

NUMERICAL RECIPES in C

The Art of Scientific Computing

Second Edition

William H. Press

Saul A. Teukolsky

William T. Vetterling

Brian P. Flannery

AMAZON-1024

7,280,599

Numerical Recipes in C

Second Edition

Numerical Recipes in C

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



CAMBRIDGE
UNIVERSITY PRESS

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, Melbourne 3166, Australia

Copyright © Cambridge University Press 1988, 1992
except for §13.10 and Appendix B, which are placed into the public domain,
and except for all other computer programs and procedures, which are
Copyright © Numerical Recipes Software 1987, 1988, 1992
All Rights Reserved.

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 1988; Second Edition originally published 1992.

Reprinted with corrections 1993.

Reprinted with corrections 1994.

Reprinted with corrections 1995.

Reprinted 1996

This reprinting is corrected to software version 2.06.

Printed in the United States of America

Typeset in $\text{\TeX}$

The computer programs in this book are available, in C, in several machine-readable formats. There are also versions of this book and its software available in the Pascal, FORTRAN, and BASIC programming languages.

To purchase diskettes in IBM PC or Apple Macintosh formats, use the order form at the back of the book or write to Cambridge University Press, 110 Midland Avenue, Port Chester, NY 10573. Also available from Cambridge University Press are the *Numerical Recipes Example Books* and coordinated diskettes in C, FORTRAN, Pascal, and BASIC. These provide demonstration programs that illustrate the use of each subroutine and procedure in this book. They too may be ordered in the above manner.

Unlicensed transfer of Numerical Recipes programs from the abovementioned IBM PC or Apple Macintosh diskettes to any other format, or to any computer except a single IBM PC or Apple Macintosh or compatible for each diskette purchased, is strictly prohibited. Licenses for authorized transfers to other computers are available from Numerical Recipes Software, P.O. Box 243, Cambridge, MA 02238 (fax 617 863-1739, email info@nr.com, or <http://world.std.com/~nr>). Technical questions, corrections, and requests for information on other available formats should also be directed to this address.

Library of Congress Cataloging in Publication Data

Numerical recipes in C : the art of scientific computing / William H. Press

... [et al.]. – 2nd ed.

Includes bibliographical references (p.) and index.

ISBN 0-521-43108-5

1. Numerical analysis—Computer programs. 2. Science—Mathematics—Computer programs.

3. C (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 43108 5 Book

ISBN 0 521 43720 2 Example book in C

ISBN 0 521 43714 8 C diskette (IBM 5.25", 1.2M)

ISBN 0 521 43724 5 C diskette (IBM 3.5", 1.44M)

ISBN 0 521 43715 6 C diskette (Mac 3.5", 800K)

Contents

<i>Preface to the Second Edition</i>	<i>xi</i>
<i>Preface to the First Edition</i>	<i>xiv</i>
<i>Legal Matters</i>	<i>xvi</i>
<i>Computer Programs by Chapter and Section</i>	<i>xix</i>
1 Preliminaries	1
1.0 Introduction	1
1.1 Program Organization and Control Structures	5
1.2 Some C Conventions for Scientific Computing	15
1.3 Error, Accuracy, and Stability	28
2 Solution of Linear Algebraic Equations	32
2.0 Introduction	32
2.1 Gauss-Jordan Elimination	36
2.2 Gaussian Elimination with Backsubstitution	41
2.3 LU Decomposition and Its Applications	43
2.4 Tridiagonal and Band Diagonal Systems of Equations	50
2.5 Iterative Improvement of a Solution to Linear Equations	55
2.6 Singular Value Decomposition	59
2.7 Sparse Linear Systems	71
2.8 Vandermonde Matrices and Toeplitz Matrices	90
2.9 Cholesky Decomposition	96
2.10 QR Decomposition	98
2.11 Is Matrix Inversion an N^3 Process?	102
3 Interpolation and Extrapolation	105
3.0 Introduction	105
3.1 Polynomial Interpolation and Extrapolation	108
3.2 Rational Function Interpolation and Extrapolation	111
3.3 Cubic Spline Interpolation	113
3.4 How to Search an Ordered Table	117
3.5 Coefficients of the Interpolating Polynomial	120
3.6 Interpolation in Two or More Dimensions	123

