-x(n-3))*r1*r2*r3
      else if(i.eq.n)then            Initial guess for the smallest root.
        r1=(1.+.37*bet)/(1.67+.28*bet)
        r2=1./(1.+.22*(n-8.)/n)
        r3=1./(1.+8.*alf/((6.28+alf)*n*n))
        z=z+(z-x(n-2))*r1*r2*r3
      else                           Initial guess for the other roots.
        z=3.*x(i-1)-3.*x(i-2)+x(i-3)
      endif
      alfbet=alf+bet
      do 12 its=1,MAXIT              Refinement by Newton's method.
        temp=2.d0+alfbet             Start the recurrence with P_0 and P_1 to avoid a divi-
        p1=(alf-bet+temp*z)/2.d0       sion by zero when α + β = 0 or −1.
        p2=1.d0
        do 11 j=2,n                  Loop up the recurrence relation to get the Jacobi
          p3=p2                        polynomial evaluated at z.
          p2=p1
          temp=2*j+alfbet
          a=2*j*(j+alfbet)*(temp-2.d0)
          b=(temp-1.d0)*(alf*alf-bet*bet+temp*
     *      (temp-2.d0)*z)
          c=2.d0*(j-1+alf)*(j-1+bet)*temp
          p1=(b*p2-c*p3)/a
        enddo 11
        pp=(n*(alf-bet-temp*z)*p1+2.d0*(n+alf)*
     *      (n+bet)*p2)/(temp*(1.d0-z*z))
        p1 is now the desired Jacobi polynomial. We next compute pp, its derivative, by a
        standard relation involving also p2, the polynomial of one lower order.
        z1=z
        z=z1-p1/pp                   Newton's formula.
        if(abs(z-z1).le.EPS)goto 1
      enddo 12
      pause 'too many iterations in gaujac'
1     x(i)=z                         Store the root and the weight.
      w(i)=exp(gammln(alf+n)+gammln(bet+n)-gammln(n+1.)-
     *    gammln(n+alfbet+1.))*temp*2.**alfbet/(pp*p2)
    enddo 13
    return
    END
```

Legendre polynomials are special cases of Jacobi polynomials with $\alpha = \beta = 0$, but it is worth having the separate routine for them, `gauleg`, given above. Chebyshev polynomials correspond to $\alpha = \beta = -1/2$ (see §5.8). They have analytic abscissas and weights:

$$x_j = \cos\left(\frac{\pi(j-\frac{1}{2})}{N}\right)$$
$$w_j = \frac{\pi}{N} \tag{4.5.24}$$

Case of Known Recurrences

Turn now to the case where you do not know good initial guesses for the zeros of your orthogonal polynomials, but you do have available the coefficients a_j and b_j that generate them. As we have seen, the zeros of $p_N(x)$ are the abscissas for the N-point Gaussian quadrature formula. The most useful computational formula for the weights is equation (4.5.9) above, since the derivative p'_N can be efficiently computed by the derivative of (4.5.6) in the general case, or by special relations for the classical polynomials. Note that (4.5.9) is valid as written only for monic polynomials; for other normalizations, there is an extra factor of λ_N/λ_{N-1}, where λ_N is the coefficient of x^N in p_N.

Except in those special cases already discussed, the best way to find the abscissas is *not* to use a root-finding method like Newton's method on $p_N(x)$. Rather, it is generally faster to use the Golub-Welsch [3] algorithm, which is based on a result of Wilf [4]. This algorithm notes that if you bring the term xp_j to the left-hand side of (4.5.6) and the term p_{j+1} to the right-hand side, the recurrence relation can be written in matrix form as

$$x\begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{N-2} \\ p_{N-1} \end{bmatrix} = \begin{bmatrix} a_0 & 1 & & & & \\ b_1 & a_1 & 1 & & & \\ & \vdots & \vdots & & & \\ & & & b_{N-2} & a_{N-2} & 1 \\ & & & & b_{N-1} & a_{N-1} \end{bmatrix} \cdot \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{N-2} \\ p_{N-1} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ p_N \end{bmatrix}$$

or

$$x\mathbf{p} = \mathbf{T} \cdot \mathbf{p} + p_N \mathbf{e}_{N-1} \tag{4.5.25}$$

Here $\mathbf{T}$ is a tridiagonal matrix, $\mathbf{p}$ is a column vector of $p_0, p_1, \ldots, p_{N-1}$, and $\mathbf{e}_{N-1}$ is a unit vector with a 1 in the $(N-1)$st (last) position and zeros elsewhere. The matrix $\mathbf{T}$ can be symmetrized by a diagonal similarity transformation $\mathbf{D}$ to give

$$\mathbf{J} = \mathbf{DTD}^{-1} = \begin{bmatrix} a_0 & \sqrt{b_1} & & & \\ \sqrt{b_1} & a_1 & \sqrt{b_2} & & \\ & \vdots & \vdots & & \\ & & \sqrt{b_{N-2}} & a_{N-2} & \sqrt{b_{N-1}} \\ & & & \sqrt{b_{N-1}} & a_{N-1} \end{bmatrix} \tag{4.5.26}$$

The matrix $\mathbf{J}$ is called the *Jacobi matrix* (not to be confused with other matrices named after Jacobi that arise in completely different problems!). Now we see from (4.5.25) that $p_N(x_j) = 0$ is equivalent to x_j being an eigenvalue of $\mathbf{T}$. Since eigenvalues are preserved by a similarity transformation, x_j is an eigenvalue of the symmetric tridiagonal matrix $\mathbf{J}$. Moreover, Wilf [4] shows that if $\mathbf{v}_j$ is the eigenvector corresponding to the eigenvalue x_j, normalized so that $\mathbf{v} \cdot \mathbf{v} = 1$, then

$$w_j = \mu_0 v_{j,1}^2 \tag{4.5.27}$$

where

$$\mu_0 = \int_a^b W(x)\,dx \tag{4.5.28}$$

and where $v_{j,1}$ is the first component of $\mathbf{v}$. As we shall see in Chapter 11, finding all eigenvalues and eigenvectors of a symmetric tridiagonal matrix is a relatively efficient and well-conditioned procedure. We accordingly give a routine, `gaucof`, for finding the abscissas and weights, given the coefficients a_j and b_j. Remember that if you know the recurrence relation for orthogonal polynomials that are not normalized to be monic, you can easily convert it to monic form by means of the quantities λ_j.

```
      SUBROUTINE gaucof(n,a,b,amu0,x,w)
      INTEGER n,NMAX
      REAL amu0,a(n),b(n),w(n),x(n)
      PARAMETER (NMAX=64)
C     USES eigsrt,tqli
        Computes the abscissas and weights for a Gaussian quadrature formula from the Jacobi
        matrix. On input, a(1:n) and b(1:n) are the coefficients of the recurrence relation for
        the set of monic orthogonal polynomials. The quantity μ0 ≡ ∫_a^b W(x) dx is input as amu0.
        The abscissas x(1:n) are returned in descending order, with the corresponding weights
        in w(1:n). The arrays a and b are modified. Execution can be speeded up by modifying
        tqli and eigsrt to compute only the first component of each eigenvector.
      INTEGER i,j
      REAL z(NMAX,NMAX)
      do 12 i=1,n
        if(i.ne.1)b(i)=sqrt(b(i))        Set up superdiagonal of Jacobi matrix.
        do 11 j=1,n                      Set up identity matrix for tqli to compute eigenvectors.
          if(i.eq.j)then
            z(i,j)=1.
          else
            z(i,j)=0.
          endif
        enddo 11
      enddo 12
      call tqli(a,b,n,NMAX,z)
      call eigsrt(a,z,n,NMAX)            Sort eigenvalues into descending order.
      do 13 i=1,n
        x(i)=a(i)
        w(i)=amu0*z(1,i)**2              Equation (4.5.12).
      enddo 13
      return
      END
```

Orthogonal Polynomials with Nonclassical Weights

This somewhat specialized subsection will tell you what to do if your weight function is not one of the classical ones dealt with above and you do not know the a_j's and b_j's of the recurrence relation (4.5.6) to use in gaucof. Then, a method of finding the a_j's and b_j's is needed.

The *procedure of Stieltjes* is to compute a_0 from (4.5.7), then $p_1(x)$ from (4.5.6). Knowing p_0 and p_1, we can compute a_1 and b_1 from (4.5.7), and so on. But how are we to compute the inner products in (4.5.7)?

The textbook approach is to represent each $p_j(x)$ explicitly as a polynomial in x and to compute the inner products by multiplying out term by term. This will be feasible if we know the first $2N$ moments of the weight function,

$$\mu_j = \int_a^b x^j W(x)dx \qquad j = 0, 1, \ldots, 2N-1 \tag{4.5.29}$$

However, the solution of the resulting set of algebraic equations for the coefficients a_j and b_j in terms of the moments μ_j is in general *extremely* ill-conditioned. Even in double precision, it is not unusual to lose all accuracy by the time $N = 12$. We thus reject any procedure based on the moments (4.5.29).

Sack and Donovan [5] discovered that the numerical stability is greatly improved if, instead of using powers of x as a set of basis functions to represent the p_j's, one uses some other known set of orthogonal polynomials $\pi_j(x)$, say. Roughly speaking, the improved stability occurs because the polynomial basis "samples" the interval (a, b) better than the power basis when the inner product integrals are evaluated, especially if its weight function resembles $W(x)$.

So assume that we know the *modified moments*

$$\nu_j = \int_a^b \pi_j(x)W(x)dx \qquad j = 0, 1, \ldots, 2N-1 \tag{4.5.30}$$

where the π_j's satisfy a recurrence relation analogous to (4.5.6),

$$\begin{aligned} \pi_{-1}(x) &\equiv 0 \\ \pi_0(x) &\equiv 1 \\ \pi_{j+1}(x) &= (x-\alpha_j)\pi_j(x) - \beta_j\pi_{j-1}(x) \qquad j = 0, 1, 2, \ldots \end{aligned} \tag{4.5.31}$$

and the coefficients α_j, β_j are known explicitly. Then Wheeler [6] has given an efficient $O(N^2)$ algorithm equivalent to that of Sack and Donovan for finding a_j and b_j via a set of intermediate quantities

$$\sigma_{k,l} = \langle p_k | \pi_l \rangle \qquad k, l \geq -1 \tag{4.5.32}$$

Initialize

$$\begin{aligned} \sigma_{-1,l} &= 0 && l = 1, 2, \ldots, 2N-2 \\ \sigma_{0,l} &= \nu_l && l = 0, 1, \ldots, 2N-1 \\ a_0 &= \alpha_0 + \frac{\nu_1}{\nu_0} \\ b_0 &= 0 \end{aligned} \tag{4.5.33}$$

Then, for $k = 1, 2, \ldots, N-1$, compute

$$\begin{aligned} \sigma_{k,l} &= \sigma_{k-1,l-1} - (a_{k-1} - \alpha_l)\sigma_{k-1,l} - b_{k-1}\sigma_{k-2,l} + \beta_l\sigma_{k-1,l-1} \\ &\qquad\qquad l = k, k+1, \ldots, 2N-k-1 \\ a_k &= \alpha_k - \frac{\sigma_{k-1,k}}{\sigma_{k-1,k-1}} + \frac{\sigma_{k,k+1}}{\sigma_{k,k}} \\ b_k &= \frac{\sigma_{k,k}}{\sigma_{k-1,k-1}} \end{aligned} \tag{4.5.34}$$

Note that the normalization factors can also easily be computed if needed:

$$\begin{aligned} \langle p_0 | p_0 \rangle &= \nu_0 \\ \langle p_j | p_j \rangle &= b_j \langle p_{j-1} | p_{j-1} \rangle \qquad j = 1, 2, \ldots \end{aligned} \tag{4.5.35}$$

You can find a derivation of the above algorithm in Ref. [7].

Wheeler's algorithm requires that the modified moments (4.5.30) be accurately computed. In practical cases there is often a closed form, or else recurrence relations can be used. The algorithm is extremely successful for *finite* intervals (a, b). For infinite intervals, the algorithm does not completely remove the ill-conditioning. In this case, Gautschi [8,9] recommends reducing the interval to a finite interval by a change of variable, and then using a suitable discretization procedure to compute the inner products. You will have to consult the references for details.

We give the routine `orthog` for generating the coefficients a_j and b_j by Wheeler's algorithm, given the coefficients α_j and β_j, and the modified moments ν_j. To conform to the usual FORTRAN convention for dimensioning subscripts, the indices of the σ matrix are increased by 2, i.e., `sig(k,l)` $= \sigma_{k-2,l-2}$, while the indices of the vectors α, β, a and b are increased by 1.

```
      SUBROUTINE orthog(n,anu,alpha,beta,a,b)
      INTEGER n,NMAX
      REAL a(n),alpha(2*n-1),anu(2*n),b(n),beta(2*n-1)
      PARAMETER (NMAX=64)
        Computes the coefficients a_j and b_j, j = 0,...N-1, of the recurrence relation for
        monic orthogonal polynomials with weight function W(x) by Wheeler's algorithm. On input,
        alpha(1:2*n-1) and beta(1:2*n-1) are the coefficients α_j and β_j, j = 0,...2N-2,
        of the recurrence relation for the chosen basis of orthogonal polynomials. The modified
        moments ν_j are input in anu(1:2*n). The first n coefficients are returned in a(1:n) and
        b(1:n).
      INTEGER k,l
      REAL sig(2*NMAX+1,2*NMAX+1)
      do 11 l=3,2*n                    Initialization, Equation (4.5.33).
        sig(1,l)=0.
      enddo 11
      do 12 l=2,2*n+1
        sig(2,l)=anu(l-1)
      enddo 12
      a(1)=alpha(1)+anu(2)/anu(1)
      b(1)=0.
      do 14 k=3,n+1                    Equation (4.5.34).
        do 13 l=k,2*n-k+3
          sig(k,l)=sig(k-1,l+1)+(alpha(l-1)-a(k-2))*sig(k-1,l)-
     *        b(k-2)*sig(k-2,l)+beta(l-1)*sig(k-1,l-1)
        enddo 13
        a(k-1)=alpha(k-1)+sig(k,k+1)/sig(k,k)-sig(k-1,k)/sig(k-1,k-1)
        b(k-1)=sig(k,k)/sig(k-1,k-1)
      enddo 14
      return
      END
```

As an example of the use of `orthog`, consider the problem [7] of generating orthogonal polynomials with the weight function $W(x) = -\log x$ on the interval $(0, 1)$. A suitable set of π_j's is the shifted Legendre polynomials

$$\pi_j = \frac{(j!)^2}{(2j)!} P_j(2x-1) \tag{4.5.36}$$

The factor in front of P_j makes the polynomials monic. The coefficients in the recurrence relation (4.5.31) are

$$\begin{aligned} \alpha_j &= \frac{1}{2} && j = 0, 1, \ldots \\ \beta_j &= \frac{1}{4(4-j^{-2})} && j = 1, 2, \ldots \end{aligned} \tag{4.5.37}$$

while the modified moments are

$$\nu_j = \begin{cases} 1 & j = 0 \\ \dfrac{(-1)^j (j!)^2}{j(j+1)(2j)!} & j \geq 1 \end{cases} \tag{4.5.38}$$

A call to `orthog` with this input allows one to generate the required polynomials to machine accuracy for very large N, and hence do Gaussian quadrature with this weight function. Before Sack and Donovan's observation, this seemingly simple problem was essentially intractable.

Extensions of Gaussian Quadrature

There are many different ways in which the ideas of Gaussian quadrature have been extended. One important extension is the case of *preassigned nodes*: Some points are required to be included in the set of abscissas, and the problem is to choose

the weights and the remaining abscissas to maximize the degree of exactness of the the quadrature rule. The most common cases are *Gauss-Radau* quadrature, where one of the nodes is an endpoint of the interval, either a or b, and *Gauss-Lobatto* quadrature, where both a and b are nodes. Golub [10] has given an algorithm similar to `gaucof` for these cases.

The second important extension is the *Gauss-Kronrod* formulas. For ordinary Gaussian quadrature formulas, as N increases the sets of abscissas have no points in common. This means that if you compare results with increasing N as a way of estimating the quadrature error, you cannot reuse the previous function evaluations. Kronrod [11] posed the problem of searching for optimal sequences of rules, each of which reuses all abscissas of its predecessor. If one starts with $N = m$, say, and then adds n new points, one has $2n + m$ free parameters: the n new abscissas and weights, and m new weights for the fixed previous abscissas. The maximum degree of exactness one would expect to achieve would therefore be $2n + m - 1$. The question is whether this maximum degree of exactness can actually be achieved in practice, when the abscissas are required to all lie inside (a, b). The answer to this question is not known in general.

Kronrod showed that if you choose $n = m + 1$, an optimal extension can be found for Gauss-Legendre quadrature. Patterson [12] showed how to compute continued extensions of this kind. Sequences such as $N = 10, 21, 43, 87, \ldots$ are popular in automatic quadrature routines [13] that attempt to integrate a function until some specified accuracy has been achieved.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, vol. 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York), §25.4. [1]

Stroud, A.H., and Secrest, D. 1966, *Gaussian Quadrature Formulas* (Englewood Cliffs, NJ: Prentice-Hall). [2]

Golub, G.H., and Welsch, J.H. 1969, *Mathematics of Computation*, vol. 23, pp. 221–230 and A1–A10. [3]

Wilf, H.S. 1962, *Mathematics for the Physical Sciences* (New York: Wiley), Problem 9, p. 80. [4]

Sack, R.A., and Donovan, A.F. 1971/72, *Numerische Mathematik*, vol. 18, pp. 465–478. [5]

Wheeler, J.C. 1974, *Rocky Mountain Journal of Mathematics*, vol. 4, pp. 287–296. [6]

Gautschi, W. 1978, in *Recent Advances in Numerical Analysis*, C. de Boor and G.H. Golub, eds. (New York: Academic Press), pp. 45–72. [7]

Gautschi, W. 1981, in *E.B. Christoffel*, P.L. Butzer and F. Fehér, eds. (Basel: Birkhauser Verlag), pp. 72–147. [8]

Gautschi, W. 1990, in *Orthogonal Polynomials*, P. Nevai, ed. (Dordrecht: Kluwer Academic Publishers), pp. 181–216. [9]

Golub, G.H. 1973, *SIAM Review*, vol. 15, pp. 318–334. [10]

Kronrod, A.S. 1964, *Doklady Akademii Nauk SSSR*, vol. 154, pp. 283–286 (in Russian). [11]

Patterson, T.N.L. 1968, *Mathematics of Computation*, vol. 22, pp. 847–856 and C1–C11; 1969, *op. cit.*, vol. 23, p. 892. [12]

Piessens, R., de Doncker, E., Uberhuber, C.W., and Kahaner, D.K. 1983, *QUADPACK: A Subroutine Package for Automatic Integration* (New York: Springer-Verlag). [13]

Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §3.6.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §6.5.

Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley), §§2.9–2.10.

Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), §§4.4–4.8.

5.8 Chebyshev Approximation

The Chebyshev polynomial of degree n is denoted $T_n(x)$, and is given by the explicit formula

$$T_n(x) = \cos(n \arccos x) \tag{5.8.1}$$

This may look trigonometric at first glance (and there is in fact a close relation between the Chebyshev polynomials and the discrete Fourier transform); however (5.8.1) can be combined with trigonometric identities to yield explicit expressions for $T_n(x)$ (see Figure 5.8.1),

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \\ T_3(x) &= 4x^3 - 3x \\ T_4(x) &= 8x^4 - 8x^2 + 1 \\ &\cdots \\ T_{n+1}(x) &= 2xT_n(x) - T_{n-1}(x) \quad n \geq 1. \end{aligned} \tag{5.8.2}$$

(There also exist inverse formulas for the powers of x in terms of the T_n's — see equations 5.11.2-5.11.3.)

The Chebyshev polynomials are orthogonal in the interval $[-1, 1]$ over a weight $(1 - x^2)^{-1/2}$. In particular,

$$\int_{-1}^{1} \frac{T_i(x)T_j(x)}{\sqrt{1-x^2}}dx = \begin{cases} 0 & i \neq j \\ \pi/2 & i = j \neq 0 \\ \pi & i = j = 0 \end{cases} \tag{5.8.3}$$

The polynomial $T_n(x)$ has n zeros in the interval $[-1, 1]$, and they are located at the points

$$x = \cos\left(\frac{\pi(k - \frac{1}{2})}{n}\right) \qquad k = 1, 2, \ldots, n \tag{5.8.4}$$

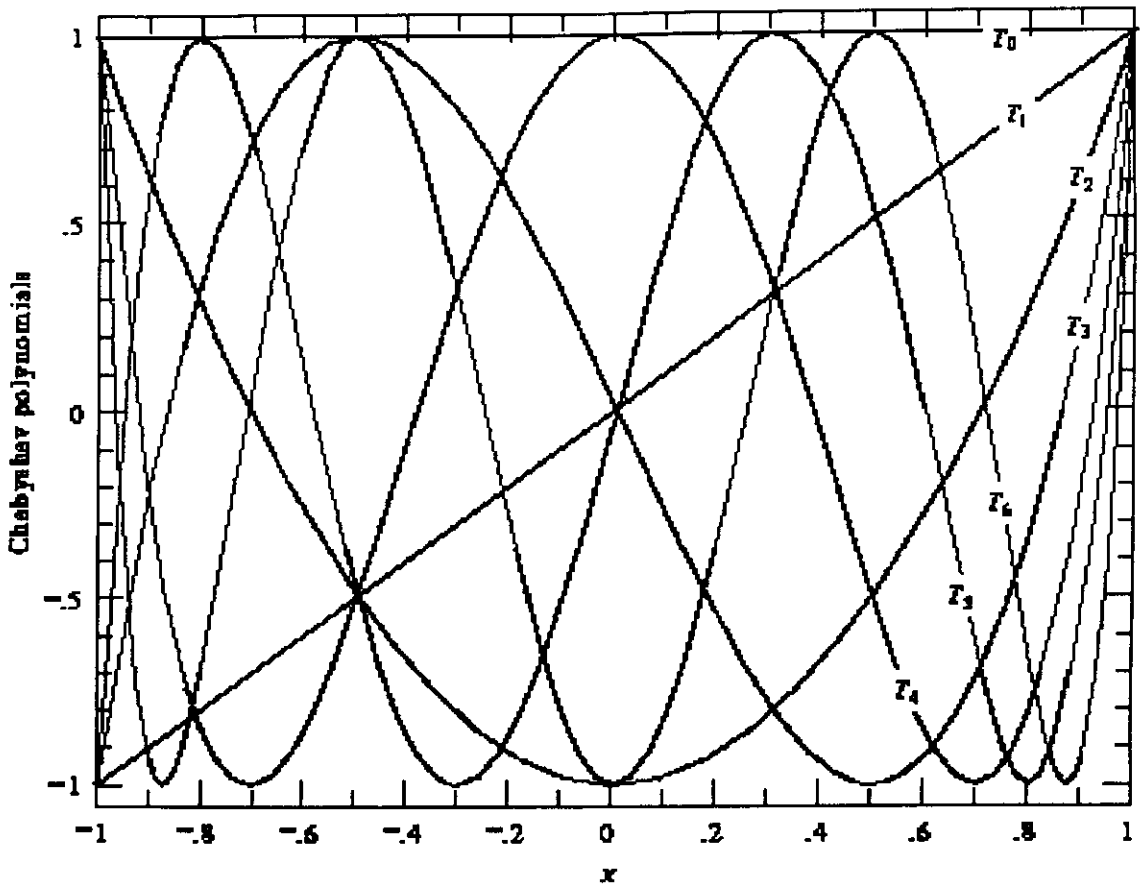


Figure 5.8.1. Chebyshev polynomials $T_0(x)$ through $T_6(x)$. Note that T_j has j roots in the interval $(-1,1)$ and that all the polynomials are bounded between ± 1.

In this same interval there are $n+1$ extrema (maxima and minima), located at

$$x = \cos\left(\frac{\pi k}{n}\right) \qquad k = 0, 1, \ldots, n \tag{5.8.5}$$

At all of the maxima $T_n(x) = 1$, while at all of the minima $T_n(x) = -1$; it is precisely this property that makes the Chebyshev polynomials so useful in polynomial approximation of functions.

The Chebyshev polynomials satisfy a discrete orthogonality relation as well as the continuous one (5.8.3): If x_k $(k = 1, \ldots, m)$ are the m zeros of $T_m(x)$ given by (5.8.4), and if $i, j < m$, then

$$\sum_{k=1}^{m} T_i(x_k) T_j(x_k) = \begin{cases} 0 & i \neq j \\ m/2 & i = j \neq 0 \\ m & i = j = 0 \end{cases} \tag{5.8.6}$$

It is not too difficult to combine equations (5.8.1), (5.8.4), and (5.8.6) to prove the following theorem: If $f(x)$ is an arbitrary function in the interval $[-1, 1]$, and if N coefficients c_j, $j = 1, \ldots, N$, are defined by

$$\begin{aligned} c_j &= \frac{2}{N} \sum_{k=1}^{N} f(x_k) T_{j-1}(x_k) \\ &= \frac{2}{N} \sum_{k=1}^{N} f\left[\cos\left(\frac{\pi(k-\frac{1}{2})}{N}\right)\right] \cos\left(\frac{\pi(j-1)(k-\frac{1}{2})}{N}\right) \end{aligned} \tag{5.8.7}$$

then the approximation formula

$$f(x) \approx \left[\sum_{k=1}^{N} c_k T_{k-1}(x) \right] - \frac{1}{2} c_1 \tag{5.8.8}$$

is *exact* for x equal to all of the N zeros of $T_N(x)$.

For a fixed N, equation (5.8.8) is a polynomial in x which approximates the function $f(x)$ in the interval $[-1, 1]$ (where all the zeros of $T_N(x)$ are located). Why is this particular approximating polynomial better than any other one, exact on some other set of N points? The answer is *not* that (5.8.8) is necessarily more accurate than some other approximating polynomial of the same order N (for some specified definition of "accurate"), but rather that (5.8.8) can be truncated to a polynomial of *lower* degree $m \ll N$ in a very graceful way, one that *does* yield the "most accurate" approximation of degree m (in a sense that can be made precise). Suppose N is so large that (5.8.8) is virtually a perfect approximation of $f(x)$. Now consider the truncated approximation

$$f(x) \approx \left[\sum_{k=1}^{m} c_k T_{k-1}(x) \right] - \frac{1}{2} c_1 \tag{5.8.9}$$

with the same c_j's, computed from (5.8.7). Since the $T_k(x)$'s are all bounded between ± 1, the difference between (5.8.9) and (5.8.8) can be no larger than the sum of the neglected c_k's ($k = m+1, \ldots, N$). In fact, if the c_k's are rapidly decreasing (which is the typical case), then the error is dominated by $c_{m+1} T_m(x)$, an oscillatory function with $m+1$ equal extrema distributed smoothly over the interval $[-1, 1]$. This smooth spreading out of the error is a very important property: The Chebyshev approximation (5.8.9) is very nearly the same polynomial as that holy grail of approximating polynomials the *minimax polynomial*, which (among all polynomials of the same degree) has the smallest maximum deviation from the true function $f(x)$. The minimax polynomial is very difficult to find; the Chebyshev approximating polynomial is almost identical and is very easy to compute!

So, given some (perhaps difficult) means of computing the function $f(x)$, we now need algorithms for implementing (5.8.7) and (after inspection of the resulting c_k's and choice of a truncating value m) evaluating (5.8.9). The latter equation then becomes an easy way of computing $f(x)$ for all subsequent time.

The first of these tasks is straightforward. A generalization of equation (5.8.7) that is here implemented is to allow the range of approximation to be between two arbitrary limits a and b, instead of just -1 to 1. This is effected by a change of variable

$$y \equiv \frac{x - \frac{1}{2}(b+a)}{\frac{1}{2}(b-a)} \tag{5.8.10}$$

and by the approximation of $f(x)$ by a Chebyshev polynomial in y.

