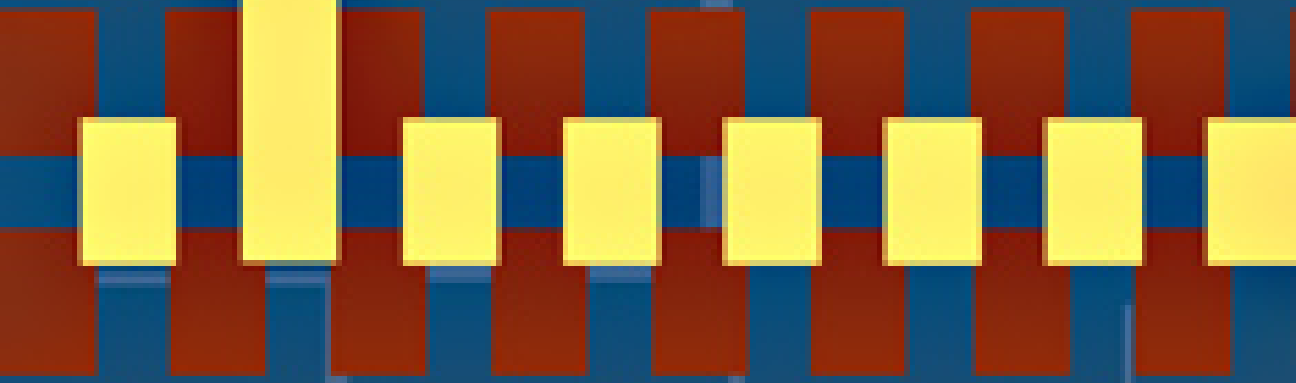
Fast Algorithms for Signal Processing

Richard E. Blahut



CAMBRIDGE

CAMBRIDGE www.cambridge.org/9780521190497

This page intentionally left blank

Fast Algorithms for Signal Processing

Efficient algorithms for signal processing are critical to very large scale future applications such as video processing and four-dimensional medical imaging. Similarly, efficient algorithms are important for embedded and power-limited applications since, by reducing the number of computations, power consumption can be reduced considerably. This unique textbook presents a broad range of computationally-efficient algorithms, describes their structure and implementation, and compares their relative strengths. All the necessary background mathematics is presented, and theorems are rigorously proved. The book is suitable for researchers and practitioners in electrical engineering, applied mathematics, and computer science.

Richard E. Blahut is a Professor of Electrical and Computer Engineering at the University of Illinois, Urbana-Champaign. He is Life Fellow of the IEEE and the recipient of many awards including the IEEE Alexander Graham Bell Medal (1998) and Claude E. Shannon Award (2005), the Tau Beta Pi Daniel C. Drucker Eminent Faculty Award, and the IEEE Millennium Medal. He was named a Fellow of the IBM Corporation in 1980, where he worked for over 30 years, and was elected to the National Academy of Engineering in 1990.

Fast Algorithms for Signal Processing

Richard E. Blahut

Henry Magnuski Professor in Electrical and Computer Engineering,
University of Illinois, Urbana-Champaign

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore,
São Paulo, Delhi, Dubai, Tokyo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521190497

First published in print format 2010

ISBN-13 978-0-511-77118-7 eBook (Adobe Reader)

ISBN-13 978-0-521-19049-7 Hardback

In loving memory of

Jeffrey Paul Blahut

May 2, 1968 – June 13, 2004

Many small make a great.

— Chaucer

Contents

Preface

A quarter of a century has passed since the previous version[1] of this book was published, and signal processing continues to be a very important part of electrical engineering. It forms an essential part of systems for telecommunications, radar and sonar, image formation systems such as medical imaging, and other large computational problems, such as in electromagnetics or fluid dynamics, geophysical exploration, and so on. Fast computational algorithms are necessary in large problems of signal processing, and the study of such algorithms is the subject of this book. Over those several decades, however, the nature of the need for fast algorithms has shifted both to much larger systems on the one hand and to embedded power-limited applications on the other.

Because many processors and many problems are much larger now than they were when the original version of this book was written, and the relative cost of addition and multiplication now may appear to be less dramatic, some of the topics of twenty years ago may be seen by some to be of less importance today. I take exactly the opposite point of view for several reasons. Very large three-dimensional or four-dimensional problems now under consideration require massive amounts of computation and this computation can be reduced by orders of magnitude in many cases by the choice of algorithm. Indeed, these very large problems can be especially suitable for the benefits of fast algorithms. At the same time, smaller signal processing problems now appear frequently in handheld or remote applications where power may be scarce or nonrenewable. The designer's care in treating an embedded application, such as a digital television, can repay itself many times by significantly reducing the power expenditure. Moreover, the unfamiliar algorithms of this book now can often be handled automatically by computerized design tools, and in embedded applications where power dissipation must be minimized, a search for the algorithm with the fewest operations may be essential.