4	<i>Integration of Functions</i>	129
4.0	Introduction	129
4.1	Classical Formulas for Equally Spaced Abscissas	130
4.2	Elementary Algorithms	136
4.3	Romberg Integration	140
4.4	Improper Integrals	141
4.5	Gaussian Quadratures and Orthogonal Polynomials	147
4.6	Multidimensional Integrals	161
5	<i>Evaluation of Functions</i>	165
5.0	Introduction	165
5.1	Series and Their Convergence	165
5.2	Evaluation of Continued Fractions	169
5.3	Polynomials and Rational Functions	173
5.4	Complex Arithmetic	176
5.5	Recurrence Relations and Clenshaw's Recurrence Formula	178
5.6	Quadratic and Cubic Equations	183
5.7	Numerical Derivatives	186
5.8	Chebyshev Approximation	190
5.9	Derivatives or Integrals of a Chebyshev-approximated Function	195
5.10	Polynomial Approximation from Chebyshev Coefficients	197
5.11	Economization of Power Series	198
5.12	Padé Approximants	200
5.13	Rational Chebyshev Approximation	204
5.14	Evaluation of Functions by Path Integration	208
6	<i>Special Functions</i>	212
6.0	Introduction	212
6.1	Gamma Function, Beta Function, Factorials, Binomial Coefficients	213
6.2	Incomplete Gamma Function, Error Function, Chi-Square Probability Function, Cumulative Poisson Function	216
6.3	Exponential Integrals	222
6.4	Incomplete Beta Function, Student's Distribution, F-Distribution, Cumulative Binomial Distribution	226
6.5	Bessel Functions of Integer Order	230
6.6	Modified Bessel Functions of Integer Order	236
6.7	Bessel Functions of Fractional Order, Airy Functions, Spherical Bessel Functions	240
6.8	Spherical Harmonics	252
6.9	Fresnel Integrals, Cosine and Sine Integrals	255
6.10	Dawson's Integral	259
6.11	Elliptic Integrals and Jacobian Elliptic Functions	261
6.12	Hypergeometric Functions	271
7	<i>Random Numbers</i>	274
7.0	Introduction	274
7.1	Uniform Deviates	275

7.2 Transformation Method: Exponential and Normal Deviates	287
7.3 Rejection Method: Gamma, Poisson, Binomial Deviates	290
7.4 Generation of Random Bits	296
7.5 Random Sequences Based on Data Encryption	300
7.6 Simple Monte Carlo Integration	304
7.7 Quasi- (that is, Sub-) Random Sequences	309
7.8 Adaptive and Recursive Monte Carlo Methods	316
8 Sorting	329
8.0 Introduction	329
8.1 Straight Insertion and Shell's Method	330
8.2 Quicksort	332
8.3 Heapsort	336
8.4 Indexing and Ranking	338
8.5 Selecting the M th Largest	341
8.6 Determination of Equivalence Classes	345
9 Root Finding and Nonlinear Sets of Equations	347
9.0 Introduction	347
9.1 Bracketing and Bisection	350
9.2 Secant Method, False Position Method, and Ridders' Method	354
9.3 Van Wijngaarden–Dekker–Brent Method	359
9.4 Newton-Raphson Method Using Derivative	362
9.5 Roots of Polynomials	369
9.6 Newton-Raphson Method for Nonlinear Systems of Equations	379
9.7 Globally Convergent Methods for Nonlinear Systems of Equations	383
10 Minimization or Maximization of Functions	394
10.0 Introduction	394
10.1 Golden Section Search in One Dimension	397
10.2 Parabolic Interpolation and Brent's Method in One Dimension	402
10.3 One-Dimensional Search with First Derivatives	405
10.4 Downhill Simplex Method in Multidimensions	408
10.5 Direction Set (Powell's) Methods in Multidimensions	412
10.6 Conjugate Gradient Methods in Multidimensions	420
10.7 Variable Metric Methods in Multidimensions	425
10.8 Linear Programming and the Simplex Method	430
10.9 Simulated Annealing Methods	444
11 Eigensystems	456
11.0 Introduction	456
11.1 Jacobi Transformations of a Symmetric Matrix	463
11.2 Reduction of a Symmetric Matrix to Tridiagonal Form: Givens and Householder Reductions	469
11.3 Eigenvalues and Eigenvectors of a Tridiagonal Matrix	475
11.4 Hermitian Matrices	481
11.5 Reduction of a General Matrix to Hessenberg Form	482

11.6 The QR Algorithm for Real Hessenberg Matrices	486
11.7 Improving Eigenvalues and/or Finding Eigenvectors by Inverse Iteration	493
12 Fast Fourier Transform	496
12.0 Introduction	496
12.1 Fourier Transform of Discretely Sampled Data	500
12.2 Fast Fourier Transform (FFT)	504
12.3 FFT of Real Functions, Sine and Cosine Transforms	510
12.4 FFT in Two or More Dimensions	521
12.5 Fourier Transforms of Real Data in Two and Three Dimensions	525
12.6 External Storage or Memory-Local FFTs	532
13 Fourier and Spectral Applications	537
13.0 Introduction	537
13.1 Convolution and Deconvolution Using the FFT	538
13.2 Correlation and Autocorrelation Using the FFT	545
13.3 Optimal (Wiener) Filtering with the FFT	547
13.4 Power Spectrum Estimation Using the FFT	549
13.5 Digital Filtering in the Time Domain	558
13.6 Linear Prediction and Linear Predictive Coding	564
13.7 Power Spectrum Estimation by the Maximum Entropy (All Poles) Method	572
13.8 Spectral Analysis of Unevenly Sampled Data	575
13.9 Computing Fourier Integrals Using the FFT	584
13.10 Wavelet Transforms	591
13.11 Numerical Use of the Sampling Theorem	606
14 Statistical Description of Data	609
14.0 Introduction	609
14.1 Moments of a Distribution: Mean, Variance, Skewness, and So Forth	610
14.2 Do Two Distributions Have the Same Means or Variances?	615
14.3 Are Two Distributions Different?	620
14.4 Contingency Table Analysis of Two Distributions	628
14.5 Linear Correlation	636
14.6 Nonparametric or Rank Correlation	639
14.7 Do Two-Dimensional Distributions Differ?	645
14.8 Savitzky-Golay Smoothing Filters	650
15 Modeling of Data	656
15.0 Introduction	656
15.1 Least Squares as a Maximum Likelihood Estimator	657
15.2 Fitting Data to a Straight Line	661
15.3 Straight-Line Data with Errors in Both Coordinates	666
15.4 General Linear Least Squares	671
15.5 Nonlinear Models	681