```
SUBROUTINE chebft(a,b,c,n,func)
INTEGER n,NMAX
REAL a,b,c(n),func,PI
EXTERNAL func
PARAMETER (NMAX=50, PI=3.14159265358979d0)
```

Chebyshev fit: Given a function `func`, lower and upper limits of the interval [a,b], and a maximum degree `n`, this routine computes the `n` coefficients c_k such that `func`$(x) \approx$

```
    [Σ_{k=1}^{n} c_k T_{k-1}(y)] - c_1/2, where y and x are related by (5.8.10). This routine is to be
    used with moderately large n (e.g., 30 or 50), the array of c's subsequently to be truncated
    at the smaller value m such that c_{m+1} and subsequent elements are negligible.
    Parameters: Maximum expected value of n, and π.
INTEGER j,k
REAL bma,bpa,fac,y,f(NMAX)
DOUBLE PRECISION sum
bma=0.5*(b-a)
bpa=0.5*(b+a)
do 11 k=1,n                          We evaluate the function at the n points required by (5.8.7).
    y=cos(PI*(k-0.5)/n)
    f(k)=func(y*bma+bpa)
enddo 11
fac=2./n
do 13 j=1,n
    sum=0.d0                         We will accumulate the sum in double precision, a nicety that
    do 12 k=1,n                          you can ignore.
        sum=sum+f(k)*cos((PI*(j-1))*((k-0.5d0)/n))
    enddo 12
    c(j)=fac*sum
enddo 13
return
END
```

(If you find that the execution time of `chebft` is dominated by the calculation of N^2 cosines, rather than by the N evaluations of your function, then you should look ahead to §12.3, especially equation 12.3.22, which shows how fast cosine transform methods can be used to evaluate equation 5.8.7.)

Now that we have the Chebyshev coefficients, how do we evaluate the approximation? One could use the recurrence relation of equation (5.8.2) to generate values for $T_k(x)$ from $T_0 = 1, T_1 = x$, while also accumulating the sum of (5.8.9). It is better to use Clenshaw's recurrence formula (§5.5), effecting the two processes simultaneously. Applied to the Chebyshev series (5.8.9), the recurrence is

$$\begin{aligned} d_{m+2} &\equiv d_{m+1} \equiv 0 \\ d_j &= 2xd_{j+1} - d_{j+2} + c_j \qquad j = m, m-1, \ldots, 2 \\ f(x) &\equiv d_0 = xd_2 - d_3 + \frac{1}{2}c_1 \end{aligned} \tag{5.8.11}$$

```
FUNCTION chebev(a,b,c,m,x)
INTEGER m
REAL chebev,a,b,x,c(m)
    Chebyshev evaluation: All arguments are input. c(1:m) is an array of Chebyshev coeffi-
    cients, the first m elements of c output from chebft (which must have been called with
    the same a and b). The Chebyshev polynomial Σ_{k=1}^{m} c_k T_{k-1}(y) - c_1/2 is evaluated at a
    point y = [x - (b + a)/2]/[(b - a)/2], and the result is returned as the function value.
INTEGER j
REAL d,dd,sv,y,y2
if ((x-a)*(x-b).gt.0.) pause 'x not in range in chebev'
d=0.
dd=0.
y=(2.*x-a-b)/(b-a)                   Change of variable.
y2=2.*y
do 11 j=m,2,-1                       Clenshaw's recurrence.
    sv=d
    d=y2*d-dd+c(j)
```

```
    dd=sv
enddo 11
chebev=y*d-dd+0.5*c(1)        Last step is different.
return
END
```

If we are approximating an *even* function on the interval $[-1, 1]$, its expansion will involve only even Chebyshev polynomials. It is wasteful to call `chebev` with all the odd coefficients zero [1]. Instead, using the half-angle identity for the cosine in equation (5.8.1), we get the relation

$$T_{2n}(x) = T_n(2x^2 - 1) \tag{5.8.12}$$

Thus we can evaluate a series of even Chebyshev polynomials by calling `chebev` with the even coefficients stored consecutively in the array `c`, but with the argument x replaced by $2x^2 - 1$.

An odd function will have an expansion involving only odd Chebysev polynomials. It is best to rewrite it as an expansion for the function $f(x)/x$, which involves only even Chebyshev polynomials. This will give accurate values for $f(x)/x$ near $x = 0$. The coefficients c_n' for $f(x)/x$ can be found from those for $f(x)$ by recurrence:

$$\begin{aligned} c_{N+1}' &= 0 \\ c_{n-1}' &= 2c_n - c_{n+1}', \qquad n = N, N-2, \ldots \end{aligned} \tag{5.8.13}$$

Equation (5.8.13) follows from the recurrence relation in equation (5.8.2).

If you insist on evaluating an odd Chebyshev series, the efficient way is to once again use `chebev` with x replaced by $y = 2x^2 - 1$, and with the odd coefficients stored consecutively in the array `c`. Now, however, you must also change the last formula in equation (5.8.11) to be

$$f(x) = x[(2y - 1)d_2 - d_3 + c_1] \tag{5.8.14}$$

and change the corresponding line in `chebev`.

CITED REFERENCES AND FURTHER READING:

Clenshaw, C.W. 1962, *Mathematical Tables*, vol. 5, National Physical Laboratory, (London: H.M. Stationery Office). [1]

Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library), Chapter 8.

Dahlquist, G., and Bjorck, A. 1974, *Numerical Methods* (Englewood Cliffs, NJ: Prentice-Hall), §4.4.1, p. 104.

Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), §6.5.2, p. 334.

Carnahan, B., Luther, H.A., and Wilkes, J.O. 1969, *Applied Numerical Methods* (New York: Wiley), §1.10, p. 39.

5.9 Derivatives or Integrals of a Chebyshev-approximated Function

If you have obtained the Chebyshev coefficients that approximate a function in a certain range (e.g., from `chebft` in §5.8), then it is a simple matter to transform them to Chebyshev coefficients corresponding to the derivative or integral of the function. Having done this, you can evaluate the derivative or integral just as if it were a function that you had Chebyshev-fitted *ab initio*.

The relevant formulas are these: If c_i, $i = 1, \ldots, m$ are the coefficients that approximate a function f in equation (5.8.9), C_i are the coefficients that approximate the indefinite integral of f, and c'_i are the coefficients that approximate the derivative of f, then

$$C_i = \frac{c_{i-1} - c_{i+1}}{2(i-1)} \qquad (i > 1) \tag{5.9.1}$$

$$c'_{i-1} = c'_{i+1} + 2(i-1)c_i \qquad (i = m-1, m-2, \ldots, 2) \tag{5.9.2}$$

Equation (5.9.1) is augmented by an arbitrary choice of C_1, corresponding to an arbitrary constant of integration. Equation (5.9.2), which is a recurrence, is started with the values $c'_m = c'_{m+1} = 0$, corresponding to no information about the $m+1$st Chebyshev coefficient of the original function f.

Here are routines for implementing equations (5.9.1) and (5.9.2).

```
SUBROUTINE chder(a,b,c,cder,n)
INTEGER n
REAL a,b,c(n),cder(n)
    Given a,b,c(1:n), as output from routine chebft §5.8, and given n, the desired degree
    of approximation (length of c to be used), this routine returns the array cder(1:n), the
    Chebyshev coefficients of the derivative of the function whose coefficients are c(1:n).
INTEGER j
REAL con
cder(n)=0.                                  n and n-1 are special cases.
cder(n-1)=2*(n-1)*c(n)
if(n.ge.3)then
    do 11 j=n-2,1,-1
        cder(j)=cder(j+2)+2*j*c(j+1)        Equation (5.9.2).
    enddo 11
endif
con=2./(b-a)
do 12 j=1,n                                 Normalize to the interval b-a.
    cder(j)=cder(j)*con
enddo 12
return
END
```

```
SUBROUTINE chint(a,b,c,cint,n)
INTEGER n
REAL a,b,c(n),cint(n)
    Given a,b,c(1:n), as output from routine chebft §5.8, and given n, the desired degree
    of approximation (length of c to be used), this routine returns the array cint(1:n), the
```

```
    Chebyshev coefficients of the integral of the function whose coefficients are c. The constant
    of integration is set so that the integral vanishes at a.
INTEGER j
REAL con,fac,sum
con=0.25*(b-a)                    Factor that normalizes to the interval b-a.
sum=0.                            Accumulates the constant of integration.
fac=1.                            Will equal ±1.
do 11 j=2,n-1
    cint(j)=con*(c(j-1)-c(j+1))/(j-1)          Equation (5.9.1).
    sum=sum+fac*cint(j)
    fac=-fac
enddo 11
cint(n)=con*c(n-1)/(n-1)          Special case of (5.9.1) for n.
sum=sum+fac*cint(n)
cint(1)=2.*sum                    Set the constant of integration.
return
END
```

Clenshaw-Curtis Quadrature

Since a smooth function's Chebyshev coefficients c_i decrease rapidly, generally exponentially, equation (5.9.1) is often quite efficient as the basis for a quadrature scheme. The routines `chebft` and `chint`, used in that order, can be followed by repeated calls to `chebev` if $\int_a^x f(x)dx$ is required for many different values of x in the range $a \le x \le b$.

If only the single definite integral $\int_a^b f(x)dx$ is required, then `chint` and `chebev` are replaced by the simpler formula, derived from equation (5.9.1),

$$\int_a^b f(x)dx = (b-a)\left[\frac{1}{2}c_1 - \frac{1}{3}c_3 - \frac{1}{15}c_5 - \cdots - \frac{1}{(2k+1)(2k-1)}c_{2k+1} - \cdots\right] \tag{5.9.3}$$

where the c_i's are as returned by `chebft`. The series can be truncated when c_{2k+1} becomes negligible, and the first neglected term gives an error estimate.

This scheme is known as *Clenshaw-Curtis quadrature* [1]. It is often combined with an adaptive choice of N, the number of Chebyshev coefficients calculated via equation (5.8.7), which is also the number of function evaluations of $f(x)$. If a modest choice of N does not give a sufficiently small c_{2k+1} in equation (5.9.3), then a larger value is tried. In this adaptive case, it is even better to replace equation (5.8.7) by the so-called "trapezoidal" or Gauss-Lobatto (§4.5) variant,

$$c_j = \frac{2}{N}{\sum_{k=0}^{N}}'' f\left[\cos\left(\frac{\pi k}{N}\right)\right]\cos\left(\frac{\pi(j-1)k}{N}\right) \qquad j = 1,\ldots,N \tag{5.9.4}$$

where (N.B.!) the two primes signify that the first and last terms in the sum are to be multiplied by 1/2. If N is doubled in equation (5.9.4), then half of the new function evaluation points are identical to the old ones, allowing the previous function evaluations to be reused. This feature, plus the analytic weights and abscissas (cosine functions in 5.9.4), give Clenshaw-Curtis quadrature an edge over high-order adaptive Gaussian quadrature (cf. §4.5), which the method otherwise resembles.

If your problem forces you to large values of N, you should be aware that equation (5.9.4) can be evaluated rapidly, and simultaneously for all the values of j, by a fast cosine transform. (See §12.3, especially equation 12.3.17.) (We already remarked that the nontrapezoidal form (5.8.7) can also be done by fast cosine methods, cf. equation 12.3.22.)

CITED REFERENCES AND FURTHER READING:

Goodwin, E.T. (ed.) 1961, *Modern Computing Methods*, 2nd ed. (New York: Philosophical Library), pp. 78–79.

Clenshaw, C.W., and Curtis, A.R. 1960, *Numerische Mathematik*, vol. 2, pp. 197–205. [1]

12.3 FFT of Real Functions, Sine and Cosine Transforms

It happens frequently that the data whose FFT is desired consist of real-valued samples f_j, $j = 0 \ldots N-1$. To use `four1`, we put these into a complex array with all imaginary parts set to zero. The resulting transform F_n, $n = 0 \ldots N-1$ satisfies ${F_{N-n}}^* = F_n$. Since this complex-valued array has real values for F_0 and $F_{N/2}$, and $(N/2)-1$ other independent values $F_1 \ldots F_{N/2-1}$, it has the same $2(N/2-1)+2 = N$ "degrees of freedom" as the original, real data set. However, the use of the full complex FFT algorithm for real data is inefficient, both in execution time and in storage required. You would think that there is a better way.

There are *two* better ways. The first is "mass production": Pack two separate real functions into the input array in such a way that their individual transforms can be separated from the result. This is implemented in the program `twofft` below. This may remind you of a one-cent sale, at which you are coerced to purchase two of an item when you only need one. However, remember that for correlations and convolutions the Fourier transforms of two functions are involved, and this is a handy way to do them both at once. The second method is to pack the real input array cleverly, without extra zeros, into a complex array of half its length. One then performs a complex FFT on this shorter length; the trick is then to get the required answer out of the result. This is done in the program `realft` below.

Transform of Two Real Functions Simultaneously

First we show how to exploit the symmetry of the transform F_n to handle two real functions at once: Since the input data f_j are real, the components of the discrete Fourier transform satisfy

$$F_{N-n} = (F_n)^* \tag{12.3.1}$$

where the asterisk denotes complex conjugation. By the same token, the discrete Fourier transform of a purely imaginary set of g_j's has the opposite symmetry.

$$G_{N-n} = -(G_n)^* \tag{12.3.2}$$

Therefore we can take the discrete Fourier transform of two real functions each of length N simultaneously by packing the two data arrays as the real and imaginary parts, respectively, of the complex input array of `four1`. Then the resulting transform array can be unpacked into two complex arrays with the aid of the two symmetries. Routine `twofft` works out these ideas.

```
      SUBROUTINE twofft(data1,data2,fft1,fft2,n)
      INTEGER n
      REAL data1(n),data2(n)
      COMPLEX fft1(n),fft2(n)
C     USES four1
```

Given two real input arrays `data1(1:n)` and `data2(1:n)`, this routine calls `four1` and returns two complex output arrays, `fft1(1:n)` and `fft2(1:n)`, each of complex length `n`

```
    (i.e., real length 2*n), which contain the discrete Fourier transforms of the respective data
    arrays. n MUST be an integer power of 2.
      INTEGER j,n2
      COMPLEX h1,h2,c1,c2
      c1=cmplx(0.5,0.0)
      c2=cmplx(0.0,-0.5)
      do 11 j=1,n
        fft1(j)=cmplx(data1(j),data2(j))          Pack the two real arrays into one complex
      enddo 11                                      array.
      call four1(fft1,n,1)                      Transform the complex array.
      fft2(1)=cmplx(aimag(fft1(1)),0.0)
      fft1(1)=cmplx(real(fft1(1)),0.0)
      n2=n+2
      do 12 j=2,n/2+1
        h1=c1*(fft1(j)+conjg(fft1(n2-j)))       Use symmetries to separate the two trans-
        h2=c2*(fft1(j)-conjg(fft1(n2-j)))           forms.
        fft1(j)=h1                              Ship them out in two complex arrays.
        fft1(n2-j)=conjg(h1)
        fft2(j)=h2
        fft2(n2-j)=conjg(h2)
      enddo 12
      return
      END
```

What about the reverse process? Suppose you have two complex transform arrays, each of which has the symmetry (12.3.1), so that you know that the inverses of both transforms are real functions. Can you invert both in a single FFT? This is even easier than the other direction. Use the fact that the FFT is linear and form the sum of the first transform plus i times the second. Invert using `four1` with `isign` $= -1$. The real and imaginary parts of the resulting complex array are the two desired real functions.

FFT of Single Real Function

To implement the second method, which allows us to perform the FFT of a *single* real function without redundancy, we split the data set in half, thereby forming two real arrays of half the size. We can apply the program above to these two, but of course the result will not be the transform of the original data. It will be a schizophrenic combination of two transforms, each of which has half of the information we need. Fortunately, this schizophrenia is treatable. It works like this:

The right way to split the original data is to take the even-numbered f_j as one data set, and the odd-numbered f_j as the other. The beauty of this is that we can take the original real array and treat it as a complex array h_j of half the length. The first data set is the real part of this array, and the second is the imaginary part, as prescribed for `twofft`. No repacking is required. In other words $h_j = f_{2j} + if_{2j+1}, \quad j = 0, \ldots, N/2 - 1$. We submit this to `four1`, and it will return a complex array $H_n = F_n^e + iF_n^o, \quad n = 0, \ldots, N/2 - 1$ with

$$\begin{aligned} F_n^e &= \sum_{k=0}^{N/2-1} f_{2k}\, e^{2\pi ikn/(N/2)} \\ F_n^o &= \sum_{k=0}^{N/2-1} f_{2k+1}\, e^{2\pi ikn/(N/2)} \end{aligned} \tag{12.3.3}$$

The discussion of program `twofft` tells you how to separate the two transforms F_n^e and F_n^o out of H_n. How do you work them into the transform F_n of the original data set f_j? Simply glance back at equation (12.2.3):

$$F_n = F_n^e + e^{2\pi i n/N} F_n^o \qquad n = 0, \ldots, N-1 \tag{12.3.4}$$

Expressed directly in terms of the transform H_n of our real (masquerading as complex) data set, the result is

$$F_n = \frac{1}{2}(H_n + H_{N/2-n}{}^*) - \frac{i}{2}(H_n - H_{N/2-n}{}^*)e^{2\pi i n/N} \qquad n = 0, \ldots, N-1 \tag{12.3.5}$$

A few remarks:

- Since $F_{N-n}{}^* = F_n$ there is no point in saving the entire spectrum. The positive frequency half is sufficient and can be stored in the same array as the original data. The operation can, in fact, be done in place.
- Even so, we need values H_n, $n = 0, \ldots, N/2$ whereas `four1` returns only the values $n = 0, \ldots, N/2 - 1$. Symmetry to the rescue, $H_{N/2} = H_0$.
- The values F_0 and $F_{N/2}$ are real and independent. In order to actually get the entire F_n in the original array space, it is convenient to return $F_{N/2}$ as the imaginary part of F_0.
- Despite its complicated form, the process above is invertible. First peel $F_{N/2}$ out of F_0. Then construct

$$\begin{aligned} F_n^e &= \frac{1}{2}(F_n + F_{N/2-n}^*) \\ F_n^o &= \frac{1}{2}e^{-2\pi i n/N}(F_n - F_{N/2-n}^*) \end{aligned} \qquad n = 0, \ldots, N/2-1 \tag{12.3.6}$$

  and use `four1` to find the inverse transform of $H_n = F_n^{(1)} + iF_n^{(2)}$. Surprisingly, the actual algebraic steps are virtually identical to those of the forward transform.

Here is a representation of what we have said:

```
      SUBROUTINE realft(data,n,isign)
      INTEGER isign,n
      REAL data(n)
C     USES four1
        Calculates the Fourier transform of a set of n real-valued data points. Replaces this data
        (which is stored in array data(1:n)) by the positive frequency half of its complex Fourier
        transform. The real-valued first and last components of the complex transform are returned
        as elements data(1) and data(2), respectively. n must be a power of 2. This routine
        also calculates the inverse transform of a complex data array if it is the transform of real
        data. (Result in this case must be multiplied by 2/n.)
      INTEGER i,i1,i2,i3,i4,n2p3
      REAL c1,c2,h1i,h1r,h2i,h2r,wis,wrs
      DOUBLE PRECISION theta,wi,wpi,wpr,
     *     wr,wtemp                    Double precision for the trigonometric recurrences.
      theta=3.141592653589793d0/dble(n/2)   Initialize the recurrence.
      c1=0.5
      if (isign.eq.1) then
        c2=-0.5
        call four1(data,n/2,+1)           The forward transform is here.
```

```
      else
        c2=0.5                                  Otherwise set up for an inverse transform.
        theta=-theta
      endif
      wpr=-2.0d0*sin(0.5d0*theta)**2
      wpi=sin(theta)
      wr=1.0d0+wpr
      wi=wpi
      n2p3=n+3
      do 11 i=2,n/4                             Case i=1 done separately below.
        i1=2*i-1
        i2=i1+1
        i3=n2p3-i2
        i4=i3+1
        wrs=sngl(wr)
        wis=sngl(wi)
        h1r=c1*(data(i1)+data(i3))              The two separate transforms are separated out of
        h1i=c1*(data(i2)-data(i4))                data.
        h2r=-c2*(data(i2)+data(i4))
        h2i=c2*(data(i1)-data(i3))
        data(i1)=h1r+wrs*h2r-wis*h2i            Here they are recombined to form the true trans-
        data(i2)=h1i+wrs*h2i+wis*h2r              form of the original real data.
        data(i3)=h1r-wrs*h2r+wis*h2i
        data(i4)=-h1i+wrs*h2i+wis*h2r
        wtemp=wr                                The recurrence.
        wr=wr*wpr-wi*wpi+wr
        wi=wi*wpr+wtemp*wpi+wi
      enddo 11
      if (isign.eq.1) then
        h1r=data(1)
        data(1)=h1r+data(2)
        data(2)=h1r-data(2)                     Squeeze the first and last data together to get
      else                                        them all within the original array.
        h1r=data(1)
        data(1)=c1*(h1r+data(2))
        data(2)=c1*(h1r-data(2))
        call four1(data,n/2,-1)                 This is the inverse transform for the case isign=-1.
      endif
      return
      END
```

Fast Sine and Cosine Transforms

Among their other uses, the Fourier transforms of functions can be used to solve differential equations (see §19.4). The most common boundary conditions for the solutions are 1) they have the value zero at the boundaries, or 2) their derivatives are zero at the boundaries. In the first instance, the natural transform to use is the *sine* transform, given by

$$F_k = \sum_{j=1}^{N-1} f_j \sin(\pi jk/N) \qquad \text{sine transform} \tag{12.3.7}$$

where f_j, $j = 0, \ldots, N-1$ is the data array, and $f_0 \equiv 0$.

At first blush this appears to be simply the imaginary part of the discrete Fourier transform. However, the argument of the sine differs by a factor of two from the value that would make this so. The sine transform uses *sines only* as a complete set

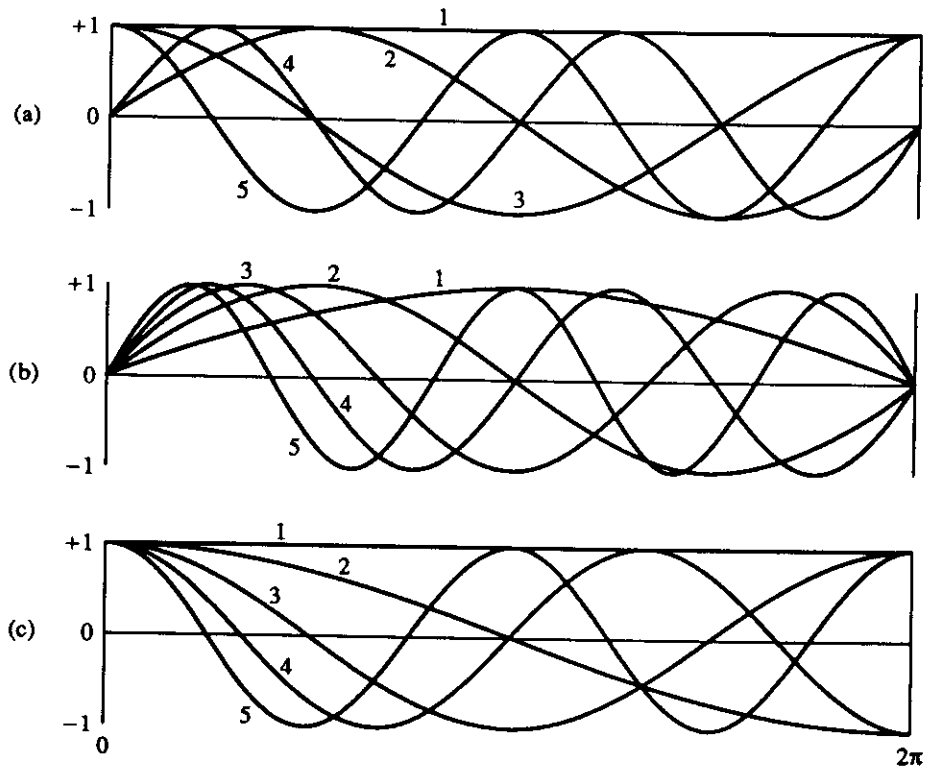


Figure 12.3.1. Basis functions used by the Fourier transform (a), sine transform (b), and cosine transform (c), are plotted. The first five basis functions are shown in each case. (For the Fourier transform, the real and imaginary parts of the basis functions are both shown.) While some basis functions occur in more than one transform, the basis sets are distinct. For example, the sine transform functions labeled (1), (3), (5) are not present in the Fourier basis. Any of the three sets can expand any function in the interval shown; however, the sine or cosine transform best expands functions matching the boundary conditions of the respective basis functions, namely zero function values for sine, zero derivatives for cosine.

of functions in the interval from 0 to 2π, and, as we shall see, the cosine transform uses *cosines only*. By contrast, the normal FFT uses both sines and cosines, but only half as many of each. (See Figure 12.3.1.)

The expression (12.3.7) can be "force-fit" into a form that allows its calculation via the FFT. The idea is to extend the given function rightward past its last tabulated value. We extend the data to twice their length in such a way as to make them an *odd* function about $j = N$, with $f_N = 0$,

$$f_{2N-j} \equiv -f_j \qquad j = 0, \ldots, N-1 \tag{12.3.8}$$

Consider the FFT of this extended function:

$$F_k = \sum_{j=0}^{2N-1} f_j e^{2\pi i j k/(2N)} \tag{12.3.9}$$

The half of this sum from $j = N$ to $j = 2N - 1$ can be rewritten with the

substitution $j' = 2N - j$

$$\begin{aligned}\sum_{j=N}^{2N-1} f_j e^{2\pi ijk/(2N)} &= \sum_{j'=1}^{N} f_{2N-j'} e^{2\pi i(2N-j')k/(2N)} \\ &= -\sum_{j'=0}^{N-1} f_{j'} e^{-2\pi ij'k/(2N)}\end{aligned} \tag{12.3.10}$$

so that

$$\begin{aligned}F_k &= \sum_{j=0}^{N-1} f_j \left[e^{2\pi ijk/(2N)} - e^{-2\pi ijk/(2N)}\right] \\ &= 2i \sum_{j=0}^{N-1} f_j \sin(\pi jk/N)\end{aligned} \tag{12.3.11}$$

Thus, up to a factor $2i$ we get the sine transform from the FFT of the extended function.

This method introduces a factor of two inefficiency into the computation by extending the data. This inefficiency shows up in the FFT output, which has zeros for the real part of every element of the transform. For a one-dimensional problem, the factor of two may be bearable, especially in view of the simplicity of the method. When we work with partial differential equations in two or three dimensions, though, the factor becomes four or eight, so efforts to eliminate the inefficiency are well rewarded.

From the original real data array f_j we will construct an auxiliary array y_j and apply to it the routine `realft`. The output will then be used to construct the desired transform. For the sine transform of data f_j, $j = 1, \ldots, N-1$, the auxiliary array is

$$\begin{aligned}y_0 &= 0 \\ y_j &= \sin(j\pi/N)(f_j + f_{N-j}) + \frac{1}{2}(f_j - f_{N-j}) \qquad j = 1, \ldots, N-1\end{aligned} \tag{12.3.12}$$

This array is of the same dimension as the original. Notice that the first term is symmetric about $j = N/2$ and the second is antisymmetric. Consequently, when `realft` is applied to y_j, the result has real parts R_k and imaginary parts I_k given by

$$\begin{aligned}R_k &= \sum_{j=0}^{N-1} y_j \cos(2\pi jk/N) \\ &= \sum_{j=1}^{N-1} (f_j + f_{N-j}) \sin(j\pi/N) \cos(2\pi jk/N) \\ &= \sum_{j=0}^{N-1} 2f_j \sin(j\pi/N) \cos(2\pi jk/N) \\ &= \sum_{j=0}^{N-1} f_j \left[\sin\frac{(2k+1)j\pi}{N} - \sin\frac{(2k-1)j\pi}{N}\right]\end{aligned}$$

$$= F_{2k+1} - F_{2k-1} \tag{12.3.13}$$

$$\begin{aligned} I_k &= \sum_{j=0}^{N-1} y_j \sin(2\pi jk/N) \\ &= \sum_{j=1}^{N-1} (f_j - f_{N-j}) \frac{1}{2} \sin(2\pi jk/N) \\ &= \sum_{j=0}^{N-1} f_j \sin(2\pi jk/N) \\ &= F_{2k} \end{aligned} \tag{12.3.14}$$

Therefore F_k can be determined as follows:

$$F_{2k} = I_k \qquad F_{2k+1} = F_{2k-1} + R_k \qquad k = 0, \ldots, (N/2 - 1) \tag{12.3.15}$$

The even terms of F_k are thus determined very directly. The odd terms require a recursion, the starting point of which follows from setting $k = 0$ in equation (12.3.15) and using $F_1 = -F_{-1}$:

$$F_1 = \frac{1}{2} R_0 \tag{12.3.16}$$

The implementing program is

```
      SUBROUTINE sinft(y,n)
      INTEGER n
      REAL y(n)
C     USES realft
        Calculates the sine transform of a set of n real-valued data points stored in array y(1:n).
        The number n must be a power of 2. On exit y is replaced by its transform. This program,
        without changes, also calculates the inverse sine transform, but in this case the output array
        should be multiplied by 2/n.
      INTEGER j
      REAL sum,y1,y2
      DOUBLE PRECISION theta,wi,wpi,wpr,
     *     wr,wtemp                          Double precision in the trigonometric recurrences.
      theta=3.141592653589793d0/dble(n)      Initialize the recurrence.
      wr=1.0d0
      wi=0.0d0
      wpr=-2.0d0*sin(0.5d0*theta)**2
      wpi=sin(theta)
      y(1)=0.0
      do 11 j=1,n/2
        wtemp=wr
        wr=wr*wpr-wi*wpi+wr                  Calculate the sine for the auxiliary array.
        wi=wi*wpr+wtemp*wpi+wi               The cosine is needed to continue the recurrence.
        y1=wi*(y(j+1)+y(n-j+1))              Construct the auxiliary array.
        y2=0.5*(y(j+1)-y(n-j+1))
        y(j+1)=y1+y2                         Terms j and N − j are related.
        y(n-j+1)=y1-y2
      enddo 11
      call realft(y,n,+1)                    Transform the auxiliary array.
      sum=0.0
      y(1)=0.5*y(1)                          Initialize the sum used for odd terms below.
      y(2)=0.0
```

```
do 12 j=1,n-1,2
    sum=sum+y(j)
    y(j)=y(j+1)
    y(j+1)=sum
enddo 12
return
END
```

Even terms in the transform are determined directly.
Odd terms are determined by this running sum.

The sine transform, curiously, is its own inverse. If you apply it twice, you get the original data, but multiplied by a factor of $N/2$.

The other common boundary condition for differential equations is that the derivative of the function is zero at the boundary. In this case the natural transform is the *cosine* transform. There are several possible ways of defining the transform. Each can be thought of as resulting from a different way of extending a given array to create an even array of double the length, and/or from whether the extended array contains $2N-1$, $2N$, or some other number of points. In practice, only two of the numerous possibilities are useful so we will restrict ourselves to just these two.

The first form of the cosine transform uses $N+1$ data points:

$$F_k = \frac{1}{2}[f_0 + (-1)^k f_N] + \sum_{j=1}^{N-1} f_j \cos(\pi jk/N) \tag{12.3.17}$$

It results from extending the given array to an even array about $j = N$, with

$$f_{2N-j} = f_j, \qquad j = 0, \ldots, N-1 \tag{12.3.18}$$

If you substitute this extended array into equation (12.3.9), and follow steps analogous to those leading up to equation (12.3.11), you will find that the Fourier transform is just twice the cosine transform (12.3.17). Another way of thinking about the formula (12.3.17) is to notice that it is the Chebyshev Gauss-Lobatto quadrature formula (see §4.5), often used in Clenshaw-Curtis adaptive quadrature (see §5.9, equation 5.9.4).

Once again the transform can be computed without the factor of two inefficiency. In this case the auxiliary function is

$$y_j = \frac{1}{2}(f_j + f_{N-j}) - \sin(j\pi/N)(f_j - f_{N-j}) \qquad j = 0, \ldots, N-1 \tag{12.3.19}$$

Instead of equation (12.3.15), `realft` now gives

$$F_{2k} = R_k \qquad F_{2k+1} = F_{2k-1} + I_k \qquad k = 0, \ldots, (N/2-1) \tag{12.3.20}$$

The starting value for the recursion for odd k in this case is

$$F_1 = \frac{1}{2}(f_0 - f_N) + \sum_{j=1}^{N-1} f_j \cos(j\pi/N) \tag{12.3.21}$$

This sum does not appear naturally among the R_k and I_k, and so we accumulate it during the generation of the array y_j.

Once again this transform is its own inverse, and so the following routine works for both directions of the transformation. Note that although this form of the cosine transform has $N+1$ input and output values, it passes an array only of length N to `realft`.