Because the book has changed in its details and the title has been slightly modernized, it is more than a second edition, although most of the topics of the original book have been retained in nearly the same form, but usually with the presentation rewritten. Possibly, in time, some of these topics will re-emerge in a new form, but that time

[1] *Fast Algorithms for Digital Signal Processing*, Addison-Wesley, Reading, MA, 1985.

is not now. A newly written book might look different in its choice of topics and its balance between topics than does this one. To accommodate this consideration here, the chapters have been rearranged and revised, even those whose content has not changed substantially. Some new sections have been added, and all of the book has been polished, revised, and re-edited. Most of the touch and feel of the original book is still evident in this new version.

The heart of the book is in the Fourier transform algorithms of Chapters 3 and 12 and the convolution algorithms of Chapters 5 and 11. Chapters 12 and 11 are the multidimensional continuations of Chapters 3 and 4, respectively, and can be partially read immediately thereafter if desired. The study of one-dimensional convolution algorithms and Fourier transform algorithms is only completed in the context of the multidimensional problems. Chapters 2 and 9 are mathematical interludes; some readers may prefer to treat them as appendices, consulting them only as needed. The remainder, Chapters 4, 7, and 8, are in large part independent of the rest of the book. Each can be read independently with little difficulty.

This book uses branches of mathematics that the typical reader with an engineering education will not know. Therefore these topics are developed in Chapters 2 and 9, and all theorems are rigorously proved. I believe that if the subject is to continue to mature and stand on its own, the necessary mathematics must be a part of such a book; appeal to a distant authority will not do. Engineers cannot confidently advance through the subject if they are frequently asked to accept an assertion or to visit their mathematics library.

Acknowledgments

My major debt in writing this book is to Shmuel Winograd. Without his many contributions to the subject, the book would be shapeless and much shorter. He was also generous with his time in clarifying many points to me, and in reviewing early drafts of the original book. The papers of Winograd and also the book of Nussbaumer were a source for much of the material discussed in this book.

The original version of this book could not have reached maturity without being tested, critiqued, and rewritten repeatedly. I remain indebted to Professor B. W. Dickinson, Professor Toby Berger, Professor C. S. Burrus, Professor J. Gibson, Professor J. G. Proakis, Professor T. W. Parks, Dr B. Rice, Professor Y. Sugiyama, Dr W. Vanderkulk, and Professor G. Verghese for their gracious criticisms of the original 1985 manuscript. That book could not have been written without the support that was given by the International Business Machines Corporation. I am deeply grateful to IBM for this support and also to Cornell University for giving me the opportunity to teach several times from the preliminary manuscript of the earlier book. The revised book was written in the wonderful collaborative environment of the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory of the University of Illinois. The quality of the book has much to with the composition skills of Mrs Francie Bridges and the editing skills of Mrs Helen Metzinger. And, as always, Barbara made it possible.

1 Introduction

Algorithms for computation are found everywhere, and efficient versions of these algorithms are highly valued by those who use them. We are mainly concerned with certain types of computation, primarily those related to signal processing, including the computations found in digital filters, discrete Fourier transforms, correlations, and spectral analysis. Our purpose is to present the advanced techniques for fast digital implementation of these computations. We are not concerned with the function of a digital filter or with how it should be designed to perform a certain task; our concern is only with the computational organization of its implementation. Nor are we concerned with why one should want to compute, for example, a discrete Fourier transform; our concern is only with how it can be computed efficiently. Surprisingly, there is an extensive body of theory dealing with this specialized topic – the topic of fast algorithms.

1.1 Introduction to fast algorithms

An algorithm, like most other engineering devices, can be described either by an input/output relationship or by a detailed explanation of its internal construction. When one applies the techniques of signal processing to a new problem one is concerned only with the input/output aspects of the algorithm. Given a signal, or a data record of some kind, one is concerned with what should be done to this data, that is, with what the output of the algorithm should be when such and such a data record is the input. Perhaps the output is a filtered version of the input, or the output is the Fourier transform of the input. The relationship between the input and the output of a computational task can be expressed mathematically without prescribing in detail all of the steps by which the calculation is to be performed.

Devising such an algorithm for an information processing problem, from this input/output point of view, may be a formidable and sophisticated task, but this is not our concern in this book. We will assume that we are given a specification of a relationship between input and output, described in terms of filters, Fourier transforms, interpolations, decimations, correlations, modulations, histograms, matrix operations,

and so forth. All of these can be expressed with mathematical formulas and so can be computed just as written. This will be referred to as the obvious implementation.

One may be content with the obvious implementation, and it might not be apparent that the obvious implementation need not be the most efficient. But once people began to compute such things, other people began to look for more efficient ways to compute them. This is the story we aim to tell, the story of fast algorithms for signal processing. By a fast algorithm, we mean a detailed description of a computational procedure that is not the obvious way to compute the required output from the input. A fast algorithm usually gives up a conceptually clear computation in favor of one that is computationally efficient.