15.6 Confidence Limits on Estimated Model Parameters	689
15.7 Robust Estimation	699
16 Integration of Ordinary Differential Equations	707
16.0 Introduction	707
16.1 Runge-Kutta Method	710
16.2 Adaptive Stepsize Control for Runge-Kutta	714
16.3 Modified Midpoint Method	722
16.4 Richardson Extrapolation and the Bulirsch-Stoer Method	724
16.5 Second-Order Conservative Equations	732
16.6 Stiff Sets of Equations	734
16.7 Multistep, Multivalued, and Predictor-Corrector Methods	747
17 Two Point Boundary Value Problems	753
17.0 Introduction	753
17.1 The Shooting Method	757
17.2 Shooting to a Fitting Point	760
17.3 Relaxation Methods	762
17.4 A Worked Example: Spheroidal Harmonics	772
17.5 Automated Allocation of Mesh Points	783
17.6 Handling Internal Boundary Conditions or Singular Points	784
18 Integral Equations and Inverse Theory	788
18.0 Introduction	788
18.1 Fredholm Equations of the Second Kind	791
18.2 Volterra Equations	794
18.3 Integral Equations with Singular Kernels	797
18.4 Inverse Problems and the Use of A Priori Information	804
18.5 Linear Regularization Methods	808
18.6 Backus-Gilbert Method	815
18.7 Maximum Entropy Image Restoration	818
19 Partial Differential Equations	827
19.0 Introduction	827
19.1 Flux-Conservative Initial Value Problems	834
19.2 Diffusive Initial Value Problems	847
19.3 Initial Value Problems in Multidimensions	853
19.4 Fourier and Cyclic Reduction Methods for Boundary Value Problems	857
19.5 Relaxation Methods for Boundary Value Problems	863
19.6 Multigrid Methods for Boundary Value Problems	871
20 Less-Numerical Algorithms	889
20.0 Introduction	889
20.1 Diagnosing Machine Parameters	889
20.2 Gray Codes	894

20.3 Cyclic Redundancy and Other Checksums	896
20.4 Huffman Coding and Compression of Data	903
20.5 Arithmetic Coding	910
20.6 Arithmetic at Arbitrary Precision	915
<i>References</i>	926
<i>Appendix A: Table of Prototype Declarations</i>	930
<i>Appendix B: Utility Routines</i>	940
<i>Appendix C: Complex Arithmetic</i>	948
<i>Index of Programs and Dependencies</i>	951
<i>General Index</i>	965

Chapter 3. Interpolation and Extrapolation

3.0 Introduction

We sometimes know the value of a function $f(x)$ at a set of points $x_1, x_2, \dots, x_N$ (say, with $x_1 < \dots < x_N$), but we don't have an analytic expression for $f(x)$ that lets us calculate its value at an arbitrary point. For example, the $f(x_i)$'s might result from some physical measurement or from long numerical calculation that cannot be cast into a simple functional form. Often the x_i 's are equally spaced, but not necessarily.

The task now is to estimate $f(x)$ for arbitrary x by, in some sense, drawing a smooth curve through (and perhaps beyond) the x_i . If the desired x is in between the largest and smallest of the x_i 's, the problem is called *interpolation*; if x is outside that range, it is called *extrapolation*, which is considerably more hazardous (as many former stock-market analysts can attest).

Interpolation and extrapolation schemes must model the function, between or beyond the known points, by some plausible functional form. The form should be sufficiently general so as to be able to approximate large classes of functions which might arise in practice. By far most common among the functional forms used are polynomials (§3.1). Rational functions (quotients of polynomials) also turn out to be extremely useful (§3.2). Trigonometric functions, sines and cosines, give rise to *trigonometric interpolation* and related Fourier methods, which we defer to Chapters 12 and 13.