```
      SUBROUTINE cosft1(y,n)
      INTEGER n
      REAL y(n+1)
C     USES realft
        Calculates the cosine transform of a set y(1:n+1) of real-valued data points. The trans-
        formed data replace the original data in array y. n must be a power of 2. This program,
        without changes, also calculates the inverse cosine transform, but in this case the output
        array should be multiplied by 2/n.
      INTEGER j
      REAL sum,y1,y2
      DOUBLE PRECISION theta,wi,wpi,wpr,wr,wtemp        For trig. recurrences.
      theta=3.14159265358979d0/n          Initialize the recurrence.
      wr=1.0d0
      wi=0.0d0
      wpr=-2.0d0*sin(0.5d0*theta)**2
      wpi=sin(theta)
      sum=0.5*(y(1)-y(n+1))
      y(1)=0.5*(y(1)+y(n+1))
      do 11 j=1,n/2-1                     j=n/2 unnecessary since y(n/2+1) unchanged.
        wtemp=wr
        wr=wr*wpr-wi*wpi+wr               Carry out the recurrence.
        wi=wi*wpr+wtemp*wpi+wi
        y1=0.5*(y(j+1)+y(n-j+1))          Calculate the auxiliary function.
        y2=(y(j+1)-y(n-j+1))
        y(j+1)=y1-wi*y2                   The values for j and N - j are related.
        y(n-j+1)=y1+wi*y2
        sum=sum+wr*y2                     Carry along this sum for later use in unfolding the
      enddo 11                              transform.
      call realft(y,n,+1)                 Calculate the transform of the auxiliary function.
      y(n+1)=y(2)
      y(2)=sum                            sum is the value of F_1 in equation (12.3.21).
      do 12 j=4,n,2
        sum=sum+y(j)                      Equation (12.3.20).
        y(j)=sum
      enddo 12
      return
      END
```

The second important form of the cosine transform is defined by

$$F_k = \sum_{j=0}^{N-1} f_j \cos \frac{\pi k(j+\frac{1}{2})}{N} \tag{12.3.22}$$

with inverse

$$f_j = \frac{2}{N} \sum_{k=0}^{N-1}{}' F_k \cos \frac{\pi k(j+\frac{1}{2})}{N} \tag{12.3.23}$$

Here the prime on the summation symbol means that the term for $k = 0$ has a coefficient of $\frac{1}{2}$ in front. This form arises by extending the given data, defined for $j = 0, \ldots, N-1$, to $j = N, \ldots, 2N-1$ in such a way that it is even about the point $N - \frac{1}{2}$ and periodic. (It is therefore also even about $j = -\frac{1}{2}$.) The form (12.3.23) is related to Gauss-Chebyshev quadrature (see equation 4.5.19), to Chebyshev approximation (§5.8, equation 5.8.7), and Clenshaw-Curtis quadrature (§5.9).

This form of the cosine transform is useful when solving differential equations on "staggered" grids, where the variables are centered midway between mesh points. It is also the standard form in the field of data compression and image processing.

The auxiliary function used in this case is similar to equation (12.3.19):

$$y_j = \frac{1}{2}(f_j + f_{N-j-1}) - \sin\frac{\pi(j+\frac{1}{2})}{N}(f_j - f_{N-j-1}) \qquad j = 0, \ldots, N-1 \tag{12.3.24}$$

Carrying out the steps similar to those used to get from (12.3.12) to (12.3.15), we find

$$F_{2k} = \cos\frac{\pi k}{N}R_k - \sin\frac{\pi k}{N}I_k \tag{12.3.25}$$

$$F_{2k-1} = \sin\frac{\pi k}{N}R_k + \cos\frac{\pi k}{N}I_k + F_{2k+1} \tag{12.3.26}$$

Note that equation (12.3.26) gives

$$F_{N-1} = \frac{1}{2}R_{N/2} \tag{12.3.27}$$

Thus the even components are found directly from (12.3.25), while the odd components are found by recursing (12.3.26) down from $k = N/2 - 1$, using (12.3.27) to start.

Since the transform is not self-inverting, we have to reverse the above steps to find the inverse. Here is the routine:

```
      SUBROUTINE cosft2(y,n,isign)
      INTEGER isign,n
      REAL y(n)
C     USES realft
        Calculates the "staggered" cosine transform of a set y(1:n) of real-valued data points.
        The transformed data replace the original data in array y. n must be a power of 2. Set
        isign to +1 for a transform, and to -1 for an inverse transform. For an inverse transform,
        the output array should be multiplied by 2/n.
      INTEGER i
      REAL sum,sum1,y1,y2,ytemp
      DOUBLE PRECISION theta,wi,wi1,wpi,wpr,wr,wr1,wtemp,PI
        Double precision for the trigonometric recurrences.
      PARAMETER (PI=3.141592653589793d0)
      theta=0.5d0*PI/n                    Initialize the recurrences.
      wr=1.0d0
      wi=0.0d0
      wr1=cos(theta)
      wi1=sin(theta)
      wpr=-2.0d0*wi1**2
      wpi=sin(2.d0*theta)
      if(isign.eq.1)then                  Forward transform.
        do 11 i=1,n/2
          y1=0.5*(y(i)+y(n-i+1))          Calculate the auxiliary function.
          y2=wi1*(y(i)-y(n-i+1))
          y(i)=y1+y2
          y(n-i+1)=y1-y2
          wtemp=wr1                       Carry out the recurrence.
          wr1=wr1*wpr-wi1*wpi+wr1
          wi1=wi1*wpr+wtemp*wpi+wi1
        enddo 11
        call realft(y,n,1)                Calculate the transform of the auxiliary function.
        do 12 i=3,n,2                     Even terms.
          wtemp=wr
          wr=wr*wpr-wi*wpi+wr
```

```
        wi=wi*wpr+wtemp*wpi+wi
        y1=y(i)*wr-y(i+1)*wi
        y2=y(i+1)*wr+y(i)*wi
        y(i)=y1
        y(i+1)=y2
    enddo 12
    sum=0.5*y(2)                    Initialize recurrence for odd terms with (1/2)R_{N/2}.
    do 13 i=n,2,-2                  Carry out recurrence for odd terms.
        sum1=sum
        sum=sum+y(i)
        y(i)=sum1
    enddo 13
else if(isign.eq.-1)then            Inverse transform.
    ytemp=y(n)
    do 14 i=n,4,-2                  Form difference of odd terms.
        y(i)=y(i-2)-y(i)
    enddo 14
    y(2)=2.0*ytemp
    do 15 i=3,n,2                   Calculate R_k and I_k.
        wtemp=wr
        wr=wr*wpr-wi*wpi+wr
        wi=wi*wpr+wtemp*wpi+wi
        y1=y(i)*wr+y(i+1)*wi
        y2=y(i+1)*wr-y(i)*wi
        y(i)=y1
        y(i+1)=y2
    enddo 15
    call realft(y,n,-1)
    do 16 i=1,n/2                   Invert auxiliary array.
        y1=y(i)+y(n-i+1)
        y2=(0.5/wi1)*(y(i)-y(n-i+1))
        y(i)=0.5*(y1+y2)
        y(n-i+1)=0.5*(y1-y2)
        wtemp=wr1
        wr1=wr1*wpr-wi1*wpi+wr1
        wi1=wi1*wpr+wtemp*wpi+wi1
    enddo 16
endif
return
END
```

An alternative way of implementing this algorithm is to form an auxiliary function by copying the even elements of f_j into the first $N/2$ locations, and the odd elements into the next $N/2$ elements in reverse order. However, it is not easy to implement the alternative algorithm without a temporary storage array and we prefer the above in-place algorithm.

Finally, we mention that there exist fast cosine transforms for small N that do not rely on an auxiliary function or use an FFT routine. Instead, they carry out the transform directly, often coded in hardware for fixed N of small dimension [1].

CITED REFERENCES AND FURTHER READING:

Brigham, E.O. 1974, *The Fast Fourier Transform* (Englewood Cliffs, NJ: Prentice-Hall), §10–10.

Sorensen, H.V., Jones, D.L., Heideman, M.T., and Burris, C.S. 1987, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, pp. 849–863.

Hou, H.S. 1987, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, pp. 1455–1461 [see for additional references].

Hockney, R.W. 1971, in *Methods in Computational Physics*, vol. 9 (New York: Academic Press).

Temperton, C. 1980, *Journal of Computational Physics*, vol. 34, pp. 314–329.

Clarke, R.J. 1985, *Transform Coding of Images*, (Reading, MA: Addison-Wesley).

Gonzalez, R.C., and Wintz, P. 1987, *Digital Image Processing*, (Reading, MA: Addison-Wesley).

Chen, W., Smith, C.H., and Fralick, S.C. 1977, *IEEE Transactions on Communications*, vol. COM-25, pp. 1004–1009. [1]

18. Integral Equations and Inverse Theory

18.0 Introduction

Many people, otherwise numerically knowledgable, imagine that the numerical solution of integral equations must be an extremely arcane topic, since, until recently, it was almost never treated in numerical analysis textbooks. Actually there is a large and growing literature on the numerical solution of integral equations; several monographs have by now appeared [1-3]. One reason for the sheer volume of this activity is that there are many different kinds of equations, each with many different possible pitfalls; often many different algorithms have been proposed to deal with a single case.

There is a close correspondence between linear integral equations, which specify linear, integral relations among functions in an infinite-dimensional function space, and plain old linear equations, which specify analogous relations among vectors in a finite-dimensional vector space. Because this correspondence lies at the heart of most computational algorithms, it is worth making it explicit as we recall how integral equations are classified.

Fredholm equations involve definite integrals with fixed upper and lower limits. An *inhomogeneous Fredholm equation of the first kind* has the form

$$g(t) = \int_a^b K(t,s)f(s)\,ds \tag{18.0.1}$$

Here $f(t)$ is the unknown function to be solved for, while $g(t)$ is a known "right-hand side." (In integral equations, for some odd reason, the familiar "right-hand side" is conventionally written on the left!) The function of two variables, $K(t,s)$ is called the *kernel*. Equation (18.0.1) is analogous to the matrix equation

$$\mathbf{K} \cdot \mathbf{f} = \mathbf{g} \tag{18.0.2}$$

whose solution is $\mathbf{f} = \mathbf{K}^{-1} \cdot \mathbf{g}$, where $\mathbf{K}^{-1}$ is the matrix inverse. Like equation (18.0.2), equation (18.0.1) has a unique solution whenever g is nonzero (the homogeneous case with $g = 0$ is almost never useful) and K is invertible. However, as we shall see, this latter condition is as often the exception as the rule.

The analog of the finite-dimensional eigenvalue problem

$$(\mathbf{K} - \sigma\mathbf{1}) \cdot \mathbf{f} = \mathbf{g} \tag{18.0.3}$$

is called a *Fredholm equation of the second kind*, usually written

$$f(t) = \lambda \int_a^b K(t,s) f(s)\, ds + g(t) \tag{18.0.4}$$

Again, the notational conventions do not exactly correspond: λ in equation (18.0.4) is $1/\sigma$ in (18.0.3), while $\mathbf{g}$ is $-g/\lambda$. If g (or $\mathbf{g}$) is zero, then the equation is said to be *homogeneous*. If the kernel $K(t,s)$ is bounded, then, like equation (18.0.3), equation (18.0.4) has the property that its homogeneous form has solutions for at most a denumerably infinite set $\lambda = \lambda_n$, $n = 1, 2, \ldots$, the *eigenvalues*. The corresponding solutions $f_n(t)$ are the *eigenfunctions*. The eigenvalues are real if the kernel is symmetric.

In the *inhomogeneous* case of nonzero g (or $\mathbf{g}$), equations (18.0.3) and (18.0.4) are soluble *except* when λ (or σ) is an eigenvalue — because the integral operator (or matrix) is singular then. In integral equations this dichotomy is called *the Fredholm alternative*.

Fredholm equations of the first kind are often extremely ill-conditioned. Applying the kernel to a function is generally a smoothing operation, so the solution, which requires inverting the operator, will be extremely sensitive to small changes or errors in the input. Smoothing often actually loses information, and there is no way to get it back in an inverse operation. Specialized methods have been developed for such equations, which are often called *inverse problems*. In general, a method must augment the information given with some prior knowledge of the nature of the solution. This prior knowledge is then used, in one way or another, to restore lost information. We will introduce such techniques in §18.4.

Inhomogeneous Fredholm equations of the second kind are much less often ill-conditioned. Equation (18.0.4) can be rewritten as

$$\int_a^b [K(t,s) - \sigma\delta(t-s)] f(s)\, ds = -\sigma g(t) \tag{18.0.5}$$

where $\delta(t-s)$ is a Dirac delta function (and where we have changed from λ to its reciprocal σ for clarity). If σ is large enough in magnitude, then equation (18.0.5) is, in effect, diagonally dominant and thus well-conditioned. Only if σ is small do we go back to the ill-conditioned case.

Homogeneous Fredholm equations of the second kind are likewise not particularly ill-posed. If K is a smoothing operator, then it will map many f's to zero, or near-zero; there will thus be a large number of degenerate or nearly degenerate eigenvalues around $\sigma = 0$ ($\lambda \to \infty$), but this will cause no particular computational difficulties. In fact, we can now see that the magnitude of σ needed to rescue the inhomogeneous equation (18.0.5) from an ill-conditioned fate is generally much *less* than that required for diagonal dominance. Since the σ term shifts all eigenvalues, it is enough that it be large enough to shift a smoothing operator's forest of near-zero eigenvalues away from zero, so that the resulting operator becomes invertible (except, of course, at the discrete eigenvalues).

Volterra equations are a special case of Fredholm equations with $K(t,s) = 0$ for $s > t$. Chopping off the unnecessary part of the integration, Volterra equations are written in a form where the upper limit of integration is the independent variable t.

The *Volterra equation of the first kind*

$$g(t) = \int_a^t K(t,s)f(s)\,ds \tag{18.0.6}$$

has as its analog the matrix equation (now written out in components)

$$\sum_{j=1}^{k} K_{kj} f_j = g_k \tag{18.0.7}$$

Comparing with equation (18.0.2), we see that the Volterra equation corresponds to a matrix $\mathbf{K}$ that is lower (i.e., left) triangular, with zero entries above the diagonal. As we know from Chapter 2, such matrix equations are trivially soluble by forward substitution. Techniques for solving Volterra equations are similarly straightforward. When experimental measurement noise does not dominate, Volterra equations of the first kind tend *not* to be ill-conditioned; the upper limit to the integral introduces a sharp step that conveniently spoils any smoothing properties of the kernel.

The Volterra equation of the second kind is written

$$f(t) = \int_a^t K(t,s)f(s)\,ds + g(t) \tag{18.0.8}$$

whose matrix analog is the equation

$$(\mathbf{K} - \mathbf{1}) \cdot \mathbf{f} = \mathbf{g} \tag{18.0.9}$$

with $\mathbf{K}$ lower triangular. The reason there is no λ in these equations is that (i) in the inhomogeneous case (nonzero g) it can be absorbed into K, while (ii) in the homogeneous case ($g = 0$), it is a theorem that Volterra equations of the second kind with bounded kernels have no eigenvalues with square-integrable eigenfunctions.

We have specialized our definitions to the case of linear integral equations. The integrand in a nonlinear version of equation (18.0.1) or (18.0.6) would be $K(t,s,f(s))$ instead of $K(t,s)f(s)$; a nonlinear version of equation (18.0.4) or (18.0.8) would have an integrand $K(t,s,f(t),f(s))$. Nonlinear Fredholm equations are considerably more complicated than their linear counterparts. Fortunately, they do not occur as frequently in practice and we shall by and large ignore them in this chapter. By contrast, solving nonlinear Volterra equations usually involves only a slight modification of the algorithm for linear equations, as we shall see.

Almost all methods for solving integral equations numerically make use of *quadrature rules*, frequently Gaussian quadratures. This would be a good time for you to go back and review §4.5, especially the advanced material towards the end of that section.

In the sections that follow, we first discuss Fredholm equations of the second kind with smooth kernels (§18.1). Nontrivial quadrature rules come into the discussion, but we will be dealing with well-conditioned systems of equations. We then return to Volterra equations (§18.2), and find that simple and straightforward methods are generally satisfactory for these equations.

In §18.3 we discuss how to proceed in the case of singular kernels, focusing largely on Fredholm equations (both first and second kinds). Singularities require

special quadrature rules, but they are also sometimes blessings in disguise, since they can spoil a kernel's smoothing and make problems well-conditioned.

In §§18.4–18.7 we face up to the issues of inverse problems. §18.4 is an introduction to this large subject.

We should note here that wavelet transforms, already discussed in §13.10, are applicable not only to data compression and signal processing, but can also be used to transform some classes of integral equations into sparse linear problems that allow fast solution. You may wish to review §13.10 as part of reading this chapter.

Some subjects, such as *integro-differential equations*, we must simply declare to be beyond our scope. For a review of methods for integro-differential equations, see Brunner [4].

It should go without saying that this one short chapter can only barely touch on a few of the most basic methods involved in this complicated subject.

CITED REFERENCES AND FURTHER READING:

Delves, L.M., and Mohamed, J.L. 1985, *Computational Methods for Integral Equations* (Cambridge, U.K.: Cambridge University Press). [1]

Linz, P. 1985, *Analytical and Numerical Methods for Volterra Equations* (Philadelphia: S.I.A.M.). [2]

Atkinson, K.E. 1976, *A Survey of Numerical Methods for the Solution of Fredholm Integral Equations of the Second Kind* (Philadelphia: S.I.A.M.). [3]

Brunner, H. 1988, in *Numerical Analysis 1987*, Pitman Research Notes in Mathematics vol. 170, D.F. Griffiths and G.A. Watson, eds. (Harlow, Essex, U.K.: Longman Scientific and Technical), pp. 18–38. [4]

Smithies, F. 1958, *Integral Equations* (Cambridge, U.K.: Cambridge University Press).

Kanwal, R.P. 1971, *Linear Integral Equations* (New York: Academic Press).

Green, C.D. 1969, *Integral Equation Methods* (New York: Barnes & Noble).

18.1 Fredholm Equations of the Second Kind

We desire a numerical solution for $f(t)$ in the equation

$$f(t) = \lambda \int_a^b K(t,s) f(s)\, ds + g(t) \tag{18.1.1}$$

The method we describe, a very basic one, is called the *Nystrom method*. It requires the choice of some approximate *quadrature rule*:

$$\int_a^b y(s)\, ds = \sum_{j=1}^{N} w_j y(s_j) \tag{18.1.2}$$

Here the set $\{w_j\}$ are the weights of the quadrature rule, while the N points $\{s_j\}$ are the abscissas.

What quadrature rule should we use? It is certainly possible to solve integral equations with low-order quadrature rules like the repeated trapezoidal or Simpson's

rules. We will see, however, that the solution method involves $O(N^3)$ operations, and so the most efficient methods tend to use high-order quadrature rules to keep N as small as possible. For smooth, nonsingular problems, nothing beats Gaussian quadrature (e.g., Gauss-Legendre quadrature, §4.5). (For non-smooth or singular kernels, see §18.3.)

Delves and Mohamed [1] investigated methods more complicated than the Nystrom method. For straightforward Fredholm equations of the second kind, they concluded "... the clear winner of this contest has been the Nystrom routine ... with the N-point Gauss-Legendre rule. This routine is extremely simple.... Such results are enough to make a numerical analyst weep."

If we apply the quadrature rule (18.1.2) to equation (18.1.1), we get

$$f(t) = \lambda \sum_{j=1}^{N} w_j K(t, s_j) f(s_j) + g(t) \tag{18.1.3}$$

Evaluate equation (18.1.3) at the quadrature points:

$$f(t_i) = \lambda \sum_{j=1}^{N} w_j K(t_i, s_j) f(s_j) + g(t_i) \tag{18.1.4}$$

Let f_i be the vector $f(t_i)$, g_i the vector $g(t_i)$, K_{ij} the matrix $K(t_i, s_j)$, and define

$$\tilde{K}_{ij} = K_{ij} w_j \tag{18.1.5}$$

Then in matrix notation equation (18.1.4) becomes

$$(\mathbf{1} - \lambda \tilde{\mathbf{K}}) \cdot \mathbf{f} = \mathbf{g} \tag{18.1.6}$$

This is a set of N linear algebraic equations in N unknowns that can be solved by standard triangular decomposition techniques (§2.3) — that is where the $O(N^3)$ operations count comes in. The solution is usually well-conditioned, unless λ is very close to an eigenvalue.

Having obtained the solution at the quadrature points $\{t_i\}$, how do you get the solution at some other point t? You do *not* simply use polynomial interpolation. This destroys all the accuracy you have worked so hard to achieve. Nystrom's key observation was that you should use equation (18.1.3) as an interpolatory formula, maintaining the accuracy of the solution.

We here give two subroutines for use with linear Fredholm equations of the second kind. The routine `fred2` sets up equation (18.1.6) and then solves it by LU decomposition with calls to the routines `ludcmp` and `lubksb`. The Gauss-Legendre quadrature is implemented by first getting the weights and abscissas with a call to `gauleg`. Routine `fred2` requires that you provide an external function that returns $g(t)$ and another that returns λK_{ij}. It then returns the solution f at the quadrature points. It also returns the quadrature points and weights. These are used by the second routine `fredin` to carry out the Nystrom interpolation of equation (18.1.3) and return the value of f at any point in the interval $[a, b]$.

```
      SUBROUTINE fred2(n,a,b,t,f,w,g,ak)
      INTEGER n,NMAX
      REAL a,b,f(n),t(n),w(n),g,ak
      EXTERNAL ak,g
      PARAMETER (NMAX=200)
C     USES ak,g,gauleg,lubksb,ludcmp
        Solves a linear Fredholm equation of the second kind. On input, a and b are the limits of
        integration, and n is the number of points to use in the Gaussian quadrature. g and ak
        are user-supplied external functions that respectively return g(t) and λK(t,s). The routine
        returns arrays t(1:n) and f(1:n) containing the abscissas t_i of the Gaussian quadrature
        and the solution f at these abscissas. Also returned is the array w(1:n) of Gaussian weights
        for use with the Nystrom interpolation routine fredin.
      INTEGER i,j,indx(NMAX)
      REAL d,omk(NMAX,NMAX)
      if(n.gt.NMAX) pause 'increase NMAX in fred2'
      call gauleg(a,b,t,w,n)              Replace gauleg with another routine if not using
      do 12 i=1,n                             Gauss-Legendre quadrature.
        do 11 j=1,n                       Form 1 − λK.
          if(i.eq.j)then
            omk(i,j)=1.
          else
            omk(i,j)=0.
          endif
          omk(i,j)=omk(i,j)-ak(t(i),t(j))*w(j)
        enddo 11
        f(i)=g(t(i))
      enddo 12
      call ludcmp(omk,n,NMAX,indx,d)      Solve linear equations.
      call lubksb(omk,n,NMAX,indx,f)
      return
      END


      FUNCTION fredin(x,n,a,b,t,f,w,g,ak)
      INTEGER n
      REAL fredin,a,b,x,f(n),t(n),w(n),g,ak
      EXTERNAL ak,g
C     USES ak,g
        Given arrays t(1:n) and w(1:n) containing the abscissas and weights of the Gaussian
        quadrature, and given the solution array f(1:n) from fred2, this function returns the
        value of f at x using the Nystrom interpolation formula. On input, a and b are the limits
        of integration, and n is the number of points used in the Gaussian quadrature. g and ak
        are user-supplied external functions that respectively return g(t) and λK(t,s).
      INTEGER i
      REAL sum
      sum=0.
      do 11 i=1,n
        sum=sum+ak(x,t(i))*w(i)*f(i)
      enddo 11
      fredin=g(x)+sum
      return
      END
```

One disadvantage of a method based on Gaussian quadrature is that there is no simple way to obtain an estimate of the error in the result. The best practical method is to increase N by 50%, say, and treat the difference between the two estimates as a conservative estimate of the error in the result obtained with the larger value of N.

Turn now to solutions of the homogeneous equation. If we set $\lambda = 1/\sigma$ and $\mathbf{g} = 0$, then equation (18.1.6) becomes a standard eigenvalue equation

$$\tilde{\mathbf{K}} \cdot \mathbf{f} = \sigma \mathbf{f} \tag{18.1.7}$$

which we can solve with any convenient matrix eigenvalue routine (see Chapter 11). Note that if our original problem had a symmetric kernel, then the matrix $\mathbf{K}$ is symmetric. However, since the weights w_j are not equal for most quadrature rules, the matrix $\tilde{\mathbf{K}}$ (equation 18.1.5) is not symmetric. The matrix eigenvalue problem is much easier for symmetric matrices, and so we should restore the symmetry if possible. Provided the weights are positive (which they are for Gaussian quadrature), we can define the diagonal matrix $\mathbf{D} = \text{diag}(w_j)$ and its square root, $\mathbf{D}^{1/2} = \text{diag}(\sqrt{w_j})$. Then equation (18.1.7) becomes

$$\mathbf{K} \cdot \mathbf{D} \cdot \mathbf{f} = \sigma \mathbf{f}$$

Multiplying by $\mathbf{D}^{1/2}$, we get

$$\left(\mathbf{D}^{1/2} \cdot \mathbf{K} \cdot \mathbf{D}^{1/2}\right) \cdot \mathbf{h} = \sigma \mathbf{h} \tag{18.1.8}$$

where $\mathbf{h} = \mathbf{D}^{1/2} \cdot \mathbf{f}$. Equation (18.1.8) is now in the form of a symmetric eigenvalue problem.

Solution of equations (18.1.7) or (18.1.8) will in general give N eigenvalues, where N is the number of quadrature points used. For square-integrable kernels, these will provide good approximations to the lowest N eigenvalues of the integral equation. Kernels of *finite rank* (also called *degenerate* or *separable* kernels) have only a finite number of nonzero eigenvalues (possibly none). You can diagnose this situation by a cluster of eigenvalues σ that are zero to machine precision. The number of nonzero eigenvalues will stay constant as you increase N to improve their accuracy. Some care is required here: A nondegenerate kernel can have an infinite number of eigenvalues that have an accumulation point at $\sigma = 0$. You distinguish the two cases by the behavior of the solution as you increase N. If you suspect a degenerate kernel, you will usually be able to solve the problem by analytic techniques described in all the textbooks.

CITED REFERENCES AND FURTHER READING:

Delves, L.M., and Mohamed, J.L. 1985, *Computational Methods for Integral Equations* (Cambridge, U.K.: Cambridge University Press). [1]

Atkinson, K.E. 1976, *A Survey of Numerical Methods for the Solution of Fredholm Integral Equations of the Second Kind* (Philadelphia: S.I.A.M.).

18.2 Volterra Equations

Let us now turn to Volterra equations, of which our prototype is the Volterra equation of the second kind,

$$f(t) = \int_a^t K(t,s)f(s)\,ds + g(t) \tag{18.2.1}$$

Most algorithms for Volterra equations march out from $t = a$, building up the solution as they go. In this sense they resemble not only forward substitution (as discussed in §18.0), but also initial-value problems for ordinary differential equations. In fact, many algorithms for ODEs have counterparts for Volterra equations.

The simplest way to proceed is to solve the equation on a mesh with uniform spacing:

$$t_i = a + ih, \quad i = 0, 1, \ldots, N, \qquad h \equiv \frac{b-a}{N} \tag{18.2.2}$$

To do so, we must choose a quadrature rule. For a uniform mesh, the simplest scheme is the trapezoidal rule, equation (4.1.11):

$$\int_a^{t_i} K(t_i,s)f(s)\,ds = h\left(\tfrac{1}{2}K_{i0}f_0 + \sum_{j=1}^{i-1} K_{ij}f_j + \tfrac{1}{2}K_{ii}f_i\right) \tag{18.2.3}$$

Thus the trapezoidal method for equation (18.2.1) is:

$$\begin{aligned} f_0 &= g_0 \\ (1 - \tfrac{1}{2}hK_{ii})f_i &= h\left(\tfrac{1}{2}K_{i0}f_0 + \sum_{j=1}^{i-1} K_{ij}f_j\right) + g_i, \qquad i = 1, \ldots, N \end{aligned} \tag{18.2.4}$$

(For a Volterra equation of the first kind, the leading 1 on the left would be absent, and g would have opposite sign, with corresponding straightforward changes in the rest of the discussion.)

Equation (18.2.4) is an explicit prescription that gives the solution in $O(N^2)$ operations. Unlike Fredholm equations, it is not necessary to solve a system of linear equations. Volterra equations thus usually involve less work than the corresponding Fredholm equations which, as we have seen, do involve the inversion of, sometimes large, linear systems.

The efficiency of solving Volterra equations is somewhat counterbalanced by the fact that *systems* of these equations occur more frequently in practice. If we interpret equation (18.2.1) as a *vector* equation for the vector of m functions $f(t)$, then the kernel $K(t,s)$ is an $m \times m$ matrix. Equation (18.2.4) must now also be understood as a vector equation. For each i, we have to solve the $m \times m$ set of linear algebraic equations by Gaussian elimination.

The routine `voltra` below implements this algorithm. You must supply an external function that returns the kth function of the vector $g(t)$ at the point t, and another that returns the (k,l) element of the matrix $K(t,s)$ at (t,s). The routine `voltra` then returns the vector $f(t)$ at the regularly spaced points t_i.