There is an extensive mathematical literature devoted to theorems about what sort of functions can be well approximated by which interpolating functions. These theorems are, alas, almost completely useless in day-to-day work: If we know enough about our function to apply a theorem of any power, we are usually not in the pitiful state of having to interpolate on a table of its values!

Interpolation is related to, but distinct from, *function approximation*. That task consists of finding an approximate (but easily computable) function to use in place of a more complicated one. In the case of interpolation, you are given the function f at points *not of your own choosing*. For the case of function approximation, you are allowed to compute the function f at *any* desired points for the purpose of developing your approximation. We deal with function approximation in Chapter 5.

One can easily find pathological functions that make a mockery of any interpolation scheme. Consider, for example, the function

$$f(x) = 3x^2 + \frac{1}{\pi^4} \ln [(\pi - x)^2] + 1 \quad (3.0.1)$$

which is well-behaved everywhere except at $x = \pi$, very mildly singular at $x = \pi$, and otherwise takes on all positive and negative values. Any interpolation based on the values $x = 3.13, 3.14, 3.15, 3.16$, will assuredly get a very wrong answer for the value $x = 3.1416$, even though a graph plotting those five points looks really quite smooth! (Try it on your calculator.)

Because pathologies can lurk anywhere, it is highly desirable that an interpolation and extrapolation routine should provide an estimate of its own error. Such an error estimate can never be foolproof, of course. We could have a function that, for reasons known only to its maker, takes off wildly and unexpectedly between two tabulated points. Interpolation always presumes some degree of smoothness for the function interpolated, but within this framework of presumption, deviations from smoothness can be detected.

Conceptually, the interpolation process has two stages: (1) Fit an interpolating function to the data points provided. (2) Evaluate that interpolating function at the target point x .

However, this two-stage method is generally not the best way to proceed in practice. Typically it is computationally less efficient, and more susceptible to roundoff error, than methods which construct a functional estimate $f(x)$ directly from the N tabulated values every time one is desired. Most practical schemes start at a nearby point $f(x_i)$, then add a sequence of (hopefully) decreasing corrections, as information from other $f(x_i)$'s is incorporated. The procedure typically takes $O(N^2)$ operations. If everything is well behaved, the last correction will be the smallest, and it can be used as an informal (though not rigorous) bound on the error.

In the case of polynomial interpolation, it sometimes does happen that the coefficients of the interpolating polynomial are of interest, even though their use in *evaluating* the interpolating function should be frowned on. We deal with this eventuality in §3.5.

Local interpolation, using a finite number of “nearest-neighbor” points, gives interpolated values $f(x)$ that do not, in general, have continuous first or higher derivatives. That happens because, as x crosses the tabulated values x_i , the interpolation scheme switches which tabulated points are the “local” ones. (If such a switch is allowed to occur anywhere *else*, then there will be a discontinuity in the interpolated function itself at that point. Bad idea!)

In situations where continuity of derivatives is a concern, one must use the “stiffer” interpolation provided by a so-called *spline* function. A spline is a polynomial between each pair of table points, but one whose coefficients are determined “slightly” nonlocally. The nonlocality is designed to guarantee global smoothness in the interpolated function up to some order of derivative. Cubic splines (§3.3) are the most popular. They produce an interpolated function that is continuous through the second derivative. Splines tend to be stabler than polynomials, with less possibility of wild oscillation between the tabulated points.

The number of points (minus one) used in an interpolation scheme is called the *order* of the interpolation. Increasing the order does not necessarily increase the accuracy, especially in polynomial interpolation. If the added points are distant from the point of interest x , the resulting higher-order polynomial, with its additional constrained points, tends to oscillate wildly between the tabulated values. This oscillation may have no relation at all to the behavior of the “true” function (see Figure 3.0.1). Of course, adding points *close* to the desired point usually does help,