```
      SUBROUTINE voltra(n,m,t0,h,t,f,g,ak)
      INTEGER m,n,MMAX
      REAL h,t0,f(m,n),t(n),g,ak
      EXTERNAL ak,g
      PARAMETER (MMAX=5)
C     USES ak,g,lubksb,ludcmp
```

Solves a set of m linear Volterra equations of the second kind using the extended trapezoidal rule. On input, t0 is the starting point of the integration and n-1 is the number of steps of size h to be taken. g(k,t) is a user-supplied external function that returns $g_k(t)$, while ak(k,l,t,s) is another user-supplied external function that returns the (k,l) element of the matrix $K(t,s)$. The solution is returned in f(1:m,1:n), with the corresponding abscissas in t(1:n).

```
      INTEGER i,j,k,l,indx(MMAX)
      REAL d,sum,a(MMAX,MMAX),b(MMAX)
      t(1)=t0
      do 11 k=1,m                              Initialize.
        f(k,1)=g(k,t(1))
      enddo 11
      do 16 i=2,n                              Take a step h.
        t(i)=t(i-1)+h
        do 14 k=1,m
          sum=g(k,t(i))                        Accumulate right-hand side of linear equations in
          do 13 l=1,m                            sum.
            sum=sum+0.5*h*ak(k,l,t(i),t(1))*f(l,1)
            do 12 j=2,i-1
              sum=sum+h*ak(k,l,t(i),t(j))*f(l,j)
            enddo 12
            if(k.eq.l)then                     Left-hand side goes in matrix a.
              a(k,l)=1.
            else
              a(k,l)=0.
            endif
            a(k,l)=a(k,l)-0.5*h*ak(k,l,t(i),t(i))
          enddo 13
          b(k)=sum
        enddo 14
        call ludcmp(a,m,MMAX,indx,d)           Solve linear equations.
        call lubksb(a,m,MMAX,indx,b)
        do 15 k=1,m
          f(k,i)=b(k)
        enddo 15
      enddo 16
      return
      END
```

For nonlinear Volterra equations, equation (18.2.4) holds with the product $K_{ii}f_i$ replaced by $K_{ii}(f_i)$, and similarly for the other two products of K's and f's. Thus for each i we solve a nonlinear equation for f_i with a known right-hand side. Newton's method (§9.4 or §9.6) with an initial guess of f_{i-1} usually works very well provided the stepsize is not too big.

Higher-order methods for solving Volterra equations are, in our opinion, not as important as for Fredholm equations, since Volterra equations are relatively easy to solve. However, there is an extensive literature on the subject. Several difficulties arise. First, any method that achieves higher order by operating on several quadrature points simultaneously will need a special method to get started, when values at the first few points are not yet known.

Second, stable quadrature rules can give rise to unexpected instabilities in integral equations. For example, suppose we try to replace the trapezoidal rule in

the algorithm above with Simpson's rule. Simpson's rule naturally integrates over an interval $2h$, so we easily get the function values at the even mesh points. For the odd mesh points, we could try appending one panel of trapezoidal rule. But to which end of the integration should we append it? We could do one step of trapezoidal rule followed by all Simpson's rule, or Simpson's rule with one step of trapezoidal rule at the end. Surprisingly, the former scheme is unstable, while the latter is fine!

A simple approach that can be used with the trapezoidal method given above is Richardson extrapolation: Compute the solution with stepsize h and $h/2$. Then, assuming the error scales with h^2, compute

$$f_{\rm E} = \frac{4f(h/2) - f(h)}{3} \tag{18.2.5}$$

This procedure can be repeated as with Romberg integration.

The general consensus is that the best of the higher order methods is the *block-by-block method* (see [1]). Another important topic is the use of variable stepsize methods, which are much more efficient if there are sharp features in K or f. Variable stepsize methods are quite a bit more complicated than their counterparts for differential equations; we refer you to the literature [1,2] for a discussion.

You should also be on the lookout for singularities in the integrand. If you find them, then look to §18.3 for additional ideas.

CITED REFERENCES AND FURTHER READING:

Linz, P. 1985, *Analytical and Numerical Methods for Volterra Equations* (Philadelphia: S.I.A.M.). [1]

Delves, L.M., and Mohamed, J.L. 1985, *Computational Methods for Integral Equations* (Cambridge, U.K.: Cambridge University Press). [2]

18.3 Integral Equations with Singular Kernels

Many integral equations have singularities in either the kernel or the solution or both. A simple quadrature method will show poor convergence with N if such singularities are ignored. There is sometimes art in how singularities are best handled.

We start with a few straightforward suggestions:

1. Integrable singularities can often be removed by a change of variable. For example, the singular behavior $K(t,s) \sim s^{1/2}$ or $s^{-1/2}$ near $s = 0$ can be removed by the transformation $z = s^{1/2}$. Note that we are assuming that the singular behavior is confined to K, whereas the quadrature actually involves the product $K(t,s)f(s)$, and it is this product that must be "fixed." Ideally, you must deduce the singular nature of the product before you try a numerical solution, and take the appropriate action. Commonly, however, a singular kernel does *not* produce a singular solution $f(t)$. (The highly singular kernel $K(t,s) = \delta(t-s)$ is simply the identity operator, for example.)

2. If $K(t,s)$ can be factored as $w(s)\overline{K}(t,s)$, where $w(s)$ is singular and $\overline{K}(t,s)$ is smooth, then a Gaussian quadrature based on $w(s)$ as a weight function will work well. Even if the factorization is only approximate, the convergence is often improved dramatically. All you have to do is replace `gauleg` in the routine `fred2` by another quadrature routine. Section 4.5 explained how to construct such quadratures; or you can find tabulated abscissas and weights in the standard references [1,2]. You must of course supply $\overline{K}$ instead of K.

This method is a special case of the *product Nystrom method* [3,4], where one factors out a singular term $p(t,s)$ depending on both t and s from K and constructs suitable weights for its Gaussian quadrature. The calculations in the general case are quite cumbersome, because the weights depend on the chosen $\{t_i\}$ as well as the form of $p(t,s)$.

We prefer to implement the product Nystrom method on a uniform grid, with a quadrature scheme that generalizes the extended Simpson's 3/8 rule (equation 4.1.5) to arbitrary weight functions. We discuss this in the subsections below.

3. Special quadrature formulas are also useful when the kernel is not strictly singular, but is "almost" so. One example is when the kernel is concentrated near $t = s$ on a scale much smaller than the scale on which the solution $f(t)$ varies. In that case, a quadrature formula can be based on locally approximating $f(s)$ by a polynomial or spline, while calculating the first few *moments* of the kernel $K(t,s)$ at the tabulation points t_i. In such a scheme the narrow width of the kernel becomes an asset, rather than a liability: The quadrature becomes exact as the width of the kernel goes to zero.

4. An infinite range of integration is also a form of singularity. Truncating the range at a large finite value should be used only as a last resort. If the kernel goes rapidly to zero, then a Gauss-Laguerre [$w \sim \exp(-\alpha s)$] or Gauss-Hermite [$w \sim \exp(-s^2)$] quadrature should work well. Long-tailed functions often succumb to the transformation

$$s = \frac{2\alpha}{z+1} - \alpha \tag{18.3.1}$$

which maps $0 < s < \infty$ to $1 > z > -1$ so that Gauss-Legendre integration can be used. Here $\alpha > 0$ is a constant that you adjust to improve the convergence.

5. A common situation in practice is that $K(t,s)$ is singular along the diagonal line $t = s$. Here the Nystrom method fails completely because the kernel gets evaluated at (t_i, s_i). *Subtraction of the singularity* is one possible cure:

$$\begin{aligned}\int_a^b K(t,s)f(s)\,ds &= \int_a^b K(t,s)[f(s)-f(t)]\,ds + \int_a^b K(t,s)f(t)\,ds \\ &= \int_a^b K(t,s)[f(s)-f(t)]\,ds + r(t)f(t)\end{aligned} \tag{18.3.2}$$

where $r(t) = \int_a^b K(t,s)\,ds$ is computed analytically or numerically. If the first term on the right-hand side is now regular, we can use the Nystrom method. Instead of equation (18.1.4), we get

$$f_i = \lambda \sum_{\substack{j=1 \\ j\neq i}}^{N} w_j K_{ij}[f_j - f_i] + \lambda r_i f_i + g_i \tag{18.3.3}$$

Sometimes the subtraction process must be repeated before the kernel is completely regularized. See [3] for details. (And read on for a different, we think better, way to handle diagonal singularities.)

Quadrature on a Uniform Mesh with Arbitrary Weight

It is possible in general to find n-point linear quadrature rules that approximate the integral of a function $f(x)$, times an arbitrary weight function $w(x)$, over an arbitrary range of integration (a,b), as the sum of weights times n evenly spaced values of the function $f(x)$, say at $x = kh, (k+1)h, \ldots, (k+n-1)h$. The general scheme for deriving such quadrature rules is to write down the n linear equations that must be satisfied if the quadrature rule is to be exact for the n functions $f(x) = \text{const}, x, x^2, \ldots, x^{n-1}$, and then solve these for the coefficients. This can be done analytically, once and for all, if the moments of the weight function over the same range of integration,

$$W_n \equiv \frac{1}{h^n}\int_a^b x^n w(x)dx \tag{18.3.4}$$

are assumed to be known. Here the prefactor h^{-n} is chosen to make W_n scale as h if (as in the usual case) $b - a$ is proportional to h.

Carrying out this prescription for the four-point case gives the result

$$\begin{aligned}\int_a^b w(x)f(x)dx = \\ &\frac{1}{6}f(kh)\Big[(k+1)(k+2)(k+3)W_0 - (3k^2+12k+11)W_1 + 3(k+2)W_2 - W_3\Big] \\ &+\frac{1}{2}f([k+1]h)\Big[-k(k+2)(k+3)W_0 + (3k^2+10k+6)W_1 - (3k+5)W_2 + W_3\Big] \\ &+\frac{1}{2}f([k+2]h)\Big[k(k+1)(k+3)W_0 - (3k^2+8k+3)W_1 + (3k+4)W_2 - W_3\Big] \\ &+\frac{1}{6}f([k+3]h)\Big[-k(k+1)(k+2)W_0 + (3k^2+6k+2)W_1 - 3(k+1)W_2 + W_3\Big]\end{aligned} \tag{18.3.5}$$

While the terms in brackets superficially appear to scale as k^2, there is typically cancellation at both $O(k^2)$ and $O(k)$.

Equation (18.3.5) can be specialized to various choices of (a, b). The obvious choice is $a = kh$, $b = (k+3)h$, in which case we get a four-point quadrature rule that generalizes Simpson's 3/8 rule (equation 4.1.5). In fact, we can recover this special case by setting $w(x) = 1$, in which case (18.3.4) becomes

$$W_n = \frac{h}{n+1}[(k+3)^{n+1} - k^{n+1}] \tag{18.3.6}$$

The four terms in square brackets equation (18.3.5) each become independent of k, and (18.3.5) in fact reduces to

$$\int_{kh}^{(k+3)h} f(x)dx = \frac{3h}{8}f(kh) + \frac{9h}{8}f([k+1]h) + \frac{9h}{8}f([k+2]h) + \frac{3h}{8}f([k+3]h) \tag{18.3.7}$$

Back to the case of general $w(x)$, some other choices for a and b are also useful. For example, we may want to choose (a, b) to be $([k+1]h, [k+3]h)$ or $([k+2]h, [k+3]h)$, allowing us to finish off an extended rule whose number of intervals is not a multiple of three, without loss of accuracy: The integral will be estimated using the four values $f(kh), \ldots, f([k+3]h)$. Even more useful is to choose (a, b) to be $([k+1]h, [k+2]h)$, thus using four points to integrate a centered single interval. These weights, when sewed together into an extended formula, give quadrature schemes that have smooth coefficients, i.e., without the Simpson-like 2, 4, 2, 4, 2 alternation. (In fact, this was the technique that we used to derive equation 4.1.14, which you may now wish to reexamine.)

All these rules are of the same order as the extended Simpson's rule, that is, exact for $f(x)$ a cubic polynomial. Rules of lower order, if desired, are similarly obtained. The three point formula is

$$\begin{aligned}\int_a^b w(x)f(x)dx = &\frac{1}{2}f(kh)\Big[(k+1)(k+2)W_0 - (2k+3)W_1 + W_2\Big] \\ &+ f([k+1]h)\Big[-k(k+2)W_0 + 2(k+1)W_1 - W_2\Big] \\ &+ \frac{1}{2}f([k+2]h)\Big[k(k+1)W_0 - (2k+1)W_1 + W_2\Big]\end{aligned} \tag{18.3.8}$$

Here the simple special case is to take, $w(x) = 1$, so that

$$W_n = \frac{h}{n+1}[(k+2)^{n+1} - k^{n+1}] \tag{18.3.9}$$

Then equation (18.3.8) becomes Simpson's rule,

$$\int_{kh}^{(k+2)h} f(x)dx = \frac{h}{3}f(kh) + \frac{4h}{3}f([k+1]h) + \frac{h}{3}f([k+2]h) \tag{18.3.10}$$

For nonconstant weight functions $w(x)$, however, equation (18.3.8) gives rules of one order less than Simpson, since they do not benefit from the extra symmetry of the constant case.

The two point formula is simply

$$\int_{kh}^{(k+1)h} w(x)f(x)dx = f(kh)[(k+1)W_0 - W_1] + f([k+1]h)[-kW_0 + W_1] \quad (18.3.11)$$

Here is a routine **wwghts** that uses the above formulas to return an extended N-point quadrature rule for the interval $(a,b) = (0,[N-1]h)$. Input to **wwghts** is a user-supplied routine, **kermom**, that is called to get the first four *indefinite-integral* moments of $w(x)$, namely

$$F_m(y) \equiv \int^y s^m w(s)ds \qquad m = 0,1,2,3 \quad (18.3.12)$$

(The lower limit is arbitrary and can be chosen for convenience.) Cautionary note: When called with $N < 4$, **wwghts** returns a rule of lower order than Simpson; you should structure your problem to avoid this.

```
      SUBROUTINE wwghts(wghts,n,h,kermom)
      INTEGER n
      REAL wghts(n),h
      EXTERNAL kermom
C     USES kermom
        Constructs in wghts(1:n) weights for the n-point equal-interval quadrature from 0 to
        (n-1)h of a function f(x) times an arbitrary (possibly singular) weight function w(x) whose
        indefinite-integral moments F_n(y) are provided by the user-supplied subroutine kermom.
      INTEGER j,k
      DOUBLE PRECISION wold(4),wnew(4),w(4),hh,hi,c,fac,a,b
      hh=h                                  Double precision on internal calculations even though
      hi=1.d0/hh                              the interface is in single precision.
      do 11 j=1,n                           Zero all the weights so we can sum into them.
        wghts(j)=0.
      enddo 11
      call kermom(wold,0.d0,4)              Evaluate indefinite integrals at lower end.
      if (n.ge.4) then                      Use highest available order.
        b=0.d0                              For another problem, you might change this lower
        do 14 j=1,n-3                         limit.
          c=j-1                             This is called k in equation (18.3.5).
          a=b                               Set upper and lower limits for this step.
          b=a+hh
          if (j.eq.n-3) b=(n-1)*hh          Last interval: go all the way to end.
          call kermom(wnew,b,4)
          fac=1.d0
          do 12 k=1,4                       Equation (18.3.4).
            w(k)=(wnew(k)-wold(k))*fac
            fac=fac*hi
          enddo 12
          wghts(j)=wghts(j)+                Equation (18.3.5).
     *         ((c+1.d0)*(c+2.d0)*(c+3.d0)*w(1)
     *         -(11.d0+c*(12.d0+c*3.d0))*w(2)
     *         +3.d0*(c+2.d0)*w(3)-w(4))/6.d0
          wghts(j+1)=wghts(j+1)+
     *         (-c*(c+2.d0)*(c+3.d0)*w(1)
     *         +(6.d0+c*(10.d0+c*3.d0))*w(2)
     *         -(3.d0*c+5.d0)*w(3)+w(4))*.5d0
          wghts(j+2)=wghts(j+2)+
     *         (c*(c+1.d0)*(c+3.d0)*w(1)
     *         -(3.d0+c*(8.d0+c*3.d0))*w(2)
     *         +(3.d0*c+4.d0)*w(3)-w(4))*.5d0
          wghts(j+3)=wghts(j+3)+
     *         (-c*(c+1.d0)*(c+2.d0)*w(1)
     *         +(2.d0+c*(6.d0+c*3.d0))*w(2)
     *         -3.d0*(c+1.d0)*w(3)+w(4))/6.d0
          do 13 k=1,4                       Reset lower limits for moments.
```

```
            wold(k)=wnew(k)
        enddo 13
    enddo 14
else if (n.eq.3) then                    Lower-order cases; not recommended.
    call kermom(wnew,hh+hh,3)
    w(1)=wnew(1)-wold(1)
    w(2)=hi*(wnew(2)-wold(2))
    w(3)=hi**2*(wnew(3)-wold(3))
    wghts(1)=w(1)-1.5d0*w(2)+0.5d0*w(3)
    wghts(2)=2.d0*w(2)-w(3)
    wghts(3)=0.5d0*(w(3)-w(2))
else if (n.eq.2) then
    call kermom(wnew,hh,2)
    wghts(2)=hi*(wnew(2)-wold(2))
    wghts(1)=wnew(1)-wold(1)-wghts(2)
endif
END
```

We will now give an example of how to apply `wwghts` to a singular integral equation.

Worked Example: A Diagonally Singular Kernel

As a particular example, consider the integral equation

$$f(x) + \int_0^\pi K(x,y) f(y) dy = \sin x \tag{18.3.13}$$

with the (arbitrarily chosen) nasty kernel

$$K(x,y) = \cos x \cos y \times \begin{cases} \ln(x-y) & y < x \\ \sqrt{y-x} & y \geq x \end{cases} \tag{18.3.14}$$

which has a logarithmic singularity on the left of the diagonal, combined with a square-root discontinuity on the right.

The first step is to do (analytically, in this case) the required moment integrals over the singular part of the kernel, equation (18.3.12). Since these integrals are done at a fixed value of x, we can use x as the lower limit. For any specified value of y, the required indefinite integral is then either

$$F_m(y;x) = \int_x^y s^m (s-x)^{1/2} ds = \int_0^{y-x} (x+t)^m t^{1/2} dt \qquad \text{if } y > x \tag{18.3.15}$$

or

$$F_m(y;x) = \int_x^y s^m \ln(x-s) ds = \int_0^{x-y} (x-t)^m \ln t \, dt \qquad \text{if } y < x \tag{18.3.16}$$

(where a change of variable has been made in the second equality in each case). Doing these integrals analytically (actually, we used a symbolic integration package!), we package the resulting formulas in the following routine. Note that `w(j+1)` returns $F_j(y;x)$.

```
SUBROUTINE kermom(w,y,m)
    Returns in w(1:m) the first m indefinite-integral moments of one row of the singular part
    of the kernel. (For this example, m is hard-wired to be 4.) The input variable y labels the
    column, while x (in COMMON) is the row.
INTEGER m
DOUBLE PRECISION w(m),y,x,d,df,clog,x2,x3,x4
COMMON /momcom/ x
  We can take x as the lower limit of integration. Thus, we return the moment integrals either
  purely to the left or purely to the right of the diagonal.
if (y.ge.x) then
    d=y-x
    df=2.d0*sqrt(d)*d
    w(1)=df/3.d0
```

```
        w(2)=df*(x/3.d0+d/5.d0)
        w(3)=df*((x/3.d0 + 0.4d0*d)*x + d**2/7.d0)
        w(4)=df*(((x/3.d0 + 0.6d0*d)*x + 3.d0*d**2/7.d0)*x
     *       + d**3/9.d0)
      else
        x2=x**2
        x3=x2*x
        x4=x2*x2
        d=x-y
        clog=log(d)
        w(1)=d*(clog-1.d0)
        w(2)=-0.25d0*(3.d0*x+y-2.d0*clog*(x+y))*d
        w(3)=(-11.d0*x3+y*(6.d0*x2+y*(3.d0*x+2.d0*y))
     *       +6.d0*clog*(x3-y**3))/18.d0
        w(4)=(-25.d0*x4+y*(12.d0*x3+y*(6.d0*x2+y*
     *       (4.d0*x+3.d0*y)))+12.d0*clog*(x4-y**4))/48.d0
      endif
      return
      END
```

Next, we write a routine that constructs the quadrature matrix.

```
      SUBROUTINE quadmx(a,n,np)
      INTEGER n,np,NMAX
      REAL a(np,np),PI
      DOUBLE PRECISION xx
      PARAMETER (PI=3.14159265,NMAX=257)
      COMMON /momcom/ xx
      EXTERNAL kermom
C     USES wwghts,kermom
```

Constructs in `a(1:n,1:n)` the quadrature matrix for an example Fredholm equation of the second kind. The nonsingular part of the kernel is computed within this routine, while the quadrature weights which integrate the singular part of the kernel are obtained via calls to `wwghts`. An external routine `kermom`, which supplies indefinite-integral moments of the singular part of the kernel, is passed to `wwghts`.

```
      INTEGER j,k
      REAL h,wt(NMAX),x,cx,y
      h=PI/(n-1)
      do 12 j=1,n
        x=(j-1)*h
        xx=x                          Put x in COMMON for use by kermom.
        call wwghts(wt,n,h,kermom)
        cx=cos(x)                     Part of nonsingular kernel.
        do 11 k=1,n
          y=(k-1)*h
          a(j,k)=wt(k)*cx*cos(y)      Put together all the pieces of the kernel.
        enddo 11
        a(j,j)=a(j,j)+1.              Since equation of the second kind, there is diagonal
      enddo 12                          piece independent of h.
      return
      END
```

Finally, we solve the linear system for any particular right-hand side, here sin x.

```
      PROGRAM fredex
      INTEGER NMAX
      REAL PI
      PARAMETER (NMAX=100,PI=3.14159265)
      INTEGER indx(NMAX),j,n
      REAL a(NMAX,NMAX),g(NMAX),x,d
C     USES quadmx,ludcmp,lubksb
```

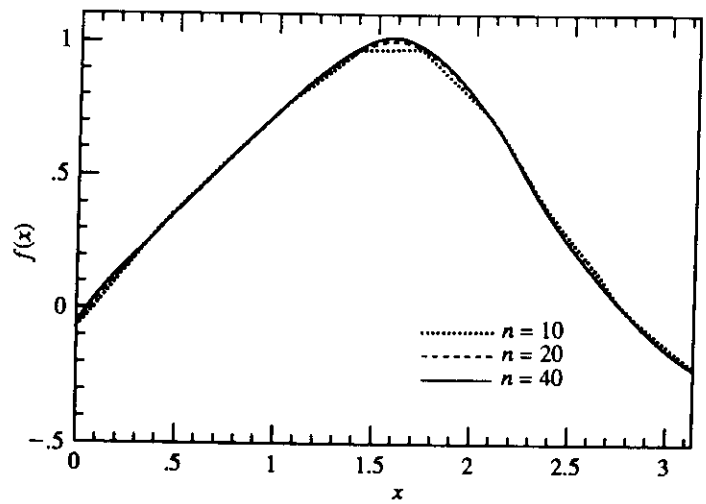


Figure 18.3.1. Solution of the example integral equation (18.3.14) with grid sizes $N = 10$, 20, and 40. The tabulated solution values have been connected by straight lines; in practice one would interpolate a small N solution more smoothly.

This sample program shows how to solve a Fredholm equation of the second kind using the product Nystrom method and a quadrature rule especially constructed for a particular, singular, kernel.

```
n=40                                    Here the size of the grid is specified.
call quadmx(a,n,NMAX)                   Make the quadrature matrix; all the action is here.
call ludcmp(a,n,NMAX,indx,d)            Decompose the matrix.
do 11 j=1,n                             Construct the right hand side, here sin x.
    x=(j-1)*PI/(n-1)
    g(j)=sin(x)
enddo 11
call lubksb(a,n,NMAX,indx,g)            Backsubstitute.
do 12 j=1,n                             Write out the solution.
    x=(j-1)*PI/(n-1)
    write (*,*) j,x,g(j)
enddo 12
write (*,*) 'normal completion'
END
```

With $N = 40$, this program gives accuracy at about the 10^{-5} level. The accuracy increases as N^4 (as it should for our Simpson-order quadrature scheme) *despite* the highly singular kernel. Figure 18.3.1 shows the solution obtained, also plotting the solution for smaller values of N, which are themselves seen to be remarkably faithful. Notice that the solution is smooth, even though the kernel is singular, a common occurrence.

CITED REFERENCES AND FURTHER READING:

Abramowitz, M., and Stegun, I.A. 1964, *Handbook of Mathematical Functions*, Applied Mathematics Series, vol. 55 (Washington: National Bureau of Standards; reprinted 1968 by Dover Publications, New York). [1]

Stroud, A.H., and Secrest, D. 1966, *Gaussian Quadrature Formulas* (Englewood Cliffs, NJ: Prentice-Hall). [2]

Delves, L.M., and Mohamed, J.L. 1985, *Computational Methods for Integral Equations* (Cambridge, U.K.: Cambridge University Press). [3]

Atkinson, K.E. 1976, *A Survey of Numerical Methods for the Solution of Fredholm Integral Equations of the Second Kind* (Philadelphia: S.I.A.M.). [4]

18.4 Inverse Problems and the Use of A Priori Information

Later discussion will be facilitated by some preliminary mention of a couple of mathematical points. Suppose that $\mathbf{u}$ is an "unknown" vector that we plan to determine by some minimization principle. Let $\mathcal{A}[\mathbf{u}] > 0$ and $\mathcal{B}[\mathbf{u}] > 0$ be two positive functionals of $\mathbf{u}$, so that we can try to determine $\mathbf{u}$ by either

$$\text{minimize: } \mathcal{A}[\mathbf{u}] \quad \text{or} \quad \text{minimize: } \mathcal{B}[\mathbf{u}] \tag{18.4.1}$$

(Of course these will generally give different answers for $\mathbf{u}$.) As another possibility, now suppose that we want to minimize $\mathcal{A}[\mathbf{u}]$ subject to the *constraint* that $\mathcal{B}[\mathbf{u}]$ have some particular value, say b. The method of Lagrange multipliers gives the variation

$$\frac{\delta}{\delta \mathbf{u}} \{\mathcal{A}[\mathbf{u}] + \lambda_1(\mathcal{B}[\mathbf{u}] - b)\} = \frac{\delta}{\delta \mathbf{u}} (\mathcal{A}[\mathbf{u}] + \lambda_1 \mathcal{B}[\mathbf{u}]) = 0 \tag{18.4.2}$$

where λ_1 is a Lagrange multiplier. Notice that b is absent in the second equality, since it doesn't depend on $\mathbf{u}$.

Next, suppose that we change our minds and decide to minimize $\mathcal{B}[\mathbf{u}]$ subject to the constraint that $\mathcal{A}[\mathbf{u}]$ have a particular value, a. Instead of equation (18.4.2) we have

$$\frac{\delta}{\delta \mathbf{u}} \{\mathcal{B}[\mathbf{u}] + \lambda_2(\mathcal{A}[\mathbf{u}] - a)\} = \frac{\delta}{\delta \mathbf{u}} (\mathcal{B}[\mathbf{u}] + \lambda_2 \mathcal{A}[\mathbf{u}]) = 0 \tag{18.4.3}$$

with, this time, λ_2 the Lagrange multiplier. Multiplying equation (18.4.3) by the constant $1/\lambda_2$, and identifying $1/\lambda_2$ with λ_1, we see that the actual variations are exactly the same in the two cases. Both cases will yield the same one-parameter family of solutions, say, $\mathbf{u}(\lambda_1)$. As λ_1 varies from 0 to ∞, the solution $\mathbf{u}(\lambda_1)$ varies along a so-called *trade-off curve* between the problem of minimizing $\mathcal{A}$ and the problem of minimizing $\mathcal{B}$. Any solution along this curve can equally well be thought of as either (i) a minimization of $\mathcal{A}$ for some constrained value of $\mathcal{B}$, or (ii) a minimization of $\mathcal{B}$ for some constrained value of $\mathcal{A}$, or (iii) a weighted minimization of the sum $\mathcal{A} + \lambda_1 \mathcal{B}$.

The second preliminary point has to do with *degenerate* minimization principles. In the example above, now suppose that $\mathcal{A}[\mathbf{u}]$ has the particular form

$$\mathcal{A}[\mathbf{u}] = |\mathbf{A} \cdot \mathbf{u} - \mathbf{c}|^2 \tag{18.4.4}$$

for some matrix $\mathbf{A}$ and vector $\mathbf{c}$. If $\mathbf{A}$ has fewer rows than columns, or if $\mathbf{A}$ is square but degenerate (has a nontrivial nullspace, see §2.6, especially Figure 2.6.1), then minimizing $\mathcal{A}[\mathbf{u}]$ will *not* give a unique solution for $\mathbf{u}$. (To see why, review §15.4, and note that for a "design matrix" $\mathbf{A}$ with fewer rows than columns, the matrix $\mathbf{A}^T \cdot \mathbf{A}$ in the normal equations 15.4.10 is degenerate.) *However*, if we add any multiple λ times a nondegenerate quadratic form $\mathcal{B}[\mathbf{u}]$, for example $\mathbf{u} \cdot \mathbf{H} \cdot \mathbf{u}$ with $\mathbf{H}$ a positive definite matrix, then minimization of $\mathcal{A}[\mathbf{u}] + \lambda \mathcal{B}[\mathbf{u}]$ *will* lead to a unique solution for $\mathbf{u}$. (The sum of two quadratic forms is itself a quadratic form, with the second piece guaranteeing nondegeneracy.)

We can combine these two points, for this conclusion: When a quadratic minimization principle is combined with a quadratic constraint, and both are positive, only *one* of the two need be nondegenerate for the overall problem to be well-posed. We are now equipped to face the subject of inverse problems.

The Inverse Problem with Zeroth-Order Regularization

Suppose that $u(x)$ is some unknown or underlying (u stands for both unknown and underlying!) physical process, which we hope to determine by a set of N measurements c_i, $i = 1, 2, \ldots, N$. The relation between $u(x)$ and the c_i's is that each c_i measures a (hopefully distinct) aspect of $u(x)$ through its own linear response kernel r_i, and with its own measurement error n_i. In other words,

$$c_i \equiv s_i + n_i = \int r_i(x)u(x)dx + n_i \tag{18.4.5}$$

(compare this to equations 13.3.1 and 13.3.2). Within the assumption of linearity, this is quite a general formulation. The c_i's might approximate values of $u(x)$ at certain locations x_i, in which case $r_i(x)$ would have the form of a more or less narrow instrumental response centered around $x = x_i$. Or, the c_i's might "live" in an entirely different function space from $u(x)$, measuring different Fourier components of $u(x)$ for example.

The *inverse problem* is, given the c_i's, the $r_i(x)$'s, and perhaps some information about the errors n_i such as their covariance matrix

$$S_{ij} \equiv \text{Covar}[n_i, n_j] \tag{18.4.6}$$

how do we find a good statistical estimator of $u(x)$, call it $\widehat{u}(x)$?

It should be obvious that this is an ill-posed problem. After all, how can we reconstruct a whole function $\widehat{u}(x)$ from only a finite number of discrete values c_i? Yet, whether formally or informally, we do this all the time in science. We routinely measure "enough points" and then "draw a curve through them." In doing so, we are making some assumptions, either about the underlying function $u(x)$, or about the nature of the response functions $r_i(x)$, or both. Our purpose now is to formalize these assumptions, and to extend our abilities to cases where the measurements and underlying function live in quite different function spaces. (How do you "draw a curve" through a scattering of Fourier coefficients?)

We can't really want every point x of the function $\widehat{u}(x)$. We do want some large number M of discrete points x_μ, $\mu = 1, 2, \ldots, M$, where M is sufficiently large, and the x_μ's are sufficiently evenly spaced, that neither $u(x)$ nor $r_i(x)$ varies much between any x_μ and $x_{\mu+1}$. (Here and following we will use Greek letters like μ to denote values in the space of the underlying process, and Roman letters like i to denote values of immediate observables.) For such a dense set of x_μ's, we can replace equation (18.4.5) by a quadrature like

$$c_i = \sum_\mu R_{i\mu}u(x_\mu) + n_i \tag{18.4.7}$$

where the $N \times M$ matrix $\mathbf{R}$ has components

$$R_{i\mu} \equiv r_i(x_\mu)(x_{\mu+1} - x_{\mu-1})/2 \tag{18.4.8}$$

(or any other simple quadrature — it rarely matters which). We will view equations (18.4.5) and (18.4.7) as being equivalent for practical purposes.

How do you solve a set of equations like equation (18.4.7) for the unknown $u(x_\mu)$'s? Here is a bad way, but one that contains the germ of some correct ideas: Form a χ^2 measure of how well a model $\hat{u}(x)$ agrees with the measured data,

$$\begin{aligned}\chi^2 &= \sum_{i=1}^{N}\sum_{j=1}^{N}\left[c_i - \sum_{\mu=1}^{M} R_{i\mu}\hat{u}(x_\mu)\right] S_{ij}^{-1}\left[c_j - \sum_{\mu=1}^{M} R_{j\mu}\hat{u}(x_\mu)\right] \\ &\approx \sum_{i=1}^{N}\left[\frac{c_i - \sum_{\mu=1}^{M} R_{i\mu}\hat{u}(x_\mu)}{\sigma_i}\right]^2\end{aligned} \tag{18.4.9}$$

(compare with equation 15.1.5). Here $\mathbf{S}^{-1}$ is the inverse of the covariance matrix, and the approximate equality holds if you can neglect the off-diagonal covariances, with $\sigma_i \equiv (\text{Covar}[i,i])^{1/2}$.

Now you can use the method of singular value decomposition (SVD) in §15.4 to find the vector $\hat{\mathbf{u}}$ that minimizes equation (18.4.9). Don't try to use the method of normal equations; since M is greater than N they will be singular, as we already discussed. The SVD process will thus surely find a large number of zero singular values, indicative of a highly non-unique solution. Among the infinity of degenerate solutions (most of them badly behaved with arbitrarily large $\hat{u}(x_\mu)$'s) SVD will select the one with smallest $|\hat{\mathbf{u}}|$ in the sense of

$$\sum_\mu [\hat{u}(x_\mu)]^2 \quad \text{a minimum} \tag{18.4.10}$$

(look at Figure 2.6.1). This solution is often called the *principal solution*. It is a limiting case of what is called *zeroth-order regularization*, corresponding to minimizing the sum of the two positive functionals

$$\text{minimize:} \quad \chi^2[\hat{\mathbf{u}}] + \lambda(\hat{\mathbf{u}}\cdot\hat{\mathbf{u}}) \tag{18.4.11}$$

in the limit of small λ. Below, we will learn how to do such minimizations, as well as more general ones, without the *ad hoc* use of SVD.

What happens if we determine $\hat{\mathbf{u}}$ by equation (18.4.11) with a non-infinitesimal value of λ? First, note that if $M \gg N$ (many more unknowns than equations), then $\mathbf{u}$ will often have enough freedom to be able to make χ^2 (equation 18.4.9) quite unrealistically small, if not zero. In the language of §15.1, the number of degrees of freedom $\nu = N - M$, which is approximately the expected value of χ^2 when ν is large, is being driven down to zero (and, not meaningfully, beyond). Yet, we know that for the *true* underlying function $u(x)$, which has no adjustable parameters, the number of degrees of freedom and the expected value of χ^2 should be about $\nu \approx N$.

Increasing λ pulls the solution away from minimizing χ^2 in favor of minimizing $\hat{\mathbf{u}}\cdot\hat{\mathbf{u}}$. From the preliminary discussion above, we can view this as minimizing $\hat{\mathbf{u}}\cdot\hat{\mathbf{u}}$ subject to the *constraint* that χ^2 have some constant nonzero value. A popular choice, in fact, is to find that value of λ which yields $\chi^2 = N$, that is, to get about as much extra regularization as a plausible value of χ^2 dictates. The resulting $\hat{u}(x)$ is called *the solution of the inverse problem with zeroth-order regularization.*

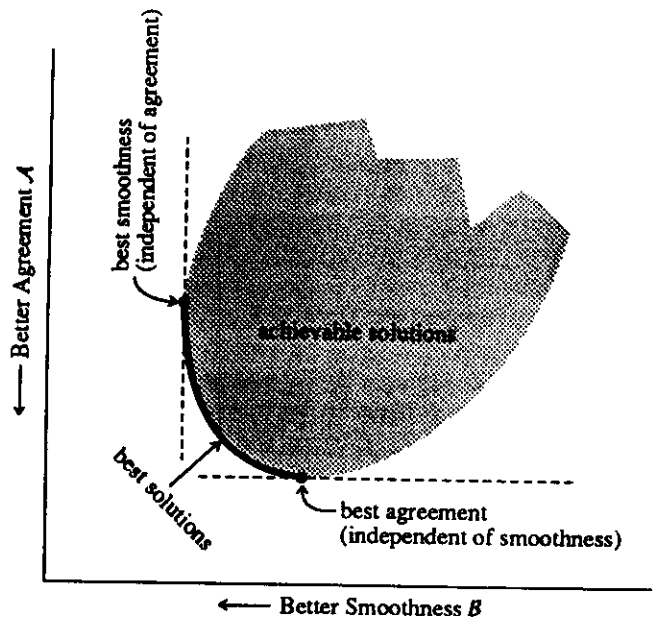


Figure 18.4.1. Almost all inverse problem methods involve a trade-off between two optimizations: agreement between data and solution, or "sharpness" of mapping between true and estimated solution (here denoted $\mathcal{A}$), and smoothness or stability of the solution (here denoted $\mathcal{B}$). Among all possible solutions, shown here schematically as the shaded region, those on the boundary connecting the unconstrained minimum of $\mathcal{A}$ and the unconstrained minimum of $\mathcal{B}$ are the "best" solutions, in the sense that every other solution is dominated by at least one solution on the curve.

The value N is actually a surrogate for any value drawn from a Gaussian distribution with mean N and standard deviation $(2N)^{1/2}$ (the asymptotic χ^2 distribution). One might equally plausibly try two values of λ, one giving $\chi^2 = N + (2N)^{1/2}$, the other $N - (2N)^{1/2}$.

Zeroth-order regularization, though dominated by better methods, demonstrates most of the basic ideas that are used in inverse problem theory. In general, there are two positive functionals, call them $\mathcal{A}$ and $\mathcal{B}$. The first, $\mathcal{A}$, measures something like the agreement of a model to the data (e.g., χ^2), or sometimes a related quantity like the "sharpness" of the mapping between the solution and the underlying function. When $\mathcal{A}$ by itself is minimized, the agreement or sharpness becomes very good (often impossibly good), but the solution becomes unstable, wildly oscillating, or in other ways unrealistic, reflecting that $\mathcal{A}$ alone typically defines a highly degenerate minimization problem.

That is where $\mathcal{B}$ comes in. It measures something like the "smoothness" of the desired solution, or sometimes a related quantity that parametrizes the stability of the solution with respect to variations in the data, or sometimes a quantity reflecting *a priori* judgments about the likelihood of a solution. $\mathcal{B}$ is called the *stabilizing functional* or *regularizing operator*. In any case, minimizing $\mathcal{B}$ by itself is supposed to give a solution that is "smooth" or "stable" or "likely" — and that has nothing at all to do with the measured data.

The single central idea in inverse theory is the prescription

$$\text{minimize:} \quad \mathcal{A} + \lambda \mathcal{B} \tag{18.4.12}$$

for various values of $0 < \lambda < \infty$ along the so-called trade-off curve (see Figure 18.4.1), and then to settle on a "best" value of λ by one or another criterion, ranging from fairly objective (e.g., making $\chi^2 = N$) to entirely subjective. Successful methods, several of which we will now describe, differ as to their choices of $\mathcal{A}$ and $\mathcal{B}$, as to whether the prescription (18.4.12) yields linear or nonlinear equations, as to their recommended method for selecting a final λ, and as to their practicality for computer-intensive two-dimensional problems like image processing.

They also differ as to the philosophical baggage that they (or rather, their proponents) carry. We have thus far avoided the word "Bayesian." (Courts have consistently held that academic license does not extend to shouting "Bayesian" in a crowded lecture hall.) But it is hard, nor have we any wish, to disguise the fact that $\mathcal{B}$ has something to do with *a priori* expectation, or knowledge, of a solution, while $\mathcal{A}$ has something to do with *a posteriori* knowledge. The constant λ adjudicates a delicate compromise between the two. Some inverse methods have acquired a more Bayesian stamp than others, but we think that this is purely an accident of history. An outsider looking only at the equations that are actually solved, and not at the accompanying philosophical justifications, would have a difficult time separating the so-called Bayesian methods from the so-called empirical ones, we think.

The next three sections discuss three different approaches to the problem of inversion, which have had considerable success in different fields. All three fit within the general framework that we have outlined, but they are quite different in detail and in implementation.

CITED REFERENCES AND FURTHER READING:

Craig, I.J.D., and Brown, J.C. 1986, *Inverse Problems in Astronomy* (Bristol, U.K.: Adam Hilger).

Twomey, S. 1977, *Introduction to the Mathematics of Inversion in Remote Sensing and Indirect Measurements* (Amsterdam: Elsevier).

Tikhonov, A.N., and Arsenin, V.Y. 1977, *Solutions of Ill-Posed Problems* (New York: Wiley).

Tikhonov, A.N., and Goncharsky, A.V. (eds.) 1987, *Ill-Posed Problems in the Natural Sciences* (Moscow: MIR).

Parker, R.L. 1977, *Annual Review of Earth and Planetary Science*, vol. 5, pp. 35–64.

Frieden, B.R. 1975, in *Picture Processing and Digital Filtering*, T.S. Huang, ed. (New York: Springer-Verlag).

Tarantola, A. 1987, *Inverse Problem Theory* (Amsterdam: Elsevier).

Baumeister, J. 1987, *Stable Solution of Inverse Problems* (Braunschweig, Germany: Friedr. Vieweg & Sohn) [mathematically oriented].

Titterington, D.M. 1985, *Astronomy and Astrophysics*, vol. 144, pp. 381–387.

Jeffrey, W., and Rosner, R. 1986, *Astrophysical Journal*, vol. 310, pp. 463–472.

18.5 Linear Regularization Methods

What we will call *linear regularization* is also called the *Phillips-Twomey method* [1,2], the *constrained linear inversion method* [3], the *method of regularization* [4], and *Tikhonov-Miller regularization* [5-7]. (It probably has other names also,

since it is so obviously a good idea.) In its simplest form, the method is an immediate generalization of zeroth-order regularization (equation 18.4.11, above). As before, the functional $\mathcal{A}$ is taken to be the χ^2 deviation, equation (18.4.9), but the functional $\mathcal{B}$ is replaced by more sophisticated measures of smoothness that derive from first or higher derivatives.

For example, suppose that your *a priori* belief is that a credible $u(x)$ is not too different from a constant. Then a reasonable functional to minimize is

$$\mathcal{B} \propto \int [\hat{u}'(x)]^2 dx \propto \sum_{\mu=1}^{M-1} [\hat{u}_\mu - \hat{u}_{\mu+1}]^2 \tag{18.5.1}$$

since it is nonnegative and equal to zero only when $\hat{u}(x)$ is constant. Here $\hat{u}_\mu \equiv \hat{u}(x_\mu)$, and the second equality (proportionality) assumes that the x_μ's are uniformly spaced. We can write the second form of $\mathcal{B}$ as

$$\mathcal{B} = |\mathbf{B} \cdot \hat{\mathbf{u}}|^2 = \hat{\mathbf{u}} \cdot (\mathbf{B}^T \cdot \mathbf{B}) \cdot \hat{\mathbf{u}} \equiv \hat{\mathbf{u}} \cdot \mathbf{H} \cdot \hat{\mathbf{u}} \tag{18.5.2}$$

where $\hat{\mathbf{u}}$ is the vector of components $\hat{u}_\mu$, $\mu = 1, \ldots, M$, $\mathbf{B}$ is the $(M-1) \times M$ first difference matrix

$$\mathbf{B} = \begin{pmatrix} -1 & 1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & & & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{pmatrix} \tag{18.5.3}$$

and $\mathbf{H}$ is the $M \times M$ matrix

$$\mathbf{H} = \mathbf{B}^T \cdot \mathbf{B} = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ -1 & 2 & -1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & & & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & \cdots & 0 & 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & \cdots & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{pmatrix} \tag{18.5.4}$$

Note that $\mathbf{B}$ has one fewer row than column. It follows that the symmetric $\mathbf{H}$ is degenerate; it has exactly one zero eigenvalue corresponding to the *value* of a constant function, any one of which makes $\mathcal{B}$ exactly zero.

If, just as in §15.4, we write

$$A_{i\mu} \equiv R_{i\mu}/\sigma_i \qquad b_i \equiv c_i/\sigma_i \tag{18.5.5}$$

then, using equation (18.4.9), the minimization principle (18.4.12) is

$$\text{minimize:} \quad \mathcal{A} + \lambda\mathcal{B} = |\mathbf{A} \cdot \hat{\mathbf{u}} - \mathbf{b}|^2 + \lambda \hat{\mathbf{u}} \cdot \mathbf{H} \cdot \hat{\mathbf{u}} \tag{18.5.6}$$

This can readily be reduced to a linear set of *normal equations*, just as in §15.4: The components $\hat{u}_\mu$ of the solution satisfy the set of M equations in M unknowns,

$$\sum_\rho \left[\left(\sum_i A_{i\mu} A_{i\rho} \right) + \lambda H_{\mu\rho} \right] \hat{u}_\rho = \sum_i A_{i\mu} b_i \qquad \mu = 1, 2, \ldots, M \tag{18.5.7}$$

or, in vector notation,

$$(\mathbf{A}^T \cdot \mathbf{A} + \lambda \mathbf{H}) \cdot \hat{\mathbf{u}} = \mathbf{A}^T \cdot \mathbf{b} \tag{18.5.8}$$

Equations (18.5.7) or (18.5.8) can be solved by the standard techniques of Chapter 2, e.g., LU decomposition. The usual warnings about normal equations being ill-conditioned do not apply, since the whole purpose of the λ term is to cure that same ill-conditioning. Note, however, that the λ term *by itself* is ill-conditioned, since it does not select a preferred constant value. You hope your data can at least do *that*!

Although inversion of the matrix $(\mathbf{A}^T \cdot \mathbf{A} + \lambda \mathbf{H})$ is not generally the best way to solve for $\hat{\mathbf{u}}$, let us digress to write the solution to equation (18.5.8) schematically as

$$\hat{\mathbf{u}} = \left(\frac{1}{\mathbf{A}^T \cdot \mathbf{A} + \lambda \mathbf{H}} \cdot \mathbf{A}^T \cdot \mathbf{A} \right) \mathbf{A}^{-1} \cdot \mathbf{b} \qquad \text{(schematic only!)} \tag{18.5.9}$$

where the identity matrix in the form $\mathbf{A} \cdot \mathbf{A}^{-1}$ has been inserted. This is schematic not only because the matrix inverse is fancifully written as a denominator, but also because, in general, the inverse matrix $\mathbf{A}^{-1}$ does not exist. However, it is illuminating to compare equation (18.5.9) with equation (13.3.6) for optimal or Wiener filtering, or with equation (13.6.6) for general linear prediction. One sees that $\mathbf{A}^T \cdot \mathbf{A}$ plays the role of S^2, the signal power or autocorrelation, while $\lambda \mathbf{H}$ plays the role of N^2, the noise power or autocorrelation. The term in parentheses in equation (18.5.9) is something like an optimal filter, whose effect is to pass the ill-posed inverse $\mathbf{A}^{-1} \cdot \mathbf{b}$ through unmodified when $\mathbf{A}^T \cdot \mathbf{A}$ is sufficiently large, but to suppress it when $\mathbf{A}^T \cdot \mathbf{A}$ is small.

The above choices of $\mathbf{B}$ and $\mathbf{H}$ are only the simplest in an obvious sequence of derivatives. If your *a priori* belief is that a *linear* function is a good approximation to $u(x)$, then minimize

$$\mathcal{B} \propto \int [\hat{u}''(x)]^2 dx \propto \sum_{\mu=1}^{M-2} [-\hat{u}_\mu + 2\hat{u}_{\mu+1} - \hat{u}_{\mu+2}]^2 \tag{18.5.10}$$

implying

$$\mathbf{B} = \begin{pmatrix} -1 & 2 & -1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & & & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & 0 & 0 & -1 & 2 & -1 & 0 \\ 0 & \cdots & 0 & 0 & 0 & 0 & -1 & 2 & -1 \end{pmatrix} \tag{18.5.11}$$

and

$$\mathbf{H} = \mathbf{B}^T \cdot \mathbf{B} = \begin{pmatrix} 1 & -2 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ -2 & 5 & -4 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 1 & -4 & 6 & -4 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & -4 & 6 & -4 & 1 & 0 & \cdots & 0 \\ \vdots & & & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & 1 & -4 & 6 & -4 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 1 & -4 & 6 & -4 & 1 \\ 0 & \cdots & 0 & 0 & 0 & 1 & -4 & 5 & -2 \\ 0 & \cdots & 0 & 0 & 0 & 0 & 1 & -2 & 1 \end{pmatrix} \tag{18.5.12}$$

This **H** has two zero eigenvalues, corresponding to the two undetermined parameters of a linear function.

If your *a priori* belief is that a *quadratic* function is preferable, then minimize

$$\mathcal{B} \propto \int [\hat{u}'''(x)]^2 dx \propto \sum_{\mu=1}^{M-3} [-\hat{u}_\mu + 3\hat{u}_{\mu+1} - 3\hat{u}_{\mu+2} + \hat{u}_{\mu+3}]^2 \tag{18.5.13}$$

with

$$\mathbf{B} = \begin{pmatrix} -1 & 3 & -3 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 3 & -3 & 1 & 0 & 0 & \cdots & 0 \\ \vdots & & & & \ddots & & & & \vdots \\ 0 & \cdots & 0 & 0 & -1 & 3 & -3 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 0 & -1 & 3 & -3 & 1 \end{pmatrix} \tag{18.5.14}$$

and now

$$\mathbf{H} = \begin{pmatrix} 1 & -3 & 3 & -1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ -3 & 10 & -12 & 6 & -1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 3 & -12 & 19 & -15 & 6 & -1 & 0 & 0 & 0 & \cdots & 0 \\ -1 & 6 & -15 & 20 & -15 & 6 & -1 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 6 & -15 & 20 & -15 & 6 & -1 & 0 & \cdots & 0 \\ \vdots & & & & & \ddots & & & & & \vdots \\ 0 & \cdots & 0 & -1 & 6 & -15 & 20 & -15 & 6 & -1 & 0 \\ 0 & \cdots & 0 & 0 & -1 & 6 & -15 & 20 & -15 & 6 & -1 \\ 0 & \cdots & 0 & 0 & 0 & -1 & 6 & -15 & 19 & -12 & 3 \\ 0 & \cdots & 0 & 0 & 0 & 0 & -1 & 6 & -12 & 10 & -3 \\ 0 & \cdots & 0 & 0 & 0 & 0 & 0 & -1 & 3 & -3 & 1 \end{pmatrix} \tag{18.5.15}$$

(We'll leave the calculation of cubics and above to the compulsive reader.)

Notice that you can regularize with "closeness to a differential equation," if you want. Just pick **B** to be the appropriate sum of finite-difference operators (the coefficients can depend on x), and calculate $\mathbf{H} = \mathbf{B}^T \cdot \mathbf{B}$. You don't need to know the values of your boundary conditions, since **B** can have fewer rows than columns, as above; hopefully, your data will determine them. Of course, if you do know some boundary conditions, you can build these into **B** too.

With all the proportionality signs above, you may have lost track of what actual value of λ to try first. A simple trick for at least getting "on the map" is to first try

$$\lambda = \text{Tr}(\mathbf{A}^T \cdot \mathbf{A})/\text{Tr}(\mathbf{H}) \tag{18.5.16}$$

where Tr is the trace of the matrix (sum of diagonal components). This choice will tend to make the two parts of the minimization have comparable weights, and you can adjust from there.

As for what is the "correct" value of λ, an objective criterion, if you know your errors σ_i with reasonable accuracy, is to make χ^2 (that is, $|\mathbf{A} \cdot \hat{\mathbf{u}} - \mathbf{b}|^2$) equal to N, the number of measurements. We remarked above on the twin acceptable choices $N \pm (2N)^{1/2}$. A subjective criterion is to pick any value that you like in the

range $0 < \lambda < \infty$, depending on your relative degree of belief in the *a priori* and *a posteriori* evidence. (Yes, people actually do that. Don't blame us.)

Two-Dimensional Problems and Iterative Methods

Up to now our notation has been indicative of a one-dimensional problem, finding $\hat{u}(x)$ or $\hat{u}_\mu = \hat{u}(x_\mu)$. However, all of the discussion easily generalizes to the problem of estimating a two-dimensional set of unknowns $\hat{u}_{\mu\kappa}$, $\mu = 1, \ldots, M$, $\kappa = 1, \ldots, K$, corresponding, say, to the pixel intensities of a measured image. In this case, equation (18.5.8) is still the one we want to solve.

In image processing, it is usual to have the same number of input pixels in a measured "raw" or "dirty" image as desired "clean" pixels in the processed output image, so the matrices **R** and **A** (equation 18.5.5) are square and of size $MK \times MK$. **A** is typically much too large to represent as a full matrix, but often it is either (i) sparse, with coefficients blurring an underlying pixel (i, j) only into measurements $(i \pm \text{few}, j \pm \text{few})$, or (ii) translationally invariant, so that $A_{(i,j)(\mu,\nu)} = A(i-\mu, j-\nu)$. Both of these situations lead to tractable problems.

In the case of translational invariance, fast Fourier transforms (FFTs) are the obvious method of choice. The general linear relation between underlying function and measured values (18.4.7) now becomes a discrete convolution like equation (13.1.1). If $\mathbf{k}$ denotes a two-dimensional wave-vector, then the two-dimensional FFT takes us back and forth between the transform pairs

$$A(i-\mu, j-\nu) \Longleftrightarrow \widetilde{\mathbf{A}}(\mathbf{k}) \qquad b_{(i,j)} \Longleftrightarrow \widetilde{b}(\mathbf{k}) \qquad \hat{u}_{(i,j)} \Longleftrightarrow \widetilde{u}(\mathbf{k}) \quad (18.5.17)$$

We also need a regularization or smoothing operator **B** and the derived $\mathbf{H} = \mathbf{B}^T \cdot \mathbf{B}$. One popular choice for **B** is the five-point finite-difference approximation of the Laplacian operator, that is, the difference between the value of each point and the average of its four Cartesian neighbors. In Fourier space, this choice implies,

$$\begin{aligned} \widetilde{B}(\mathbf{k}) &\propto \sin^2(\pi k_1/M)\sin^2(\pi k_2/K) \\ \widetilde{H}(\mathbf{k}) &\propto \sin^4(\pi k_1/M)\sin^4(\pi k_2/K) \end{aligned} \qquad (18.5.18)$$

In Fourier space, equation (18.5.7) is merely algebraic, with solution

$$\widetilde{u}(\mathbf{k}) = \frac{\widetilde{A}^*(\mathbf{k})\widetilde{b}(\mathbf{k})}{|\widetilde{A}(\mathbf{k})|^2 + \lambda \widetilde{H}(\mathbf{k})} \qquad (18.5.19)$$

where asterisk denotes complex conjugation. You can make use of the FFT routines for real data in §12.5.

Turn now to the case where **A** is not translationally invariant. Direct solution of (18.5.8) is now hopeless, since the matrix **A** is just too large. We need some kind of iterative scheme.

One way to proceed is to use the full machinery of the conjugate gradient method in §10.6 to find the minimum of $\mathcal{A} + \lambda\mathcal{B}$, equation (18.5.6). Of the various methods in Chapter 10, conjugate gradient is the unique best choice because (i) it does not require storage of a Hessian matrix, which would be infeasible here,

and (ii) it does exploit gradient information, which we can readily compute: The gradient of equation (18.5.6) is

$$\nabla(\mathcal{A} + \lambda\mathcal{B}) = 2[(\mathbf{A}^T \cdot \mathbf{A} + \lambda\mathbf{H}) \cdot \hat{\mathbf{u}} - \mathbf{A}^T \cdot \mathbf{b}] \tag{18.5.20}$$

(cf. 18.5.8). Evaluation of both the function and the gradient should of course take advantage of the sparsity of A, for example via the routines `sprsax` and `sprstx` in §2.7. We will discuss the conjugate gradient technique further in §18.7, in the context of the (nonlinear) maximum entropy method. Some of that discussion can apply here as well.

The conjugate gradient method notwithstanding, application of the unsophisticated steepest descent method (see §10.6) can sometimes produce useful results, particularly when combined with projections onto convex sets (see below). If the solution after k iterations is denoted $\hat{\mathbf{u}}^{(k)}$, then after $k+1$ iterations we have

$$\hat{\mathbf{u}}^{(k+1)} = [\mathbf{1} - \epsilon(\mathbf{A}^T \cdot \mathbf{A} + \lambda\mathbf{H})] \cdot \hat{\mathbf{u}}^{(k)} + \epsilon\mathbf{A}^T \cdot \mathbf{b} \tag{18.5.21}$$

Here ϵ is a parameter that dictates how far to move in the downhill gradient direction. The method converges when ϵ is small enough, in particular satisfying

$$0 < \epsilon < \frac{2}{\text{max eigenvalue}\,(\mathbf{A}^T \cdot \mathbf{A} + \lambda\mathbf{H})} \tag{18.5.22}$$

There exist complicated schemes for finding optimal values or sequences for ϵ, see [7]; or, one can adopt an experimental approach, evaluating (18.5.6) to be sure that downhill steps are in fact being taken.

In those image processing problems where the final measure of success is somewhat subjective (e.g., "how good does the picture look?"), iteration (18.5.21) sometimes produces significantly improved images long before convergence is achieved. This probably accounts for much of its use, since its mathematical convergence is extremely slow. In fact, (18.5.21) can be used with $\mathbf{H} = 0$, in which case the solution is not regularized at all, and full convergence would be disastrous! This is called *Van Cittert's method* and goes back to the 1930s. A number of iterations the order of 1000 is not uncommon [7].

Deterministic Constraints: Projections onto Convex Sets

A set of possible underlying functions (or images) $\{\hat{\mathbf{u}}\}$ is said to be *convex* if, for any two elements $\hat{\mathbf{u}}_a$ and $\hat{\mathbf{u}}_b$ in the set, all the linearly interpolated combinations

$$(1-\eta)\hat{\mathbf{u}}_a + \eta\hat{\mathbf{u}}_b \qquad 0 \le \eta \le 1 \tag{18.5.23}$$

are also in the set. Many *deterministic constraints* that one might want to impose on the solution $\hat{\mathbf{u}}$ to an inverse problem in fact define convex sets, for example:

- positivity
- compact support (i.e., zero value outside of a certain region)

- known bounds (i.e., $u_L(x) \leq \hat{u}(x) \leq u_U(x)$ for specified functions u_L and u_U).

(In this last case, the bounds might be related to an initial estimate and its error bars, e.g., $\hat{u}_0(x) \pm \gamma\sigma(x)$, where γ is of order 1 or 2.) Notice that these, and similar, constraints can be either in the image space, or in the Fourier transform space, or (in fact) in the space of any linear transformation of $\hat{\mathbf{u}}$.

If C_i is a convex set, then $\mathcal{P}_i$ is called a *nonexpansive projection operator* onto that set if (i) $\mathcal{P}_i$ leaves unchanged any $\hat{\mathbf{u}}$ already in C_i, and (ii) $\mathcal{P}_i$ maps any $\hat{\mathbf{u}}$ outside C_i to the *closest* element of C_i, in the sense that

$$|\mathcal{P}_i\hat{\mathbf{u}} - \hat{\mathbf{u}}| \leq |\hat{\mathbf{u}}_a - \hat{\mathbf{u}}| \quad \text{for all } \hat{\mathbf{u}}_a \text{ in } C_i \tag{18.5.24}$$

While this definition sounds complicated, examples are very simple: A nonexpansive projection onto the set of positive $\hat{\mathbf{u}}$'s is "set all negative components of $\hat{\mathbf{u}}$ equal to zero." A nonexpansive projection onto the set of $\hat{u}(x)$'s bounded by $u_L(x) \leq \hat{u}(x) \leq u_U(x)$ is "set all values less than the lower bound equal to that bound, and set all values greater than the upper bound equal to *that* bound." A nonexpansive projection onto functions with compact support is "zero the values outside of the region of support."

The usefulness of these definitions is the following remarkable theorem: Let C be the intersection of m convex sets $C_1, C_2, \ldots, C_m$. Then the iteration

$$\hat{\mathbf{u}}^{(k+1)} = (\mathcal{P}_1\mathcal{P}_2\cdots\mathcal{P}_m)\hat{\mathbf{u}}^{(k)} \tag{18.5.25}$$

will converge to C from all starting points, as $k \to \infty$. Also, if C is empty (there is no intersection), then the iteration will have no limit point. Application of this theorem is called the *method of projections onto convex sets* or sometimes *POCS* [7].

A generalization of the POCS theorem is that the $\mathcal{P}_i$'s can be replaced by a set of $\mathcal{T}_i$'s,

$$\mathcal{T}_i \equiv \mathbf{1} + \beta_i(\mathcal{P}_i - \mathbf{1}) \qquad 0 < \beta_i < 2 \tag{18.5.26}$$

A well-chosen set of β_i's can accelerate the convergence to the intersection set C.

Some inverse problems can be completely solved by iteration (18.5.25) alone! For example, a problem that occurs in both astronomical imaging and X-ray diffraction work is to recover an image given only the *modulus* of its Fourier transform (equivalent to its power spectrum or autocorrelation) and not the *phase*. Here three convex sets can be utilized: the set of all images whose Fourier transform has the specified modulus to within specified error bounds; the set of all positive images; and the set of all images with zero intensity outside of some specified region. In this case the POCS iteration (18.5.25) cycles among these three, imposing each constraint in turn; FFTs are used to get in and out of Fourier space each time the Fourier constraint is imposed.

The specific application of POCS to constraints alternately in the spatial and Fourier domains is also known as the *Gerchberg-Saxton* algorithm [8]. While this algorithm is non-expansive, and is frequently convergent in practice, it has not been proved to converge in all cases [9]. In the phase-retrieval problem mentioned above, the algorithm often "gets stuck" on a plateau for many iterations before making sudden, dramatic improvements. As many as 10^4 to 10^5 iterations are sometimes

necessary. (For "unsticking" procedures, see [10].) The uniqueness of the solution is also not well understood, although for two-dimensional images of reasonable complexity it is believed to be unique.

Deterministic constraints can be incorporated, via projection operators, into iterative methods of linear regularization. In particular, rearranging terms somewhat, we can write the iteration (18.5.21) as

$$\hat{\mathbf{u}}^{(k+1)} = [\mathbf{1} - \epsilon\lambda\mathbf{H}] \cdot \hat{\mathbf{u}}^{(k)} + \epsilon\mathbf{A}^T \cdot (\mathbf{b} - \mathbf{A} \cdot \hat{\mathbf{u}}^{(k)}) \tag{18.5.27}$$

If the iteration is modified by the insertion of projection operators at each step

$$\hat{\mathbf{u}}^{(k+1)} = (\mathcal{P}_1\mathcal{P}_2 \cdots \mathcal{P}_m)[\mathbf{1} - \epsilon\lambda\mathbf{H}] \cdot \hat{\mathbf{u}}^{(k)} + \epsilon\mathbf{A}^T \cdot (\mathbf{b} - \mathbf{A} \cdot \hat{\mathbf{u}}^{(k)}) \tag{18.5.28}$$

(or, instead of $\mathcal{P}_i$'s, the $\mathcal{T}_i$ operators of equation 18.5.26), then it can be shown that the convergence condition (18.5.22) is unmodified, and the iteration will converge to minimize the quadratic functional (18.5.6) subject to the desired nonlinear deterministic constraints. See [7] for references to more sophisticated, and faster converging, iterations along these lines.

CITED REFERENCES AND FURTHER READING:

Phillips, D.L. 1962, *Journal of the Association for Computing Machinery*, vol. 9, pp. 84–97. [1]

Twomey, S. 1963, *Journal of the Association for Computing Machinery*, vol. 10, pp. 97–101. [2]

Twomey, S. 1977, *Introduction to the Mathematics of Inversion in Remote Sensing and Indirect Measurements* (Amsterdam: Elsevier). [3]

Craig, I.J.D., and Brown, J.C. 1986, *Inverse Problems in Astronomy* (Bristol, U.K.: Adam Hilger). [4]

Tikhonov, A.N., and Arsenin, V.Y. 1977, *Solutions of Ill-Posed Problems* (New York: Wiley). [5]

Tikhonov, A.N., and Goncharsky, A.V. (eds.) 1987, *Ill-Posed Problems in the Natural Sciences* (Moscow: MIR).

Miller, K. 1970, *SIAM Journal on Mathematical Analysis*, vol. 1, pp. 52–74. [6]

Schafer, R.W., Mersereau, R.M., and Richards, M.A. 1981, *Proceedings of the IEEE*, vol. 69, pp. 432–450.

Biemond, J., Lagendijk, R.L., and Mersereau, R.M. 1990, *Proceedings of the IEEE*, vol. 78, pp. 856–883. [7]

Gerchberg, R.W., and Saxton, W.O. 1972, *Optik*, vol. 35, pp. 237–246. [8]

Fienup, J.R. 1982, *Applied Optics*, vol. 15, pp. 2758–2769. [9]

Fienup, J.R., and Wackerman, C.C. 1986, *Journal of the Optical Society of America A*, vol. 3, pp. 1897–1907. [10]

18.6 Backus-Gilbert Method

The *Backus-Gilbert method* [1,2] (see, e.g., [3] or [4] for summaries) differs from other regularization methods in the nature of its functionals $\mathcal{A}$ and $\mathcal{B}$. For $\mathcal{B}$, the method seeks to maximize the *stability* of the solution $\hat{u}(x)$ rather than, in the first instance, its smoothness. That is,

$$\mathcal{B} \equiv \mathrm{Var}[\hat{u}(x)] \tag{18.6.1}$$

is used as a measure of how much the solution $\widehat{u}(x)$ varies as the data vary within their measurement errors. Note that this variance is not the expected deviation of $\widehat{u}(x)$ from the true $u(x)$ — that will be constrained by $\mathcal{A}$ — but rather measures the expected experiment-to-experiment scatter among estimates $\widehat{u}(x)$ if the whole experiment were to be repeated many times.

For $\mathcal{A}$ the Backus-Gilbert method looks at the relationship between the solution $\widehat{u}(x)$ and the true function $u(x)$, and seeks to make the mapping between these as close to the identity map as possible in the limit of error-free data. The method is linear, so the relationship between $\widehat{u}(x)$ and $u(x)$ can be written as

$$\widehat{u}(x) = \int \widehat{\delta}(x, x')u(x')dx' \tag{18.6.2}$$

for some so-called *resolution function* or *averaging kernel* $\widehat{\delta}(x, x')$. The Backus-Gilbert method seeks to minimize the width or *spread* of $\widehat{\delta}$ (that is, maximize the resolving power). $\mathcal{A}$ is chosen to be some positive measure of the spread.

While Backus-Gilbert's philosophy is thus rather different from that of Phillips-Twomey and related methods, in practice the differences between the methods are less than one might think. A *stable* solution is almost inevitably bound to be *smooth*: The wild, unstable oscillations that result from an unregularized solution are always exquisitely sensitive to small changes in the data. Likewise, making $\widehat{u}(x)$ close to $u(x)$ inevitably will bring error-free data into agreement with the model. Thus $\mathcal{A}$ and $\mathcal{B}$ play roles closely analogous to their corresponding roles in the previous two sections.

The principal advantage of the Backus-Gilbert formulation is that it gives good control over just those properties that it seeks to measure, namely stability and resolving power. Moreover, in the Backus-Gilbert method, the choice of λ (playing its usual role of compromise between $\mathcal{A}$ and $\mathcal{B}$) is conventionally made, or at least can easily be made, *before* any actual data are processed. One's uneasiness at making a *post hoc*, and therefore potentially subjectively biased, choice of λ is thus removed. Backus-Gilbert is often recommended as the method of choice for designing, and predicting the performance of, experiments that require data inversion.