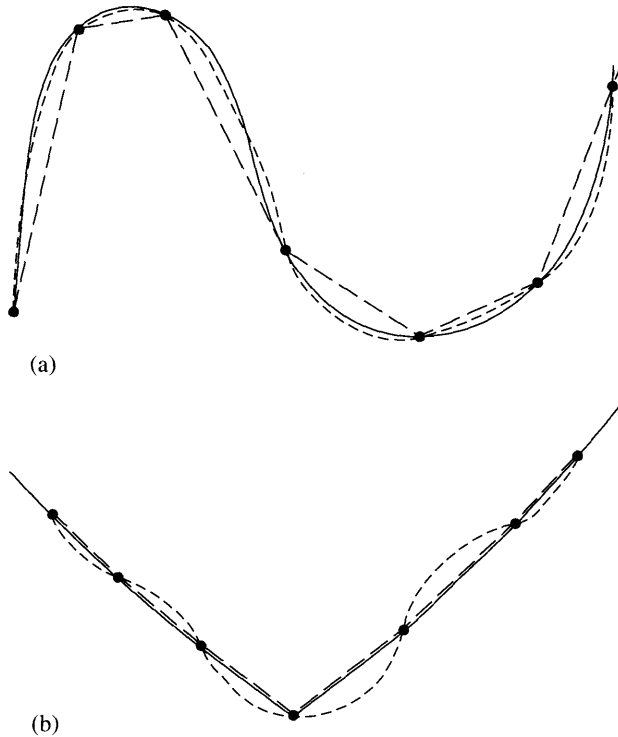


Figure 3.0.1. (a) A smooth function (solid line) is more accurately interpolated by a high-order polynomial (shown schematically as dotted line) than by a low-order polynomial (shown as a piecewise linear dashed line). (b) A function with sharp corners or rapidly changing higher derivatives is *less* accurately approximated by a high-order polynomial (dotted line), which is too “stiff,” than by a low-order polynomial (dashed lines). Even some smooth functions, such as exponentials or rational functions, can be badly approximated by high-order polynomials.

but a finer mesh implies a larger table of values, not always available.

Unless there is solid evidence that the interpolating function is close in form to the true function f , it is a good idea to be cautious about high-order interpolation. We enthusiastically endorse interpolations with 3 or 4 points, we are perhaps tolerant of 5 or 6; but we rarely go higher than that unless there is quite rigorous monitoring of estimated errors.

When your table of values contains many more points than the desirable order of interpolation, you must begin each interpolation with a search for the right “local” place in the table. While not strictly a part of the subject of interpolation, this task is important enough (and often enough botched) that we devote §3.4 to its discussion.

The routines given for interpolation are also routines for extrapolation. An important application, in Chapter 16, is their use in the integration of ordinary differential equations. There, considerable care *is* taken with the monitoring of errors. Otherwise, the dangers of extrapolation cannot be overemphasized: An interpolating function, which is performe an extrapolating function, will typically go berserk when the argument x is outside the range of tabulated values by more than the typical spacing of tabulated points.

Interpolation can be done in more than one dimension, e.g., for a function

$f(x, y, z)$. Multidimensional interpolation is often accomplished by a sequence of one-dimensional interpolations. We discuss this in §3.6.

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), §25.2.
- Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), Chapter 2.
- Acton, F.S. 1970, *Numerical Methods That Work*; 1990, corrected edition (Washington: Mathematical Association of America), Chapter 3.
- Kahaner, D., Moler, C., and Nash, S. 1989, *Numerical Methods and Software* (Englewood Cliffs, NJ: Prentice Hall), Chapter 4.
- Johnson, L.W., and Riess, R.D. 1982, *Numerical Analysis*, 2nd ed. (Reading, MA: Addison-Wesley), Chapter 5.
- Ralston, A., and Rabinowitz, P. 1978, *A First Course in Numerical Analysis*, 2nd ed. (New York: McGraw-Hill), Chapter 3.
- Isaacson, E., and Keller, H.B. 1966, *Analysis of Numerical Methods* (New York: Wiley), Chapter 6.

3.1 Polynomial Interpolation and Extrapolation

Through any two points there is a unique line. Through any three points, a unique quadratic. Et cetera. The interpolating polynomial of degree $N - 1$ through the N points $y_1 = f(x_1), y_2 = f(x_2), \dots, y_N = f(x_N)$ is given explicitly by Lagrange's classical formula,

$$P(x) = \frac{(x - x_2)(x - x_3) \dots (x - x_N)}{(x_1 - x_2)(x_1 - x_3) \dots (x_1 - x_N)} y_1 + \frac{(x - x_1)(x - x_3) \dots (x - x_N)}{(x_2 - x_1)(x_2 - x_3) \dots (x_2 - x_N)} y_2 \\ + \dots + \frac{(x - x_1)(x - x_2) \dots (x - x_{N-1})}{(x_N - x_1)(x_N - x_2) \dots (x_N - x_{N-1})} y_N \quad (3.1.1)$$

There are N terms, each a polynomial of degree $N - 1$ and each constructed to be zero at all of the x_i except one, at which it is constructed to be y_i .

It is not terribly wrong to implement the Lagrange formula straightforwardly, but it is not terribly right either. The resulting algorithm gives no error estimate, and it is also somewhat awkward to program. A much better algorithm (for constructing the same, unique, interpolating polynomial) is *Neville's algorithm*, closely related to and sometimes confused with *Aitken's algorithm*, the latter now considered obsolete.

Let P_1 be the value at x of the unique polynomial of degree zero (i.e., a constant) passing through the point (x_1, y_1) ; so $P_1 = y_1$. Likewise define $P_2, P_3, \dots, P_N$. Now let P_{12} be the value at x of the unique polynomial of degree one passing through both (x_1, y_1) and (x_2, y_2) . Likewise $P_{23}, P_{34}, \dots, P_{(N-1)N}$. Similarly, for higher-order polynomials, up to $P_{123 \dots N}$, which is the value of the unique interpolating polynomial through all N points, i.e., the desired answer.

The various P 's form a "tableau" with "ancestors" on the left leading to a single "descendant" at the extreme right. For example, with $N = 4$,

$$\begin{array}{rclcl}
 x_1 : & y_1 = P_1 & & & \\
 & & P_{12} & & \\
 x_2 : & y_2 = P_2 & & P_{123} & \\
 & & P_{23} & & P_{1234} \\
 x_3 : & y_3 = P_3 & & P_{234} & \\
 & & P_{34} & & \\
 x_4 : & y_4 = P_4 & & &
 \end{array} \tag{3.1.2}$$

Neville's algorithm is a recursive way of filling in the numbers in the tableau a column at a time, from left to right. It is based on the relationship between a "daughter" P and its two "parents,"

$$P_{i(i+1)\dots(i+m)} = \frac{(x - x_{i+m})P_{i(i+1)\dots(i+m-1)} + (x_i - x)P_{(i+1)(i+2)\dots(i+m)}}{x_i - x_{i+m}} \tag{3.1.3}$$

This recurrence works because the two parents already agree at points $x_{i+1} \dots x_{i+m-1}$.

An improvement on the recurrence (3.1.3) is to keep track of the small *differences* between parents and daughters, namely to define (for $m = 1, 2, \dots, N - 1$),

$$\begin{aligned}
 C_{m,i} &\equiv P_{i\dots(i+m)} - P_{i\dots(i+m-1)} \\
 D_{m,i} &\equiv P_{i\dots(i+m)} - P_{(i+1)\dots(i+m)}.
 \end{aligned} \tag{3.1.4}$$

Then one can easily derive from (3.1.3) the relations

$$\begin{aligned}
 D_{m+1,i} &= \frac{(x_{i+m+1} - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}} \\
 C_{m+1,i} &= \frac{(x_i - x)(C_{m,i+1} - D_{m,i})}{x_i - x_{i+m+1}}
 \end{aligned} \tag{3.1.5}$$

At each level m , the C 's and D 's are the corrections that make the interpolation one order higher. The final answer $P_{1\dots N}$ is equal to the sum of *any* y_i plus a set of C 's and/or D 's that form a path through the family tree to the rightmost daughter.

Here is a routine for polynomial interpolation or extrapolation from N input points. Note that the input arrays are assumed to be unit-offset. If you have zero-offset arrays, remember to subtract 1 (see §1.2):

```
#include <math.h>
#include "nrutil.h"

void polint(float xa[], float ya[], int n, float x, float *y, float *dy)
Given arrays xa[1..n] and ya[1..n], and given a value x, this routine returns a value y, and
an error estimate dy. If P(x) is the polynomial of degree N - 1 such that P(xa_i) = ya_i, i =
1, ..., n, then the returned value y = P(x).
{
    int i,m,ns=1;
    float den,dif,dift,ho,hp,w;
```

```

float *c,*d;

dif=fabs(x-xa[1]);
c=vector(1,n);
d=vector(1,n);
for (i=1;i<=n;i++) {           Here we find the index ns of the closest table entry,
    if ( (dift=fabs(x-xa[i])) < dif) {
        ns=i;
        dif=dift;
    }
    c[i]=ya[i];                 and initialize the tableau of c's and d's.
    d[i]=ya[i];
}
*y=ya[ns--];                   This is the initial approximation to y.
for (m=1;m<n;m++) {           For each column of the tableau,
    for (i=1;i<=n-m;i++) {     we loop over the current c's and d's and update
        ho=xa[i]-x;             them.
        hp=xa[i+m]-x;
        w=c[i+1]-d[i];
        if ( (den=ho-hp) == 0.0) nrerror("Error in routine polint");
        This error can occur only if two input xa's are (to within roundoff) identical.
        den=w/den;
        d[i]=hp*den;            Here the c's and d's are updated.
        c[i]=ho*den;
    }
    *y += (*dy=(2*ns < (n-m) ? c[ns+1] : d[ns--]));
    After each column in the tableau is completed, we decide which correction, c or d,
    we want to add to our accumulating value of y, i.e., which path to take through the
    tableau—forking up or down. We do this in such a way as to take the most "straight
    line" route through the tableau to its apex, updating ns accordingly to keep track of
    where we are. This route keeps the partial approximations centered (insofar as possible)
    on the target x. The last dy added is thus the error indication.
}
free_vector(d,1,n);
free_vector(c,1,n);
}

```

Quite often you will want to call `polint` with the dummy arguments `xa` and `ya` replaced by actual arrays *with offsets*. For example, the construction `polint(&xx[14], &yy[14], 4, x, y, dy)` performs 4-point interpolation on the tabulated values `xx[15..18]`, `yy[15..18]`. For more on this, see the end of §3.4.

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), §25.2.
- Stoer, J., and Bulirsch, R. 1980, *Introduction to Numerical Analysis* (New York: Springer-Verlag), §2.1.
- Gear, C.W. 1971, *Numerical Initial Value Problems in Ordinary Differential Equations* (Englewood Cliffs, NJ: Prentice-Hall), §6.1.

```

    }
    den += sumw;
}
den *= m4;
for (j=1; j<=m; j++) p[j] /= den;
free_vector(w2, 1, m);
free_vector(w1, 1, m4);
}

```

Correct normalization.
Normalize the output.