Let's see how this all works. Starting with equation (18.4.5),

$$c_i \equiv s_i + n_i = \int r_i(x)u(x)dx + n_i \tag{18.6.3}$$

and building in linearity from the start, we seek a set of *inverse response kernels* $q_i(x)$ such that

$$\widehat{u}(x) = \sum_i q_i(x)c_i \tag{18.6.4}$$

is the desired estimator of $u(x)$. It is useful to define the integrals of the response kernels for each data point,

$$R_i \equiv \int r_i(x)dx \tag{18.6.5}$$

Substituting equation (18.6.4) into equation (18.6.3), and comparing with equation (18.6.2), we see that

$$\widehat{\delta}(x,x') = \sum_i q_i(x) r_i(x') \tag{18.6.6}$$

We can require this averaging kernel to have unit area at every x, giving

$$1 = \int \widehat{\delta}(x,x')dx' = \sum_i q_i(x) \int r_i(x')dx' = \sum_i q_i(x) R_i \equiv \mathbf{q}(x) \cdot \mathbf{R} \tag{18.6.7}$$

where $\mathbf{q}(x)$ and $\mathbf{R}$ are each vectors of length N, the number of measurements.

Standard propagation of errors, and equation (18.6.1), give

$$\mathcal{B} = \mathrm{Var}[\widehat{u}(x)] = \sum_i \sum_j q_i(x) S_{ij} q_j(x) = \mathbf{q}(x) \cdot \mathbf{S} \cdot \mathbf{q}(x) \tag{18.6.8}$$

where S_{ij} is the covariance matrix (equation 18.4.6). If one can neglect off-diagonal covariances (as when the errors on the c_i's are independent), then $S_{ij} = \delta_{ij}\sigma_i^2$ is diagonal.

We now need to define a measure of the width or spread of $\widehat{\delta}(x,x')$ at each value of x. While many choices are possible, Backus and Gilbert choose the second moment of its square. This measure becomes the functional $\mathcal{A}$,

$$\begin{aligned} \mathcal{A} \equiv w(x) &= \int (x'-x)^2 [\widehat{\delta}(x,x')]^2 dx' \\ &= \sum_i \sum_j q_i(x) W_{ij}(x) q_j(x) \equiv \mathbf{q}(x) \cdot \mathbf{W}(x) \cdot \mathbf{q}(x) \end{aligned} \tag{18.6.9}$$

where we have here used equation (18.6.6) and defined the *spread matrix* $\mathbf{W}(x)$ by

$$W_{ij}(x) \equiv \int (x'-x)^2 r_i(x') r_j(x') dx' \tag{18.6.10}$$

The functions $q_i(x)$ are now determined by the minimization principle

$$\text{minimize:} \quad \mathcal{A} + \lambda \mathcal{B} = \mathbf{q}(x) \cdot [\mathbf{W}(x) + \lambda \mathbf{S}] \cdot \mathbf{q}(x) \tag{18.6.11}$$

subject to the constraint (18.6.7) that $\mathbf{q}(x) \cdot \mathbf{R} = 1$.

The solution of equation (18.6.11) is

$$\mathbf{q}(x) = \frac{[\mathbf{W}(x) + \lambda \mathbf{S}]^{-1} \cdot \mathbf{R}}{\mathbf{R} \cdot [\mathbf{W}(x) + \lambda \mathbf{S}]^{-1} \cdot \mathbf{R}} \tag{18.6.12}$$

(Reference [4] gives an accessible proof.) For any particular data set $\mathbf{c}$ (set of measurements c_i), the solution $\widehat{u}(x)$ is thus

$$\widehat{u}(x) = \frac{\mathbf{c} \cdot [\mathbf{W}(x) + \lambda \mathbf{S}]^{-1} \cdot \mathbf{R}}{\mathbf{R} \cdot [\mathbf{W}(x) + \lambda \mathbf{S}]^{-1} \cdot \mathbf{R}} \tag{18.6.13}$$

(Don't let this notation mislead you into inverting the full matrix $\mathbf{W}(x) + \lambda\mathbf{S}$. You only need to solve for some $\mathbf{y}$ the linear system $(\mathbf{W}(x) + \lambda\mathbf{S}) \cdot \mathbf{y} = \mathbf{R}$, and then substitute $\mathbf{y}$ into both the numerators and denominators of 18.6.12 or 18.6.13.)

Equations (18.6.12) and (18.6.13) have a completely different character from the linearly regularized solutions to (18.5.7) and (18.5.8). The vectors and matrices in (18.6.12) all have size N, the number of measurements. There is no discretization of the underlying variable x, so M does not come into play at all. One solves a different $N \times N$ set of linear equations for each desired value of x. By contrast, in (18.5.8), one solves an $M \times M$ linear set, but only once. In general, the computational burden of repeatedly solving linear systems makes the Backus-Gilbert method unsuitable for other than one-dimensional problems.

How does one choose λ within the Backus-Gilbert scheme? As already mentioned, you can (in some cases *should*) make the choice *before* you see any actual data. For a given trial value of λ, and for a sequence of x's, use equation (18.6.12) to calculate $\mathbf{q}(x)$; then use equation (18.6.6) to plot the resolution functions $\widehat{\delta}(x, x')$ as a function of x'. These plots will exhibit the amplitude with which different underlying values x' contribute to the point $\widehat{u}(x)$ of your estimate. For the same value of λ, also plot the function $\sqrt{\mathrm{Var}[\widehat{u}(x)]}$ using equation (18.6.8). (You need an estimate of your measurement covariance matrix for this.)

As you change λ you will see very explicitly the trade-off between resolution and stability. Pick the value that meets your needs. You can even choose λ to be a function of x, $\lambda = \lambda(x)$, in equations (18.6.12) and (18.6.13), should you desire to do so. (This is one benefit of solving a separate set of equations for each x.) For the chosen value or values of λ, you now have a quantitative understanding of your inverse solution procedure. This can prove invaluable if — once you are processing real data — you need to judge whether a particular feature, a spike or jump for example, is genuine, and/or is actually resolved. The Backus-Gilbert method has found particular success among geophysicists, who use it to obtain information about the structure of the Earth (e.g., density run with depth) from seismic travel time data.

CITED REFERENCES AND FURTHER READING:

Backus, G.E., and Gilbert, F. 1968, *Geophysical Journal of the Royal Astronomical Society*, vol. 16, pp. 169–205. [1]

Backus, G.E., and Gilbert, F. 1970, *Philosophical Transactions of the Royal Society of London A*, vol. 266, pp. 123–192. [2]

Parker, R.L. 1977, *Annual Review of Earth and Planetary Science*, vol. 5, pp. 35–64. [3]

Loredo, T.J., and Epstein, R.I. 1989, *Astrophysical Journal*, vol. 336, pp. 896–919. [4]

18.7 Maximum Entropy Image Restoration

Above, we commented that the association of certain inversion methods with Bayesian arguments is more historical accident than intellectual imperative. *Maximum entropy methods*, so-called, are notorious in this regard; to summarize these methods without some, at least introductory, Bayesian invocations would be to serve a steak without the sizzle, or a sundae without the cherry. We should

also comment in passing that the connection between maximum entropy inversion methods, considered here, and maximum entropy spectral estimation, discussed in §13.7, is rather abstract. For practical purposes the two techniques, though both named *maximum entropy method* or *MEM*, are unrelated.

Bayes' Theorem, which follows from the standard axioms of probability, relates the conditional probabilities of two events, say A and B:

$$\text{Prob}(A|B) = \text{Prob}(A)\frac{\text{Prob}(B|A)}{\text{Prob}(B)} \tag{18.7.1}$$

Here $\text{Prob}(A|B)$ is the probability of A *given* that B has occurred, and similarly for $\text{Prob}(B|A)$, while $\text{Prob}(A)$ and $\text{Prob}(B)$ are unconditional probabilities.

"Bayesians" (so-called) adopt a broader interpretation of probabilities than do so-called "frequentists." To a Bayesian, $P(A|B)$ is a measure of the degree of plausibility of A (given B) on a scale ranging from zero to one. In this broader view, A and B need not be repeatable events; they can be propositions or hypotheses. The equations of probability theory then become a set of consistent rules for conducting inference [1,2]. Since plausibility is itself always conditioned on some, perhaps unarticulated, set of assumptions, all Bayesian probabilities are viewed as conditional on some collective background information I.

Suppose H is some hypothesis. Even before there exist any explicit data, a Bayesian can assign to H some degree of plausibility $\text{Prob}(H|I)$, called the "Bayesian prior." Now, when some data D_1 comes along, Bayes theorem tells how to reassess the plausibility of H,

$$\text{Prob}(H|D_1 I) = \text{Prob}(H|I)\frac{\text{Prob}(D_1|HI)}{\text{Prob}(D_1|I)} \tag{18.7.2}$$

The factor in the numerator on the right of equation (18.7.2) is calculable as the probability of a data set *given* the hypothesis (compare with "likelihood" in §15.1). The denominator, called the "prior predictive probability" of the data, is in this case merely a normalization constant which can be calculated by the requirement that the probability of all hypotheses should sum to unity. (In other Bayesian contexts, the prior predictive probabilities of two qualitatively different models can be used to assess their relative plausibility.)

If some additional data D_2 comes along tomorrow, we can further refine our estimate of H's probability, as

$$\text{Prob}(H|D_2 D_1 I) = \text{Prob}(H|D_1 I)\frac{\text{Prob}(D_2|H D_1 I)}{\text{Prob}(D_1|D_1 I)} \tag{18.7.3}$$

Using the product rule for probabilities, $\text{Prob}(AB|C) = \text{Prob}(A|C)\text{Prob}(B|AC)$, we find that equations (18.7.2) and (18.7.3) imply

$$\text{Prob}(H|D_2 D_1 I) = \text{Prob}(H|I)\frac{\text{Prob}(D_2 D_1|HI)}{\text{Prob}(D_2 D_1|I)} \tag{18.7.4}$$

which shows that we would have gotten the same answer if all the data $D_1 D_2$ had been taken together.

From a Bayesian perspective, inverse problems are inference problems [3,4]. The underlying parameter set **u** is a hypothesis whose probability, given the measured data values **c**, and the Bayesian prior $\text{Prob}(\mathbf{u}|I)$ can be calculated. We might want to report a single "best" inverse **u**, the one that maximizes

$$\text{Prob}(\mathbf{u}|\mathbf{c}I) = \text{Prob}(\mathbf{c}|\mathbf{u}I)\frac{\text{Prob}(\mathbf{u}|I)}{\text{Prob}(\mathbf{c}|I)} \tag{18.7.5}$$

over all possible choices of **u**. Bayesian analysis also admits the possibility of reporting additional information that characterizes the region of possible **u**'s with high relative probability, the so-called "posterior bubble" in **u**.

The calculation of the probability of the data **c**, given the hypothesis **u** proceeds exactly as in the maximum likelihood method. For Gaussian errors, e.g., it is given by

$$\text{Prob}(\mathbf{c}|\mathbf{u}I) = \exp(-\frac{1}{2}\chi^2)\Delta u_1 \Delta u_2 \cdots \Delta u_M \tag{18.7.6}$$

where χ^2 is calculated from **u** and **c** using equation (18.4.9), and the Δu_μ's are constant, small ranges of the components of **u** whose actual magnitude is irrelevant, because they do not depend on **u** (compare equations 15.1.3 and 15.1.4).

In maximum likelihood estimation we, in effect, chose the prior $\text{Prob}(\mathbf{u}|I)$ to be constant. That was a luxury that we could afford when estimating a small number of parameters from a large amount of data. Here, the number of "parameters" (components of **u**) is comparable to or larger than the number of measured values (components of **c**); we *need* to have a nontrivial prior, $\text{Prob}(\mathbf{u}|I)$, to resolve the degeneracy of the solution.

In maximum entropy image restoration, that is where *entropy* comes in. The entropy of a physical system in some macroscopic state, usually denoted S, is the logarithm of the number of microscopically distinct configurations that all have the same macroscopic observables (i.e., consistent with the observed macroscopic state). Actually, we will find it useful to denote the *negative* of the entropy, also called the *negentropy*, by $H \equiv -S$ (a notation that goes back to Boltzmann). In situations where there is reason to believe that the *a priori* probabilities of the *microscopic* configurations are all the same (these situations are called *ergodic*), then the Bayesian prior $\text{Prob}(\mathbf{u}|I)$ for a *macroscopic* state with entropy S is proportional to $\exp(S)$ or $\exp(-H)$.

MEM uses this concept to assign a prior probability to any given underlying function **u**. For example [5-7], suppose that the measurement of luminance in each pixel is quantized to (in some units) an integer value. Let

$$U = \sum_{\mu=1}^{M} u_\mu \tag{18.7.7}$$

be the total number of luminance quanta in the whole image. Then we can base our "prior" on the notion that each luminance quantum has an equal *a priori* chance of being in any pixel. (See [8] for a more abstract justification of this idea.) The number of ways of getting a particular configuration **u** is

$$\frac{U!}{u_1!u_2!\cdots u_M!} \propto \exp\left[-\sum_\mu u_\mu \ln(u_\mu/U) + \frac{1}{2}\left(\ln U - \sum_\mu \ln u_\mu\right)\right] \tag{18.7.8}$$

Here the left side can be understood as the number of distinct orderings of all the luminance quanta, divided by the numbers of equivalent reorderings within each pixel, while the right side follows by Stirling's approximation to the factorial function. Taking the negative of the logarithm, and neglecting terms of order $\log U$ in the presence of terms of order U, we get the negentropy

$$H(\mathbf{u}) = \sum_{\mu=1}^{M} u_\mu \ln(u_\mu/U) \tag{18.7.9}$$

From equations (18.7.5), (18.7.6), and (18.7.9) we now seek to maximize

$$\text{Prob}(\mathbf{u}|\mathbf{c}) \propto \exp\left[-\frac{1}{2}\chi^2\right] \exp[-H(\mathbf{u})] \tag{18.7.10}$$

or, equivalently,

$$\text{minimize:} \quad -\ln[\text{Prob}(\mathbf{u}|\mathbf{c})] = \frac{1}{2}\chi^2[\mathbf{u}] + H(\mathbf{u}) = \frac{1}{2}\chi^2[\mathbf{u}] + \sum_{\mu=1}^{M} u_\mu \ln(u_\mu/U) \tag{18.7.11}$$

This ought to remind you of equation (18.4.11), or equation (18.5.6), or in fact any of our previous minimization principles along the lines of $\mathcal{A} + \lambda\mathcal{B}$, where $\lambda\mathcal{B} = H(\mathbf{u})$ is a regularizing operator. Where is λ? We need to put it in for exactly the reason discussed following equation (18.4.11): Degenerate inversions are likely to be able to achieve unrealistically small values of χ^2. We need an adjustable parameter to bring χ^2 into its expected narrow statistical range of $N \pm (2N)^{1/2}$. The discussion at the beginning of §18.4 showed that it makes no difference which term we attach the λ to. For consistency in notation, we absorb a factor 2 into λ and put it on the entropy term. (Another way to see the necessity of an undetermined λ factor is to note that it is necessary if our minimization principle is to be invariant under changing the units in which $\mathbf{u}$ is quantized, e.g., if an 8-bit analog-to-digital converter is replaced by a 12-bit one.) We can now also put "hats" back to indicate that this is the procedure for obtaining our chosen statistical estimator:

$$\text{minimize:} \quad \mathcal{A} + \lambda\mathcal{B} = \chi^2[\hat{\mathbf{u}}] + \lambda H(\hat{\mathbf{u}}) = \chi^2[\hat{\mathbf{u}}] + \lambda \sum_{\mu=1}^{M} \hat{u}_\mu \ln(\hat{u}_\mu) \tag{18.7.12}$$

(Formally, we might also add a second Lagrange multiplier $\lambda' U$, to constrain the total intensity U to be constant.)

It is not hard to see that the negentropy, $H(\hat{\mathbf{u}})$, is in fact a regularizing operator, similar to $\hat{\mathbf{u}} \cdot \hat{\mathbf{u}}$ (equation 18.4.11) or $\hat{\mathbf{u}} \cdot \mathbf{H} \cdot \hat{\mathbf{u}}$ (equation 18.5.6). The following of its properties are noteworthy:

1. When U is held constant, $H(\hat{\mathbf{u}})$ is minimized for $\hat{u}_\mu = U/M = \text{constant}$, so it smooths in the sense of trying to achieve a constant solution, similar to equation (18.5.4). The fact that the constant solution is a minimum follows from the fact that the second derivative of $u \ln u$ is positive.

2. Unlike equation (18.5.4), however, $H(\hat{\mathbf{u}})$ is *local*, in the sense that it does not difference neighboring pixels. It simply sums some function f, here

$$f(u) = u \ln u \tag{18.7.13}$$

over all pixels; it is invariant, in fact, under a complete scrambling of the pixels in an image. This form implies that $H(\hat{\mathbf{u}})$ is not seriously increased by the occurrence of a small number of very bright pixels (point sources) embedded in a low-intensity smooth background.
3. $H(\hat{\mathbf{u}})$ goes to infinite slope as any one pixel goes to zero. This causes it to enforce positivity of the image, without the necessity of additional deterministic constraints.
4. The biggest difference between $H(\hat{\mathbf{u}})$ and the other regularizing operators that we have met is that $H(\hat{\mathbf{u}})$ is not a quadratic functional of $\hat{\mathbf{u}}$, so the equations obtained by varying equation (18.7.12) are *nonlinear*. This fact is itself worthy of some additional discussion.

Nonlinear equations are harder to solve than linear equations. For image processing, however, the large number of equations usually dictates an iterative solution procedure, even for linear equations, so the practical effect of the nonlinearity is somewhat mitigated. Below, we will summarize some of the methods that are successfully used for MEM inverse problems.

For some problems, notably the problem in radio-astronomy of image recovery from an incomplete set of Fourier coefficients, the superior performance of MEM inversion can be, in part, traced to the nonlinearity of $H(\hat{\mathbf{u}})$. One way to see this [5] is to consider the limit of perfect measurements $\sigma_i \to 0$. In this case the χ^2 term in the minimization principle (18.7.12) gets replaced by a set of constraints, each with its own Lagrange multiplier, requiring agreement between model and data; that is,

$$\text{minimize:} \quad \sum_j \lambda_j \left[c_j - \sum_\mu R_{j\mu} \hat{u}_\mu \right] + H(\hat{\mathbf{u}}) \tag{18.7.14}$$

(cf. equation 18.4.7). Setting the formal derivative with respect to $\hat{u}_\mu$ to zero gives

$$\frac{\partial H}{\partial \hat{u}_\mu} = f'(\hat{u}_\mu) = \sum_j \lambda_j R_{j\mu} \tag{18.7.15}$$

or defining a function G as the inverse function of f',

$$\hat{u}_\mu = G\left(\sum_j \lambda_j R_{j\mu} \right) \tag{18.7.16}$$

This solution is only formal, since the λ_j's must be found by requiring that equation (18.7.16) satisfy all the constraints built into equation (18.7.14). However, equation (18.7.16) does show the crucial fact that if G is *linear*, then the solution $\hat{\mathbf{u}}$ contains *only* a linear combination of basis functions $R_{j\mu}$ corresponding to actual measurements j. This is equivalent to setting unmeasured c_j's to zero. Notice that the principal solution obtained from equation (18.4.11) in fact has a linear G.

In the problem of incomplete Fourier image reconstruction, the typical $R_{j\mu}$ has the form $\exp(-2\pi i \mathbf{k}_j \cdot \mathbf{x}_\mu)$, where $\mathbf{x}_\mu$ is a two-dimensional vector in the image space and $\mathbf{k}_\mu$ is a two-dimensional wave-vector. If an image contains strong point sources, then the effect of setting unmeasured c_j's to zero is to produce sidelobe ripples throughout the image plane. These ripples can mask any actual extended, low-intensity image features lying between the point sources. If, however, the slope of G is smaller for small values of its argument, larger for large values, then ripples in low-intensity portions of the image are relatively suppressed, while strong point sources will be relatively sharpened ("superresolution"). This behavior on the slope of G is equivalent to requiring $f'''(u) < 0$. For $f(u) = u \ln u$, we in fact have $f'''(u) = -1/u^2 < 0$.

In more picturesque language, the nonlinearity acts to "create" nonzero values for the unmeasured c_i's, so as to suppress the low-intensity ripple and sharpen the point sources.

Is MEM Really Magical?

How unique is the negentropy functional (18.7.9)? Recall that that equation is based on the assumption that luminance elements are *a priori* distributed over the pixels uniformly. If we instead had some other preferred *a priori* image in mind, one with pixel intensities m_μ, then it is easy to show that the negentropy becomes

$$H(\mathbf{u}) = \sum_{\mu=1}^{M} u_\mu \ln(u_\mu/m_\mu) + \text{constant} \tag{18.7.17}$$

(the constant can then be ignored). All the rest of the discussion then goes through.

More fundamentally, and despite statements by zealots to the contrary [7], there is actually nothing universal about the functional form $f(u) = u \ln u$. In some other physical situations (for example, the entropy of an electromagnetic field in the limit of many photons per mode, as in radio-astronomy) the physical negentropy functional is actually $f(u) = -\ln u$ (see [5] for other examples). In general, the question, "Entropy of what?" is not uniquely answerable in any particular situation. (See reference [9] for an attempt at articulating a more general principle that reduces to one or another entropy functional under appropriate circumstances.)

The four numbered properties summarized above, plus the desirable sign for nonlinearity, $f'''(u) < 0$, are all as true for $f(u) = -\ln u$ as for $f(u) = u \ln u$. In fact these properties are shared by a nonlinear function as simple as $f(u) = -\sqrt{u}$, which has no information theoretic justification at all (no logarithms!). MEM reconstructions of test images using any of these entropy forms are virtually indistinguishable [5].

By all available evidence, MEM seems to be neither more nor less than one usefully nonlinear version of the general regularization scheme $\mathcal{A} + \lambda\mathcal{B}$ that we have by now considered in many forms. Its peculiarities become strengths when applied to the reconstruction from incomplete Fourier data of images that are expected to be dominated by very bright point sources, but which also contain interesting low-intensity, extended sources. For images of some other character, there is no reason to suppose that MEM methods will generally dominate other regularization schemes, either ones already known or yet to be invented.

Algorithms for MEM

The goal is to find the vector $\hat{\mathbf{u}}$ that minimizes $\mathcal{A} + \lambda\mathcal{B}$ where in the notation of equations (18.5.5), (18.5.6), and (18.7.13),

$$\mathcal{A} = |\mathbf{b} - \mathbf{A} \cdot \hat{\mathbf{u}}|^2 \qquad \mathcal{B} = \sum_\mu f(\hat{u}_\mu) \tag{18.7.18}$$

Compared with a "general" minimization problem, we have the advantage that we can compute the gradients and the second partial derivative matrices (Hessian matrices) explicitly,

$$\begin{aligned} \nabla\mathcal{A} &= 2(\mathbf{A}^T \cdot \mathbf{A} \cdot \hat{\mathbf{u}} - \mathbf{A}^T \cdot \mathbf{b}) & \frac{\partial^2\mathcal{A}}{\partial\hat{u}_\mu \partial\hat{u}_\rho} &= [2\mathbf{A}^T \cdot \mathbf{A}]_{\mu\rho} \\ [\nabla\mathcal{B}]_\mu &= f'(\hat{u}_\mu) & \frac{\partial^2\mathcal{B}}{\partial\hat{u}_\mu \partial\hat{u}_\rho} &= \delta_{\mu\rho} f''(\hat{u}_\mu) \end{aligned} \tag{18.7.19}$$

It is important to note that while $\mathcal{A}$'s second partial derivative matrix cannot be stored (its size is the square of the number of pixels), it can be applied to any vector by first applying $\mathbf{A}$, then $\mathbf{A}^T$. In the case of reconstruction from incomplete Fourier data, or in the case of convolution with a translation invariant point spread function, these applications will typically involve several FFTs. Likewise, the calculation of the gradient $\nabla\mathcal{A}$ will involve FFTs in the application of $\mathbf{A}$ and $\mathbf{A}^T$.

While some success has been achieved with the classical conjugate gradient method (§10.6), it is often found that the nonlinearity in $f(u) = u \ln u$ causes problems. Attempted steps that give $\hat{\mathbf{u}}$ with even one negative value must be cut in magnitude, sometimes so severely as to slow the solution to a crawl. The underlying problem is that the conjugate gradient method develops its information about the inverse of the Hessian matrix a bit at a time, while changing its location in the search space. When a nonlinear function is quite different from a pure quadratic form, the old information becomes obsolete before it gets usefully exploited.

Skilling and collaborators [6,7,10,11] developed a complicated but highly successful scheme, wherein a minimum is repeatedly sought not along a single search direction, but in a small- (typically three-) dimensional subspace, spanned by vectors that are calculated anew at each landing point. The subspace basis vectors are chosen in such a way as to avoid directions leading to negative values. One of the most successful choices is the three-dimensional subspace spanned by the vectors with components given by

$$\begin{aligned} e_\mu^{(1)} &= \hat{u}_\mu [\nabla\mathcal{A}]_\mu \\ e_\mu^{(2)} &= \hat{u}_\mu [\nabla\mathcal{B}]_\mu \\ e_\mu^{(3)} &= \frac{\hat{u}_\mu \sum_\rho (\partial^2\mathcal{A}/\partial\hat{u}_\mu\partial\hat{u}_\rho)\hat{u}_\rho [\nabla\mathcal{B}]_\rho}{\sqrt{\sum_\rho \hat{u}_\rho ([\nabla\mathcal{B}]_\rho)^2}} - \frac{\hat{u}_\mu \sum_\rho (\partial^2\mathcal{A}/\partial\hat{u}_\mu\partial\hat{u}_\rho)\hat{u}_\rho [\nabla\mathcal{A}]_\rho}{\sqrt{\sum_\rho \hat{u}_\rho ([\nabla\mathcal{A}]_\rho)^2}} \end{aligned} \tag{18.7.20}$$

(In these equations there is no sum over μ.) The form of the $\mathbf{e}^{(3)}$ has some justification if one views dot products as occurring in a space with the metric $g_{\mu\nu} = \delta_{\mu\nu}/u_\mu$, chosen to make zero values "far away"; see [6].

Within the three-dimensional subspace, the three-component gradient and nine-component Hessian matrix are computed by projection from the large space, and the minimum in the subspace is estimated by (trivially) solving three simultaneous linear equations, as in §10.7, equation (10.7.4). The size of a step $\Delta\hat{\mathbf{u}}$ is required to be limited by the inequality

$$\sum_{\mu}(\Delta\hat{u}_\mu)^2/\hat{u}_\mu < (0.1 \text{ to } 0.5)U \tag{18.7.21}$$

Because the gradient directions $\nabla\mathcal{A}$ and $\nabla\mathcal{B}$ are separately available, it is possible to combine the minimum search with a simultaneous adjustment of λ so as finally to satisfy the desired constraint. There are various further tricks employed.

A less general, but in practice often equally satisfactory, approach is due to Cornwell and Evans [12]. Here, noting that $\mathcal{B}$'s Hessian (second partial derivative) matrix is diagonal, one asks whether there is a useful diagonal approximation to $\mathcal{A}$'s Hessian, namely $2\mathbf{A}^T \cdot \mathbf{A}$. If Λ_μ denotes the diagonal components of such an approximation, then a useful step in $\hat{\mathbf{u}}$ would be

$$\Delta\hat{u}_\mu = -\frac{1}{\Lambda_\mu + \lambda f''(\hat{u}_\mu)}(\nabla\mathcal{A} + \lambda\nabla\mathcal{B}) \tag{18.7.22}$$

(again compare equation 10.7.4). Even more extreme, one might seek an approximation with constant diagonal elements, $\Lambda_\mu = \Lambda$, so that

$$\Delta\hat{u}_\mu = -\frac{1}{\Lambda + \lambda f''(\hat{u}_\mu)}(\nabla\mathcal{A} + \lambda\nabla\mathcal{B}) \tag{18.7.23}$$

Since $\mathbf{A}^T \cdot \mathbf{A}$ has something of the nature of a doubly convolved point spread function, and since in real cases one often has a point spread function with a sharp central peak, even the more extreme of these approximations is often fruitful. One starts with a rough estimate of Λ obtained from the $A_{i\mu}$'s, e.g.,

$$\Lambda \sim \left\langle \sum_i [A_{i\mu}]^2 \right\rangle \tag{18.7.24}$$

An accurate value is not important, since in practice Λ is adjusted adaptively: If Λ is too large, then equation (18.7.23)'s steps will be too small (that is, larger steps in the same direction will produce even greater decrease in $\mathcal{A} + \lambda\mathcal{B}$). If Λ is too small, then attempted steps will land in an unfeasible region (negative values of $\hat{u}_\mu$), or will result in an increased $\mathcal{A} + \lambda\mathcal{B}$. There is an obvious similarity between the adjustment of Λ here and the Levenberg-Marquardt method of §15.5; this should not be too surprising, since MEM is closely akin to the problem of nonlinear least-squares fitting. Reference [12] also discusses how the value of $\Lambda + \lambda f''(\hat{u}_\mu)$ can be used to adjust the Lagrange multiplier λ so as to converge to the desired value of χ^2.

All practical MEM algorithms are found to require on the order of 30 to 50 iterations to converge. This convergence behavior is not now understood in any fundamental way.

"Bayesian" versus "Historic" Maximum Entropy

Several more recent developments in maximum entropy image restoration go under the rubric "Bayesian" to distinguish them from the previous "historic" methods. See [13] for details and references.

- Better priors: We already noted that the entropy functional (equation 18.7.13) is invariant under scrambling all pixels and has no notion of smoothness. The so-called "instrinsic correlation function" (ICF) model is similar enough to the entropy functional to allow similar algorithms, but it makes the values of neighboring pixels correlated, enforcing smoothness.
- Better estimation of λ: Above we chose λ to bring χ^2 into its expected narrow statistical range of $N \pm (2N)^{1/2}$. This in effect overestimates χ^2, however, since some effective number γ of parameters are being "fitted" in doing the reconstruction. A Bayesian approach leads to a self-consistent estimate of this γ and an objectively better choice for λ.

CITED REFERENCES AND FURTHER READING:

Jaynes, E.T. 1976, in *Foundations of Probability Theory, Statistical Inference, and Statistical Theories of Science*, W.L. Harper and C.A. Hooker, eds. (Dordrecht: Reidel). [1]

Jaynes, E.T. 1985, in *Maximum-Entropy and Bayesian Methods in Inverse Problems*, C.R. Smith and W.T. Grandy, Jr., eds. (Dordrecht: Reidel). [2]

Jaynes, E.T. 1984, in *SIAM-AMS Proceedings*, vol. 14, D.W. McLaughlin, ed. (Providence, RI: American Mathematical Society). [3]

Titterington, D.M. 1985, *Astronomy and Astrophysics*, vol. 144, 381–387. [4]

Narayan, R., and Nityananda, R. 1986, *Annual Review of Astronomy and Astrophysics*, vol. 24, pp. 127–170. [5]

Skilling, J., and Bryan, R.K. 1984, *Monthly Notices of the Royal Astronomical Society*, vol. 211, pp. 111–124. [6]

Burch, S.F., Gull, S.F., and Skilling, J. 1983, *Computer Vision, Graphics and Image Processing*, vol. 23, pp. 113–128. [7]

Skilling, J. 1989, in *Maximum Entropy and Bayesian Methods*, J. Skilling, ed. (Boston: Kluwer). [8]

Frieden, B.R. 1983, *Journal of the Optical Society of America*, vol. 73, pp. 927–938. [9]

Skilling, J., and Gull, S.F. 1985, in *Maximum-Entropy and Bayesian Methods in Inverse Problems*, C.R. Smith and W.T. Grandy, Jr., eds. (Dordrecht: Reidel). [10]

Skilling, J. 1986, in *Maximum Entropy and Bayesian Methods in Applied Statistics*, J.H. Justice, ed. (Cambridge: Cambridge University Press). [11]

Cornwell, T.J., and Evans, K.F. 1985, *Astronomy and Astrophysics*, vol. 143, pp. 77–83. [12]

Gull, S.F. 1989, in *Maximum Entropy and Bayesian Methods*, J. Skilling, ed. (Boston: Kluwer). [13]

13.10 Wavelet Transforms

Like the fast Fourier transform (FFT), the discrete wavelet transform (DWT) is a fast, linear operation that operates on a data vector whose length is an integer power of two, transforming it into a numerically different vector of the same length. Also like the FFT, the wavelet transform is invertible and in fact orthogonal — the inverse transform, when viewed as a big matrix, is simply the transpose of the transform. Both FFT and DWT, therefore, can be viewed as a rotation in function space, from the input space (or time) domain, where the basis functions are the unit vectors $\mathbf{e}_i$, or Dirac delta functions in the continuum limit, to a different domain. For the FFT, this new domain has basis functions that are the familiar sines and cosines. In the wavelet domain, the basis functions are somewhat more complicated and have the fanciful names "mother functions" and "wavelets."

Of course there are an infinity of possible bases for function space, almost all of them uninteresting! What makes the wavelet basis interesting is that, *unlike* sines and cosines, individual wavelet functions are quite localized in space; simultaneously, *like* sines and cosines, individual wavelet functions are quite localized in frequency or (more precisely) characteristic scale. As we will see below, the particular kind of dual localization achieved by wavelets renders large classes of functions and operators sparse, or sparse to some high accuracy, when transformed into the wavelet domain. Analogously with the Fourier domain, where a class of computations, like convolutions, become computationally fast, there is a large class of computations — those that can take advantage of sparsity — that become computationally fast in the wavelet domain [1].

Unlike sines and cosines, which define a unique Fourier transform, there is not one single unique set of wavelets; in fact, there are infinitely many possible sets. Roughly, the different sets of wavelets make different trade-offs between how compactly they are localized in space and how smooth they are. (There are further fine distinctions.)

Daubechies Wavelet Filter Coefficients

A particular set of wavelets is specified by a particular set of numbers, called *wavelet filter coefficients*. Here, we will largely restrict ourselves to wavelet filters in a class discovered by Daubechies [2]. This class includes members ranging from highly localized to highly smooth. The simplest (and most localized) member, often called *DAUB4*, has only four coefficients, $c_0, \ldots, c_3$. For the moment we specialize to this case for ease of notation.

Consider the following transformation matrix acting on a column vector of data to its right:

$$\begin{bmatrix} c_0 & c_1 & c_2 & c_3 & & & & & & & & \\ c_3 & -c_2 & c_1 & -c_0 & & & & & & & & \\ & & c_0 & c_1 & c_2 & c_3 & & & & & & \\ & & c_3 & -c_2 & c_1 & -c_0 & & & & & & \\ \vdots & \vdots & & & & & \ddots & & & & & \\ & & & & & & & c_0 & c_1 & c_2 & c_3 & \\ & & & & & & & c_3 & -c_2 & c_1 & -c_0 & \\ c_2 & c_3 & & & & & & & & c_0 & c_1 & \\ c_1 & -c_0 & & & & & & & & c_3 & -c_2 & \end{bmatrix} \tag{13.10.1}$$

Here blank entries signify zeroes. Note the structure of this matrix. The first row generates one component of the data convolved with the filter coefficients $c_0 \ldots, c_3$. Likewise the third, fifth, and other odd rows. If the even rows followed this pattern, offset by one, then the matrix would be a circulant, that is, an ordinary convolution that could be done by FFT methods. (Note how the last two rows wrap around like convolutions with periodic boundary conditions.) Instead of convolving with $c_0, \ldots, c_3$, however, the even rows perform a different convolution, with coefficients $c_3, -c_2, c_1, -c_0$. The action of the matrix, overall, is thus to perform two related convolutions, then to decimate each of them by half (throw away half the values), and interleave the remaining halves.

It is useful to think of the filter $c_0, \ldots, c_3$ as being a smoothing filter, call it H, something like a moving average of four points. Then, because of the minus signs,

the filter $c_3, -c_2, c_1, -c_0$, call it G, is *not* a smoothing filter. (In signal processing contexts, H and G are called *quadrature mirror filters* [3].) In fact, the c's are chosen so as to make G yield, insofar as possible, a *zero* response to a sufficiently smooth data vector. This is done by requiring the sequence $c_3, -c_2, c_1, -c_0$ to have a certain number of vanishing moments. When this is the case for p moments (starting with the zeroth), a set of wavelets is said to satisfy an "approximation condition of order p." This results in the output of H, decimated by half, accurately representing the data's "smooth" information. The output of G, also decimated, is referred to as the data's "detail" information [4].

For such a characterization to be useful, it must be possible to reconstruct the original data vector of length N from its $N/2$ smooth or s-components and its $N/2$ detail or d-components. That is effected by requiring the matrix (13.10.1) to be orthogonal, so that its inverse is just the transposed matrix

$$\begin{bmatrix}
c_0 & c_3 & & & \cdots & & & & & c_2 & c_1 \\
c_1 & -c_2 & & & \cdots & & & & & c_3 & -c_0 \\
c_2 & c_1 & c_0 & c_3 & & & & & & & \\
c_3 & -c_0 & c_1 & -c_2 & & & & & & & \\
 & & & & \ddots & & & & & & \\
 & & & & & c_2 & c_1 & c_0 & c_3 & & \\
 & & & & & c_3 & -c_0 & c_1 & -c_2 & & \\
 & & & & & & & c_2 & c_1 & c_0 & c_3 \\
 & & & & & & & c_3 & -c_0 & c_1 & -c_2
\end{bmatrix} \tag{13.10.2}$$

One sees immediately that matrix (13.10.2) is inverse to matrix (13.10.1) if and only if these two equations hold,

$$\begin{aligned} c_0^2 + c_1^2 + c_2^2 + c_3^2 &= 1 \\ c_2 c_0 + c_3 c_1 &= 0 \end{aligned} \tag{13.10.3}$$

If additionally we require the approximation condition of order $p = 2$, then two additional relations are required,

$$\begin{aligned} c_3 - c_2 + c_1 - c_0 &= 0 \\ 0c_3 - 1c_2 + 2c_1 - 3c_0 &= 0 \end{aligned} \tag{13.10.4}$$

Equations (13.10.3) and (13.10.4) are 4 equations for the 4 unknowns $c_0, \ldots, c_3$, first recognized and solved by Daubechies. The unique solution (up to a left-right reversal) is

$$\begin{aligned} c_0 &= (1 + \sqrt{3})/4\sqrt{2} \qquad c_1 = (3 + \sqrt{3})/4\sqrt{2} \\ c_2 &= (3 - \sqrt{3})/4\sqrt{2} \qquad c_3 = (1 - \sqrt{3})/4\sqrt{2} \end{aligned} \tag{13.10.5}$$

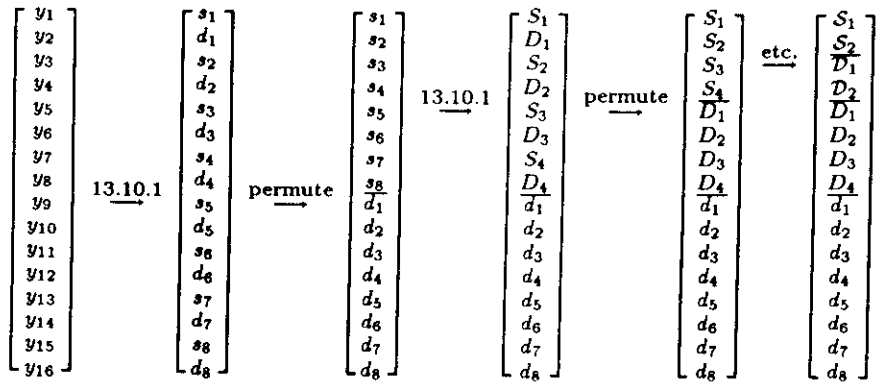
In fact, DAUB4 is only the most compact of a sequence of wavelet sets: If we had six coefficients instead of four, there would be three orthogonality requirements in equation (13.10.3) (with offsets of zero, two, and four), and we could require the vanishing of $p = 3$ moments in equation (13.10.4). In this case, DAUB6, the solution coefficients can also be expressed in closed form,

$$\begin{aligned} c_0 &= (1 + \sqrt{10} + \sqrt{5 + 2\sqrt{10}})/16\sqrt{2} \qquad c_1 = (5 + \sqrt{10} + 3\sqrt{5 + 2\sqrt{10}})/16\sqrt{2} \\ c_2 &= (10 - 2\sqrt{10} + 2\sqrt{5 + 2\sqrt{10}})/16\sqrt{2} \qquad c_3 = (10 - 2\sqrt{10} - 2\sqrt{5 + 2\sqrt{10}})/16\sqrt{2} \\ c_4 &= (5 + \sqrt{10} - 3\sqrt{5 + 2\sqrt{10}})/16\sqrt{2} \qquad c_5 = (1 + \sqrt{10} - \sqrt{5 + 2\sqrt{10}})/16\sqrt{2} \end{aligned} \tag{13.10.6}$$

For higher p, up to 10, Daubechies [2] has tabulated the coefficients numerically. The number of coefficients increases by two each time p is increased by one.

Discrete Wavelet Transform

We have not yet defined the discrete wavelet transform (DWT), but we are almost there: The DWT consists of applying a wavelet coefficient matrix like (13.10.1) *hierarchically*, first to the full data vector of length N, then to the "smooth" vector of length $N/2$, then to the "smooth-smooth" vector of length $N/4$, and so on until only a trivial number of "smooth-...-smooth" components (usually 2) remain. The procedure is sometimes called a *pyramidal algorithm* [4], for obvious reasons. The output of the DWT consists of these remaining components and all the "detail" components that were accumulated along the way. A diagram should make the procedure clear:



(13.10.7)

If the length of the data vector were a higher power of two, there would be more stages of applying (13.10.1) (or any other wavelet coefficients) and permuting. The endpoint will always be a vector with two $\mathcal{S}$'s and a hierarchy of $\mathcal{D}$'s, D's, d's, etc. Notice that once d's are generated, they simply propagate through to all subsequent stages.

A value d_i of any level is termed a "wavelet coefficient" of the original data vector; the final values $\mathcal{S}_1$, $\mathcal{S}_2$ should strictly be called "mother-function coefficients," although the term "wavelet coefficients" is often used loosely for both d's and final $\mathcal{S}$'s. Since the full procedure is a composition of orthogonal linear operations, the whole DWT is itself an orthogonal linear operator.

To invert the DWT, one simply reverses the procedure, starting with the smallest level of the hierarchy and working (in equation 13.10.7) from right to left. The inverse matrix (13.10.2) is of course used instead of the matrix (13.10.1).

As already noted, the matrices (13.10.1) and (13.10.2) embody periodic ("wrap-around") boundary conditions on the data vector. One normally accepts this is a minor inconvenience: the last few wavelet coefficients at each level of the hierarchy are affected by data from both ends of the data vector. By circularly shifting the matrix (13.10.1) $N/2$ columns to the left, one can symmetrize the wrap-around; but this does not eliminate it. It is in fact possible to eliminate the wrap-around completely by altering the coefficients in the first and last N rows of (13.10.1), giving an orthogonal matrix that is purely band-diagonal [5]. This variant, beyond our scope here, is useful when, e.g., the data varies by many orders of magnitude from one end of the data vector to the other.

Here is a routine, `wt1`, that performs the pyramidal algorithm (or its inverse if `isign` is negative) on some data vector `a(1:n)`. Successive applications of the wavelet filter, and accompanying permutations, are done by an assumed routine `wtstep`, which must be provided. (We give examples of several different `wtstep` routines just below.)