CITED REFERENCES AND FURTHER READING:

- Oppenheim, A.V., and Schaffer, R.W. 1989, *Discrete-Time Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall). [1]
- Harris, F.J. 1978, *Proceedings of the IEEE*, vol. 66, pp. 51–83. [2]
- Childers, D.G. (ed.) 1978, *Modern Spectrum Analysis* (New York: IEEE Press), paper by P.D. Welch. [3]
- Champeney, D.C. 1973, *Fourier Transforms and Their Physical Applications* (New York: Academic Press).
- Elliott, D.F., and Rao, K.R. 1982, *Fast Transforms: Algorithms, Analyses, Applications* (New York: Academic Press).
- Bloomfield, P. 1976, *Fourier Analysis of Time Series – An Introduction* (New York: Wiley).
- Rabiner, L.R., and Gold, B. 1975, *Theory and Application of Digital Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall).

13.5 Digital Filtering in the Time Domain

Suppose that you have a signal that you want to filter digitally. For example, perhaps you want to apply *high-pass* or *low-pass* filtering, to eliminate noise at low or high frequencies respectively; or perhaps the interesting part of your signal lies only in a certain frequency band, so that you need a *bandpass* filter. Or, if your measurements are contaminated by 60 Hz power-line interference, you may need a *notch filter* to remove only a narrow band around that frequency. This section speaks particularly about the case in which you have chosen to do such filtering in the time domain.

Before continuing, we hope you will reconsider this choice. Remember how convenient it is to filter in the Fourier domain. You just take your whole data record, FFT it, multiply the FFT output by a filter function $\mathcal{H}(f)$, and then do an inverse FFT to get back a filtered data set in time domain. Here is some additional background on the Fourier technique that you will want to take into account.

- Remember that you must define your filter function $\mathcal{H}(f)$ for both positive and negative frequencies, and that the magnitude of the frequency extremes is always the Nyquist frequency $1/(2\Delta)$, where Δ is the sampling interval. The magnitude of the smallest nonzero frequencies in the FFT is $\pm 1/(N\Delta)$, where N is the number of (complex) points in the FFT. The positive and negative frequencies to which this filter are applied are arranged in wrap-around order.
- If the measured data are real, and you want the filtered output also to be real, then your arbitrary filter function should obey $\mathcal{H}(-f) = \mathcal{H}(f)^*$. You can arrange this most easily by picking an $\mathcal{H}$ that is real and even in f .

- If your chosen $\mathcal{H}(f)$ has sharp vertical edges in it, then the *impulse response* of your filter (the output arising from a short impulse as input) will have damped “ringing” at frequencies corresponding to these edges. There is nothing wrong with this, but if you don’t like it, then pick a smoother $\mathcal{H}(f)$. To get a first-hand look at the impulse response of your filter, just take the inverse FFT of your $\mathcal{H}(f)$. If you smooth all edges of the filter function over some number k of points, then the impulse response function of your filter will have a span on the order of a fraction $1/k$ of the whole data record.
- If your data set is too long to FFT all at once, then break it up into segments of any convenient size, as long as they are much longer than the impulse response function of the filter. Use zero-padding, if necessary.
- You should probably remove any trend from the data, by subtracting from it a straight line through the first and last points (i.e., make the first and last points equal to zero). If you are segmenting the data, then you can pick overlapping segments and use only the middle section of each, comfortably distant from edge effects.
- A digital filter is said to be *causal* or *physically realizable* if its output for a particular time-step depends only on inputs at that particular time-step or earlier. It is said to be *acausal* if its output can depend on both earlier and later inputs. Filtering in the Fourier domain is, in general, acausal, since the data are processed “in a batch,” without regard to time ordering. Don’t let this bother you! Acausal filters can generally give superior performance (e.g., less dispersion of phases, sharper edges, less asymmetric impulse response functions). People use causal filters not because they are better, but because some situations just don’t allow access to out-of-time-order data. Time domain filters can, in principle, be either causal or acausal, but they are most often used in applications where physical realizability is a constraint. For this reason we will restrict ourselves to the causal case in what follows.

If you are still favoring time-domain filtering after all we have said, it is probably because you have a real-time application, for which you must process a continuous data stream and wish to output filtered values at the same rate as you receive raw data. Otherwise, it may be that the quantity of data to be processed is so large that you can afford only a very small number of floating operations on each data point and cannot afford even a modest-sized FFT (with a number of floating operations per data point several times the logarithm of the number of points in the data set or segment).