```
      SUBROUTINE wt1(a,n,isign,wtstep)
      INTEGER isign,n
      REAL a(n)
      EXTERNAL wtstep
C     USES wtstep
```

One-dimensional discrete wavelet transform. This routine implements the pyramid algorithm, replacing `a(1:n)` by its wavelet transform (for `isign=1`), or performing the inverse operation (for `isign=-1`). Note that `n` MUST be an integer power of 2. The subroutine

wtstep, whose actual name must be supplied in calling this routine, is the underlying wavelet filter. Examples of wtstep are daub4 and (preceded by pwtset) pwt.

```
      INTEGER nn
      if (n.lt.4) return
      if (isign.ge.0) then                    Wavelet transform.
        nn=n                                  Start at largest hierarchy,
1       if (nn.ge.4) then
          call wtstep(a,nn,isign)             and work towards smallest.
          nn=nn/2
        goto 1
        endif
      else                                    Inverse wavelet transform.
        nn=4                                  Start at smallest hierarchy,
2       if (nn.le.n) then
          call wtstep(a,nn,isign)
          nn=nn*2                             and work towards largest.
        goto 2
        endif
      endif
      return
      END
```

Here, as a specific instance of wtstep, is a routine for the DAUB4 wavelets:

```
      SUBROUTINE daub4(a,n,isign)
      INTEGER n,isign,NMAX          NMAX is the maximum allowed value of n.
      REAL a(n),C3,C2,C1,C0
      PARAMETER (C0=0.4829629131445341,C1=0.8365163037378079,
     *    C2=0.2241438680420134,C3=-0.1294095225512604,NMAX=1024)
        Applies the Daubechies 4-coefficient wavelet filter to data vector a(1:n) (for isign=1) or
        applies its transpose (for isign=-1). Used hierarchically by routines wt1 and wtn.
      REAL wksp(NMAX)
      INTEGER nh,nh1,i,j
      if(n.lt.4)return
      if(n.gt.NMAX) pause 'wksp too small in daub4'
      nh=n/2
      nh1=nh+1
      if (isign.ge.0) then                          Apply filter.
        i=1
        do 11 j=1,n-3,2
          wksp(i)=C0*a(j)+C1*a(j+1)+C2*a(j+2)+C3*a(j+3)
          wksp(i+nh)=C3*a(j)-C2*a(j+1)+C1*a(j+2)-C0*a(j+3)
          i=i+1
        enddo 11
        wksp(i)=C0*a(n-1)+C1*a(n)+C2*a(1)+C3*a(2)
        wksp(i+nh)=C3*a(n-1)-C2*a(n)+C1*a(1)-C0*a(2)
      else                                          Apply transpose filter.
        wksp(1)=C2*a(nh)+C1*a(n)+C0*a(1)+C3*a(nh1)
        wksp(2)=C3*a(nh)-C0*a(n)+C1*a(1)-C2*a(nh1)
        j=3
        do 12 i=1,nh-1
          wksp(j)=C2*a(i)+C1*a(i+nh)+C0*a(i+1)+C3*a(i+nh1)
          wksp(j+1)=C3*a(i)-C0*a(i+nh)+C1*a(i+1)-C2*a(i+nh1)
          j=j+2
        enddo 12
      endif
      do 13 i=1,n
        a(i)=wksp(i)
      enddo 13
      return
      END
```

For larger sets of wavelet coefficients, the wrap-around of the last rows or columns is a programming inconvenience. An efficient implementation would handle the wrap-arounds as special cases, outside of the main loop. Here, we will content ourselves with a more general scheme involving some extra arithmetic at run time. The following routine sets up any particular wavelet coefficients whose values you happen to know.