Linear Filters

The most general linear filter takes a sequence x_k of input points and produces a sequence y_n of output points by the formula

$$y_n = \sum_{k=0}^M c_k x_{n-k} + \sum_{j=1}^N d_j y_{n-j} \quad (13.5.1)$$

Here the $M + 1$ coefficients c_k and the N coefficients d_j are fixed and define the filter response. The filter (13.5.1) produces each new output value from the current and M previous input values, and from its own N previous output values. If $N = 0$, so that there is no second sum in (13.5.1), then the filter is called *nonrecursive* or *finite impulse response (FIR)*. If $N \neq 0$, then it is called *recursive* or *infinite impulse response (IIR)*. (The term “IIR” connotes only that such filters are *capable* of having infinitely long impulse responses, not that their impulse response is necessarily long in a particular application. Typically the response of an IIR filter will drop off exponentially at late times, rapidly becoming negligible.)

The relation between the c_k ’s and d_j ’s and the filter response function $\mathcal{H}(f)$ is

$$\mathcal{H}(f) = \frac{\sum_{k=0}^M c_k e^{-2\pi i k(f\Delta)}}{1 - \sum_{j=1}^N d_j e^{-2\pi i j(f\Delta)}} \quad (13.5.2)$$

where Δ is, as usual, the sampling interval. The Nyquist interval corresponds to $f\Delta$ between $-1/2$ and $1/2$. For FIR filters the denominator of (13.5.2) is just unity.

Equation (13.5.2) tells how to determine $\mathcal{H}(f)$ from the c 's and d 's. To design a filter, though, we need a way of doing the inverse, getting a suitable set of c 's and d 's — as small a set as possible, to minimize the computational burden — from a desired $\mathcal{H}(f)$. Entire books are devoted to this issue. Like many other “inverse problems,” it has no all-purpose solution. One clearly has to make compromises, since $\mathcal{H}(f)$ is a full continuous function, while the short list of c 's and d 's represents only a few adjustable parameters. The subject of digital filter design concerns itself with the various ways of making these compromises. We cannot hope to give any sort of complete treatment of the subject. We can, however, sketch a couple of basic techniques to get you started. For further details, you will have to consult some specialized books (see references).

FIR (Nonrecursive) Filters

When the denominator in (13.5.2) is unity, the right-hand side is just a discrete Fourier transform. The transform is easily invertible, giving the desired small number of c_k coefficients in terms of the same small number of values of $\mathcal{H}(f_i)$ at some discrete frequencies f_i . This fact, however, is not very useful. The reason is that, for values of c_k computed in this way, $\mathcal{H}(f)$ will tend to oscillate wildly in between the discrete frequencies where it is pinned down to specific values.

A better strategy, and one which is the basis of several formal methods in the literature, is this: Start by pretending that you are willing to have a relatively large number of filter coefficients, that is, a relatively large value of M . Then $\mathcal{H}(f)$ can be fixed to desired values on a relatively fine mesh, and the M coefficients c_k , $k = 0, \dots, M - 1$ can be found by an FFT. Next, truncate (set to zero) most of the c_k 's, leaving nonzero only the first, say, K , ($c_0, c_1, \dots, c_{K-1}$) and last $K - 1$, ($c_{M-K+1}, \dots, c_{M-1}$). The last few c_k 's are filter coefficients at *negative lag*, because of the wrap-around property of the FFT. But we don't want coefficients at negative lag. Therefore we cyclically shift the array of c_k 's, to bring everything to positive lag. (This corresponds to introducing a time-delay into the filter.) Do this by copying the c_k 's into a new array of length M in the following order:

$$(c_{M-K+1}, \dots, c_{M-1}, c_0, c_1, \dots, c_{K-1}, 0, 0, \dots, 0) \quad (13.5.3)$$

To see if your truncation is acceptable, take the FFT of the array (13.5.3), giving an approximation to your original $\mathcal{H}(f)$. You will generally want to compare the *modulus* $|\mathcal{H}(f)|$ to your original function, since the time-delay will have introduced complex phases into the filter response.

If the new filter function is acceptable, then you are done and have a set of $2K - 1$ filter coefficients. If it is not acceptable, then you can either (i) increase K and try again, or (ii) do something fancier to improve the acceptability for the same K . An example of something fancier is to modify the magnitudes (but not the phases) of the unacceptable $\mathcal{H}(f)$ to bring it more in line with your ideal, and then to FFT to get new c_k 's. Once again set to zero all but the first $2K - 1$ values of these (no need to cyclically shift since you have preserved the time-delaying phases), then inverse transform to get a new $\mathcal{H}(f)$, which will often be more acceptable. You can iterate this procedure. Note, however, that the procedure will not converge if your requirements for acceptability are more stringent than your $2K - 1$ coefficients can handle.

The key idea, in other words, is to iterate between the space of coefficients and the space of functions $\mathcal{H}(f)$, until a Fourier conjugate pair that satisfies the imposed constraints in *both spaces* is found. A more formal technique for this kind of iteration is the *Remez Exchange Algorithm* which produces the best Chebyshev approximation to a given desired frequency response with a fixed number of filter coefficients (cf. §5.13).