```
      SUBROUTINE pwtset(n)
      INTEGER n,NCMAX,ncof,ioff,joff
      PARAMETER (NCMAX=50)          Maximum number of wavelet coefficients passed to pwt.
      REAL cc(NCMAX),cr(NCMAX)
      COMMON /pwtcom/ cc,cr,ncof,ioff,joff
        Initializing routine for pwt, here implementing the Daubechies wavelet filters with 4, 12,
        and 20 coefficients, as selected by the input value n. Further wavelet filters can be included
        in the obvious manner. This routine must be called (once) before the first use of pwt. (For
        the case n=4, the specific routine daub4 is considerably faster than pwt.)
      INTEGER k
      REAL sig,c4(4),c12(12),c20(20)
      SAVE c4,c12,c20,/pwtcom/
      DATA c4/0.4829629131445341, 0.8365163037378079,
     *     0.2241438680420134,-0.1294095225512604/
      DATA c12 /.111540743350, .494623890398, .751133908021,
     *     .315250351709,-.226264693965,-.129766867567,
     *     .097501605587, .027522865530,-.031582039318,
     *     .000553842201, .004777257511,-.001077301085/
      DATA c20 /.026670057901, .188176800078, .527201188932,
     *     .688459039454, .281172343661,-.249846424327,
     *    -.195946274377, .127369340336, .093057364604,
     *    -.071394147166,-.029457536822, .033212674059,
     *     .003606553567,-.010733175483, .001395351747,
     *     .001992405295,-.000685856695,-.000116466855,
     *     .000093588670,-.000013264203 /
      ncof=n
      sig=-1.
      do 11 k=1,n
        if(n.eq.4)then
          cc(k)=c4(k)
        else if(n.eq.12)then
          cc(k)=c12(k)
        else if(n.eq.20)then
          cc(k)=c20(k)
        else
          pause 'unimplemented value n in pwtset'
        endif
        cr(ncof+1-k)=sig*cc(k)
        sig=-sig
      enddo 11
      ioff=-n/2                  These values center the "support" of the wavelets at each level.
      joff=-n/2                    Alternatively, the "peaks" of the wavelets can be approxi-
      return                       mately centered by the choices ioff=-2 and joff=-n+2.
      END
```

Once `pwtset` has been called, the following routine can be used as a specific instance of `wtstep`.

```
      SUBROUTINE pwt(a,n,isign)
      INTEGER isign,n,NMAX,NCMAX,ncof,ioff,joff
      PARAMETER (NMAX=2048,NCMAX=50)
      REAL a(n),wksp(NMAX),cc(NCMAX),cr(NCMAX)
      COMMON /pwtcom/ cc,cr,ncof,ioff,joff
```

Partial wavelet transform: applies an arbitrary wavelet filter to data vector a(1:n) (for isign=1) or applies its transpose (for isign=-1). Used hierarchically by routines wt1 and wtn. The actual filter is determined by a preceding (and required) call to pwtset, which initializes the common block pwtcom.

```
      INTEGER i,ii,j,jf,jr,k,n1,ni,nj,nh,nmod
      REAL ai,ai1
      if (n.lt.4) return
      nmod=ncof*n                          A positive constant equal to zero mod n.
      n1=n-1                               Mask of all bits, since n a power of 2.
      nh=n/2
      do 11 j=1,n
        wksp(j)=0.
      enddo 11
      if (isign.ge.0) then                 Apply filter.
        ii=1
        do 13 i=1,n,2
          ni=i+nmod+ioff                   Pointer to be incremented and wrapped-around.
          nj=i+nmod+joff
          do 12 k=1,ncof
            jf=iand(n1,ni+k)               We use bitwise and to wrap-around the pointers.
            jr=iand(n1,nj+k)
            wksp(ii)=wksp(ii)+cc(k)*a(jf+1)
            wksp(ii+nh)=wksp(ii+nh)+cr(k)*a(jr+1)
          enddo 12
          ii=ii+1
        enddo 13
      else                                 Apply transpose filter.
        ii=1
        do 15 i=1,n,2
          ai=a(ii)
          ai1=a(ii+nh)
          ni=i+nmod+ioff                   See comments above.
          nj=i+nmod+joff
          do 14 k=1,ncof
            jf=iand(n1,ni+k)+1
            jr=iand(n1,nj+k)+1
            wksp(jf)=wksp(jf)+cc(k)*ai
            wksp(jr)=wksp(jr)+cr(k)*ai1
          enddo 14
          ii=ii+1
        enddo 15
      endif
      do 16 j=1,n                          Copy the results back from workspace.
        a(j)=wksp(j)
      enddo 16
      return
      END
```

What Do Wavelets Look Like?

We are now in a position actually to see some wavelets. To do so, we simply run unit vectors through any of the above discrete wavelet transforms, with isign negative so that the inverse transform is performed. Figure 13.10.1 shows the DAUB4 wavelet that is the inverse DWT of a unit vector in the 5th component of a vector of length 1024, and also the DAUB20 wavelet that is the inverse of the 24th component. (One needs to go to a later hierarchical level for DAUB20, to avoid a wavelet with a wrapped-around tail.) Other unit vectors would give wavelets with the same shapes, but different positions and scales.

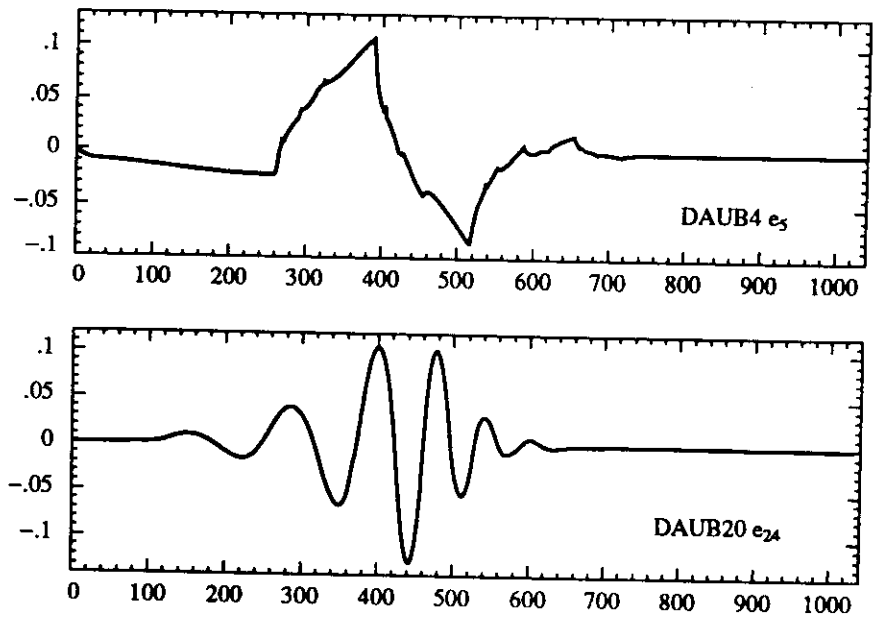


Figure 13.10.1. Wavelet functions, that is, single basis functions from the wavelet families DAUB4 and DAUB20. A complete, orthonormal wavelet basis consists of scalings and translations of either one of these functions. DAUB4 has an infinite number of cusps; DAUB20 would show similar behavior in a higher derivative.

One sees that both DAUB4 and DAUB20 have wavelets that are continuous. DAUB20 wavelets also have higher continuous derivatives. DAUB4 has the peculiar property that its derivative exists only *almost* everywhere. Examples of where it fails to exist are the points $p/2^n$, where p and n are integers; at such points, DAUB4 is left differentiable, but not right differentiable! This kind of discontinuity — at least in some derivative — is a necessary feature of wavelets with compact support, like the Daubechies series. For every increase in the number of wavelet coefficients by two, the Daubechies wavelets gain about *half* a derivative of continuity. (But not exactly half; the actual orders of regularity are irrational numbers!)

Note that the fact that wavelets are not smooth does not prevent their having exact representations for some smooth functions, as demanded by their approximation order p. The continuity of a wavelet is not the same as the continuity of functions that a set of wavelets can represent. For example, DAUB4 can represent (piecewise) linear functions of arbitrary slope: in the correct linear combinations, the cusps all cancel out. Every increase of two in the number of coefficients allows one higher order of polynomial to be exactly represented.

Figure 13.10.2 shows the result of performing the inverse DWT on the input vector $\mathbf{e}_{10} + \mathbf{e}_{58}$, again for the two different particular wavelets. Since 10 lies early in the hierarchical range of $9 - 16$, that wavelet lies on the left side of the picture. Since 58 lies in a later (smaller-scale) hierarchy, it is a narrower wavelet; in the range of 33–64 it is towards the end, so it lies on the right side of the picture. Note that smaller-scale wavelets are taller, so as to have the same squared integral.

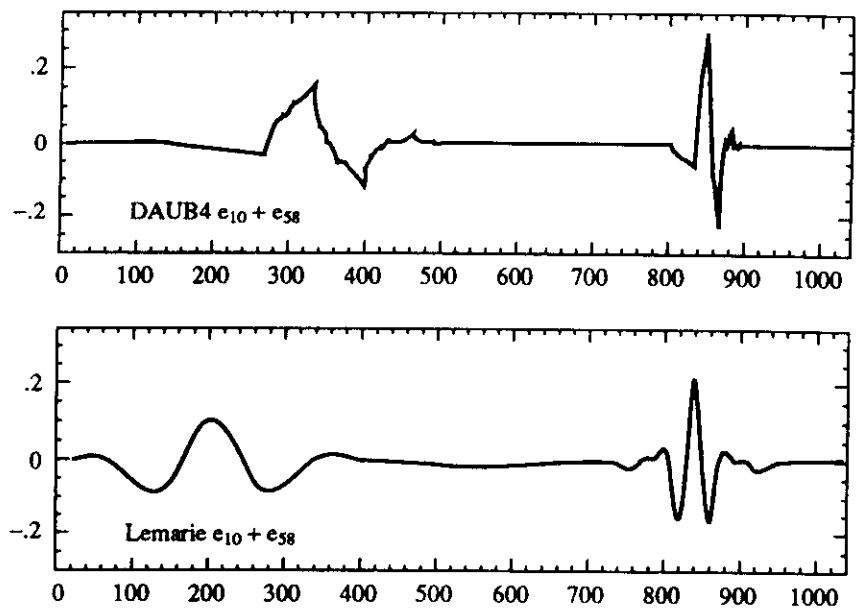


Figure 13.10.2. More wavelets, here generated from the sum of two unit vectors, $e_{10} + e_{58}$, which are in different hierarchical levels of scale, and also at different spatial positions. DAUB4 wavelets (a) are defined by a filter in coordinate space (equation 13.10.5), while Lemarie wavelets (b) are defined by a filter most easily written in Fourier space (equation 13.10.14).

Wavelet Filters in the Fourier Domain

The Fourier transform of a set of filter coefficients c_j is given by

$$H(\omega) = \sum_j c_j e^{ij\omega} \tag{13.10.8}$$

Here H is a function periodic in 2π, and it has the same meaning as before: It is the wavelet filter, now written in the Fourier domain. A very useful fact is that the orthogonality conditions for the c's (e.g., equation 13.10.3 above) collapse to two simple relations in the Fourier domain,

$$\frac{1}{2}|H(0)|^2 = 1 \tag{13.10.9}$$

and

$$\frac{1}{2}\left[|H(\omega)|^2 + |H(\omega+\pi)|^2\right] = 1 \tag{13.10.10}$$

Likewise the approximation condition of order p (e.g., equation 13.10.4 above) has a simple formulation, requiring that $H(\omega)$ have a pth order zero at $\omega = \pi$, or (equivalently)

$$H^{(m)}(\pi) = 0 \qquad m = 0, 1, \ldots, p-1 \tag{13.10.11}$$

It is thus relatively straightforward to invent wavelet sets in the Fourier domain. You simply invent a function $H(\omega)$ satisfying equations (13.10.9)–(13.10.11). To find the actual c_j's applicable to a data (or s-component) vector of length N, and with periodic wrap-around as in matrices (13.10.1) and (13.10.2), you invert equation (13.10.8) by the discrete Fourier transform

$$c_j = \frac{1}{N}\sum_{k=0}^{N-1} H\left(\frac{2\pi k}{N}\right) e^{-2\pi i j k/N} \tag{13.10.12}$$

The quadrature mirror filter G (reversed c_j's with alternating signs), incidentally, has the Fourier representation

$$G(\omega) = e^{-i\omega} H^*(\omega + \pi) \tag{13.10.13}$$

where asterisk denotes complex conjugation.

In general the above procedure will *not* produce wavelet filters with compact support. In other words, all N of the c_j's, $j = 0, \ldots, N-1$ will in general be nonzero (though they may be rapidly decreasing in magnitude). The Daubechies wavelets, or other wavelets with compact support, are specially chosen so that $H(\omega)$ is a trigonometric polynomial with only a small number of Fourier components, guaranteeing that there will be only a small number of nonzero c_j's.

On the other hand, there is sometimes no particular reason to demand compact support. Giving it up in fact allows the ready construction of relatively smoother wavelets (higher values of p). Even without compact support, the convolutions implicit in the matrix (13.10.1) can be done efficiently by FFT methods.

Lemarie's wavelet (see [4]) has $p = 4$, does not have compact support, and is defined by the choice of $H(\omega)$,

$$H(\omega) = \left[2(1-u)^4 \frac{315 - 420u + 126u^2 - 4u^3}{315 - 420v + 126v^2 - 4v^3}\right]^{1/2} \tag{13.10.14}$$

where

$$u \equiv \sin^2\frac{\omega}{2} \qquad v \equiv \sin^2\omega \tag{13.10.15}$$

It is beyond our scope to explain where equation (13.10.14) comes from. An informal description is that the quadrature mirror filter $G(\omega)$ deriving from equation (13.10.14) has the property that it gives identically zero when applied to any function whose odd-numbered samples are equal to the cubic spline interpolation of its even-numbered samples. Since this class of functions includes many very smooth members, it follows that $H(\omega)$ does a good job of truly selecting a function's smooth information content. Sample Lemarie wavelets are shown in Figure 13.10.2.

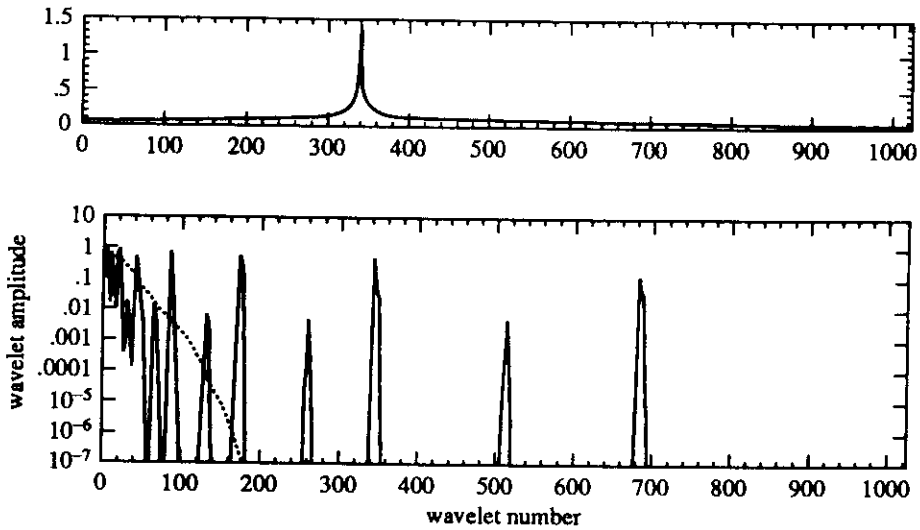


Figure 13.10.3. (a) Arbitrary test function, with cusp, sampled on a vector of length 1024. (b) Absolute value of the 1024 wavelet coefficients produced by the discrete wavelet transform of (a). Note log scale. The dotted curve plots the same amplitudes when sorted by decreasing size. One sees that only 130 out of 1024 coefficients are larger than 10^{-4} (or larger than about 10^{-5} times the largest coefficient, whose value is ~ 10).

Truncated Wavelet Approximations

Most of the usefulness of wavelets rests on the fact that wavelet transforms can usefully be severely truncated, that is, turned into sparse expansions. The case of Fourier transforms is different: FFTs are ordinarily used without truncation, to compute fast convolutions, for example. This works because the convolution operator is particularly simple in the Fourier basis. There are not, however, any standard mathematical operations that are especially simple in the wavelet basis.

To see how truncation works, consider the simple example shown in Figure 13.10.3. The upper panel shows an arbitrarily chosen test function, smooth except for a square-root cusp, sampled onto a vector of length 2^{10}. The bottom panel (solid curve) shows, on a log scale, the absolute value of the vector's components after it has been run through the DAUB4 discrete wavelet transform. One notes, from right to left, the different levels of hierarchy, 513–1024, 257–512, 129–256, etc. Within each level, the wavelet coefficients are non-negligible only very near the location of the cusp, or very near the left and right boundaries of the hierarchical range (edge effects).

The dotted curve in the lower panel of Figure 13.10.3 plots the same amplitudes as the solid curve, but sorted into decreasing order of size. One can read off, for example, that the 130th largest wavelet coefficient has an amplitude less than 10^{-5} of the largest coefficient, whose magnitude is ~ 10 (power or square integral ratio less than 10^{-10}). Thus, the example function can be represented quite accurately by only 130, rather than 1024, coefficients — the remaining ones being set to zero. Note that this kind of truncation makes the vector sparse, but not shorter than 1024. It is very important that vectors in wavelet space be truncated according to the *amplitude* of the components, not their position in the vector. Keeping the first 256

components of the vector (all levels of the hierarchy except the last two) would give an extremely poor, and jagged, approximation to the function. When you compress a function with wavelets, you have to record both the values *and the positions* of the nonzero coefficients.

Generally, compact (and therefore unsmooth) wavelets are better for lower accuracy approximation and for functions with discontinuities (like edges), while smooth (and therefore noncompact) wavelets are better for achieving high numerical accuracy. This makes compact wavelets a good choice for image compression, for example, while it makes smooth wavelets best for fast solution of integral equations.

Wavelet Transform in Multidimensions

A wavelet transform of a d-dimensional array is most easily obtained by transforming the array sequentially on its first index (for all values of its other indices), then on its second, and so on. Each transformation corresponds to multiplication by an orthogonal matrix. By matrix associativity, the result is independent of the order in which the indices were transformed. The situation is exactly like that for multidimensional FFTs. A routine for effecting the multidimensional DWT can thus be modeled on a multidimensional FFT routine like `fourn`:

```
      SUBROUTINE wtn(a,nn,ndim,isign,wtstep)
      INTEGER isign,ndim,nn(ndim),NMAX
      REAL a(*)
      EXTERNAL wtstep
      PARAMETER (NMAX=1024)
C     USES wtstep
        Replaces a by its ndim-dimensional discrete wavelet transform, if isign is input as 1. nn
        is an integer array of length ndim, containing the lengths of each dimension (number of real
        values), which MUST all be powers of 2. a is a real array of length equal to the product
        of these lengths, in which the data are stored as in a multidimensional real FORTRAN array.
        If isign is input as -1, a is replaced by its inverse wavelet transform. The subroutine
        wtstep, whose actual name must be supplied in calling this routine, is the underlying
        wavelet filter. Examples of wtstep are daub4 and (preceded by pwtset) pwt.
      INTEGER i1,i2,i3,idim,k,n,nnew,nprev,nt,ntot
      REAL wksp(NMAX)
      ntot=1
      do 11 idim=1,ndim
        ntot=ntot*nn(idim)
      enddo 11
      nprev=1
      do 16 idim=1,ndim                          Main loop over the dimensions.
        n=nn(idim)
        nnew=n*nprev
        if (n.gt.4) then
          do 15 i2=0,ntot-1,nnew
            do 14 i1=1,nprev
              i3=i1+i2
              do 12 k=1,n                        Copy the relevant row or column or etc. into
                wksp(k)=a(i3)                      workspace.
                i3=i3+nprev
              enddo 12
              if (isign.ge.0) then               Do one-dimensional wavelet transform.
                nt=n
1               if (nt.ge.4) then
                call wtstep(wksp,nt,isign)
                nt=nt/2
                goto 1
                endif
```

```
                else                              Or inverse transform.
                    nt=4
2                   if (nt.le.n) then
                    call wtstep(wksp,nt,isign)
                    nt=nt*2
                    goto 2
                    endif
                endif
                i3=i1+i2
                do 13 k=1,n                       Copy back from workspace.
                    a(i3)=wksp(k)
                    i3=i3+nprev
                enddo 13
            enddo 14
        enddo 15
    endif
    nprev=nnew
enddo 16
return
END
```

Here, as before, `wtstep` is an individual wavelet step, either `daub4` or `pwt`.

Compression of Images

An immediate application of the multidimensional transform `wtn` is to image compression. The overall procedure is to take the wavelet transform of a digitized image, and then to "allocate bits" among the wavelet coefficients in some highly nonuniform, optimized, manner. In general, large wavelet coefficients get quantized accurately, while small coefficients are quantized coarsely with only a bit or two — or else are truncated completely. If the resulting quantization levels are still statistically nonuniform, they may then be further compressed by a technique like Huffman coding (§20.4).

While a more detailed description of the "back end" of this process, namely the quantization and coding of the image, is beyond our scope, it is quite straightforward to demonstrate the "front-end" wavelet encoding with a simple truncation: We keep (with full accuracy) all wavelet coefficients larger than some threshold, and we delete (set to zero) all smaller wavelet coefficients. We can then adjust the threshold to vary the fraction of preserved coefficients.

Figure 13.10.4 shows a sequence of images that differ in the number of wavelet coefficients that have been kept. The original picture (a), which is an official IEEE test image, has 256 by 256 pixels with an 8-bit grayscale. The two reproductions following are reconstructed with 23% (b) and 5.5% (c) of the 65536 wavelet coefficients. The latter image illustrates the kind of compromises made by the truncated wavelet representation. High-contrast edges (the model's right cheek and hair highlights, e.g.) are maintained at a relatively high resolution, while low-contrast areas (the model's left eye and cheek, e.g.) are washed out into what amounts to large constant pixels. Figure 13.10.4 (d) is the result of performing the identical procedure with Fourier, instead of wavelet, transforms: The figure is reconstructed from the 5.5% of 65536 real Fourier components having the largest magnitudes. One sees that, since sines and cosines are nonlocal, the resolution is uniformly poor across the picture; also, the deletion of any components produces a mottled "ringing" everywhere. (Practical Fourier image compression schemes therefore break up an

Figure 13.10.4. (a) IEEE test image, 256 x 256 pixels with 8-bit grayscale. (b) The image is transformed into the wavelet basis; 77% of its wavelet components are set to zero (those of smallest magnitude); it is then reconstructed from the remaining 23%. (c) Same as (b), but 94.5% of the wavelet components are deleted. (d) Same as (c), but the Fourier transform is used instead of the wavelet transform. Wavelet coefficients are better than the Fourier coefficients at preserving relevant details.

image into small blocks of pixels, 16 × 16, say, and do rather elaborate smoothing across block boundaries when the image is reconstructed.)

Fast Solution of Linear Systems

One of the most interesting, and promising, wavelet applications is linear algebra. The basic idea [1] is to think of an integral operator (that is, a large matrix) as a digital image. Suppose that the operator compresses well under a two-dimensional wavelet transform, i.e., that a large fraction of its wavelet coefficients are so small as to be negligible. Then any linear system involving the operator

becomes a sparse system in the wavelet basis. In other words, to solve

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \tag{13.10.16}$$

we first wavelet-transform the operator $\mathbf{A}$ and the right-hand side $\mathbf{b}$ by

$$\tilde{\mathbf{A}} \equiv \mathbf{W} \cdot \mathbf{A} \cdot \mathbf{W}^T, \qquad \tilde{\mathbf{b}} \equiv \mathbf{W} \cdot \mathbf{b} \tag{13.10.17}$$

where $\mathbf{W}$ represents the one-dimensional wavelet transform, then solve

$$\tilde{\mathbf{A}} \cdot \tilde{\mathbf{x}} = \tilde{\mathbf{b}} \tag{13.10.18}$$

and finally transform to the answer by the inverse wavelet transform

$$\mathbf{x} = \mathbf{W}^T \cdot \tilde{\mathbf{x}} \tag{13.10.19}$$

(Note that the routine `wtn` does the complete transformation of $\mathbf{A}$ into $\tilde{\mathbf{A}}$.)

A typical integral operator that compresses well into wavelets has arbitrary (or even nearly singular) elements near to its main diagonal, but becomes smooth away from the diagonal. An example might be

$$A_{ij} = \begin{cases} -1 & \text{if } i = j \\ |i-j|^{-1/2} & \text{otherwise} \end{cases} \tag{13.10.20}$$

Figure 13.10.5 shows a graphical representation of the wavelet transform of this matrix, where i and j range over $1 \ldots 256$, using the DAUB12 wavelets. Elements larger in magnitude than 10^{-3} times the maximum element are shown as black pixels, while elements between 10^{-3} and 10^{-6} are shown in gray. White pixels are $< 10^{-6}$. The indices i and j each number from the lower left.

In the figure, one sees the hierarchical decomposition into power-of-two sized blocks. At the edges or corners of the various blocks, one sees edge effects caused by the wrap-around wavelet boundary conditions. Apart from edge effects, within each block, the nonnegligible elements are concentrated along the block diagonals. This is a statement that, for this type of linear operator, a wavelet is coupled mainly to near neighbors in its own hierarchy (square blocks along the main diagonal) and near neighbors in other hierarchies (rectangular blocks off the diagonal).

The number of nonnegligible elements in a matrix like that in Figure 13.10.5 scales only as N, the linear size of the matrix; as a rough rule of thumb it is about $10N \log_{10}(1/\epsilon)$, where ϵ is the truncation level, e.g., 10^{-6}. For a 2000 by 2000 matrix, then, the matrix is sparse by a factor on the order of 30.

Various numerical schemes can be used to solve sparse linear systems of this "hierarchically band diagonal" form. Beylkin, Coifman, and Rokhlin [1] make the interesting observations that (1) the product of two such matrices is itself hierarchically band diagonal (truncating, of course, newly generated elements that are smaller than the predetermined threshold ϵ); and moreover that (2) the product can be formed in order N operations.

Fast matrix multiplication makes it possible to find the matrix inverse by Schultz's (or Hotelling's) method, see §2.5.

Figure 13.10.5. Wavelet transform of a 256×256 matrix, represented graphically. The original matrix has a discontinuous cusp along its diagonal, decaying smoothly away on both sides of the diagonal. In wavelet basis, the matrix becomes sparse: Components larger than 10^{-3} are shown as black, components larger than 10^{-6} as gray, and smaller-magnitude components are white. The matrix indices i and j number from the lower left.

Other schemes are also possible for fast solution of hierarchically band diagonal forms. For example, one can use the conjugate gradient method, implemented in §2.7 as `linbcg`.

CITED REFERENCES AND FURTHER READING:

Daubechies, I. 1992, *Wavelets* (Philadelphia: S.I.A.M.).

Strang, G. 1989, *SIAM Review*, vol. 31, pp. 614–627.

Beylkin, G., Coifman, R., and Rokhlin, V. 1991, *Communications on Pure and Applied Mathematics*, vol. 44, pp. 141–183. [1]

Daubechies, I. 1988, *Communications on Pure and Applied Mathematics*, vol. 41, pp. 909–996. [2]

Vaidyanathan, P.P. 1990, *Proceedings of the IEEE*, vol. 78, pp. 56–93. [3]

Mallat, S.G. 1989, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, pp. 674–693. [4]

Freedman, M.H., and Press, W.H. 1992, preprint. [5]