Suppose we need to compute a number A, given by

$$A = ac + ad + bc + bd.$$

As written, this requires four multiplications and three additions to compute. If we need to compute A many times with different sets of data, we will quickly notice that

$$A = (a + b)(c + d)$$

is an equivalent form that requires only one multiplication and two additions, and so it is to be preferred. This simple example is quite obvious, but really illustrates most of what we shall talk about. Everything we do can be thought of in terms of the clever insertion of parentheses in a computational problem. But in a big problem, the fast algorithms cannot be found by inspection. It will require a considerable amount of theory to find them.

A nontrivial yet simple example of a fast algorithm is an algorithm for complex multiplication. The complex product[1]

$$(e + \mathrm{j}f) = (a + \mathrm{j}b) \cdot (c + \mathrm{j}d)$$

can be defined in terms of real multiplications and real additions as

$$e = ac - bd$$
$$f = ad + bc.$$

We see that these formulas require four real multiplications and two real additions. A more efficient "algorithm" is

$$e = (a - b)d + a(c - d)$$
$$f = (a - b)d + b(c + d)$$

whenever multiplication is harder than addition. This form requires three real multiplications and five real additions. If c and d are constants for a series of complex

[1] The letter j is used for $\sqrt{-1}$ and j is used as an index throughout the book. This should not cause any confusion.

multiplications, then the terms $c + d$ and $c - d$ are constants also and can be computed off-line. It then requires three real multiplications and three real additions to do one complex multiplication.

We have traded one multiplication for an addition. This can be a worthwhile saving, but only if the signal processor is designed to take advantage of it. Most signal processors, however, have been designed with a prejudice for a complex multiplication that uses four multiplications. Then the advantage of the improved algorithm has no value. The storage and movement of data between additions and multiplications are also important considerations in determining the speed of a computation and of some importance in determining power dissipation.

We can dwell further on this example as a foretaste of things to come. The complex multiplication above can be rewritten as a matrix product

$$\begin{bmatrix} e \\ f \end{bmatrix} = \begin{bmatrix} c & -d \\ d & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix},$$

where the vector $\begin{bmatrix} a \\ b \end{bmatrix}$ represents the complex number $a + \mathrm{j}b$, the matrix $\begin{bmatrix} c & -d \\ d & c \end{bmatrix}$ represents the complex number $c + \mathrm{j}d$, and the vector $\begin{bmatrix} e \\ f \end{bmatrix}$ represents the complex number $e + \mathrm{j}f$. The matrix–vector product is an unconventional way to represent complex multiplication. The alternative computational algorithm can be written in matrix form as

$$\begin{bmatrix} e \\ f \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} (c-d) & 0 & 0 \\ 0 & (c+d) & 0 \\ 0 & 0 & d \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}.$$

The algorithm, then, can be thought of as nothing more than the unusual matrix factorization:

$$\begin{bmatrix} c & -d \\ d & c \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} (c-d) & 0 & 0 \\ 0 & (c+d) & 0 \\ 0 & 0 & d \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & -1 \end{bmatrix}.$$

We can abbreviate the algorithm as

$$\begin{bmatrix} e \\ f \end{bmatrix} = \boldsymbol{BDA} \begin{bmatrix} a \\ b \end{bmatrix},$$

where $\boldsymbol{A}$ is a three by two matrix that we call a matrix of preadditions; $\boldsymbol{D}$ is a three by three diagonal matrix that is responsible for all of the general multiplications; and $\boldsymbol{B}$ is a two by three matrix that we call a matrix of postadditions.

We shall find that many fast computational procedures for convolution and for the discrete Fourier transform can be put into this factored form of a diagonal matrix in the center, and on each side of which is a matrix whose elements are 1, 0, and -1. Multiplication by a matrix whose elements are 0 and ± 1 requires only additions and subtractions. Fast algorithms in this form will have the structure of a batch of additions, followed by a batch of multiplications, followed by another batch of additions.

The final example of this introductory section is a fast algorithm for multiplying two arbitrary matrices. Let

$$\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B},$$

where $\boldsymbol{A}$ and $\boldsymbol{B}$ are any ℓ by n, and n by m, matrices, respectively. The standard method for computing the matrix $\boldsymbol{C}$ is

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \qquad \begin{matrix} i = 1, \ldots, \ell \\ j = 1, \ldots, m, \end{matrix}$$

which, as it is written, requires $m\ell n$ multiplications and $(n-1)\ell m$ additions. We shall give an algorithm that reduces the number of multiplications by almost a factor of two but increases the number of additions. The total number of operations increases slightly.

We use the identity

$$a_1 b_1 + a_2 b_2 = (a_1 + b_2)(a_2 + b_1) - a_1 a_2 - b_1 b_2$$

on the elements of $\boldsymbol{A}$ and $\boldsymbol{B}$. Suppose that n is even (otherwise append a column of zeros to $\boldsymbol{A}$ and a row of zeros to $\boldsymbol{B}$, which does not change the product $\boldsymbol{C}$). Apply the above identity to pairs of columns of $\boldsymbol{A}$ and pairs of rows of $\boldsymbol{B}$ to write

$$\begin{aligned} c_{ij} &= \sum_{i=1}^{n/2} (a_{i,2k-1} b_{2k-1,j} + a_{i,2k} b_{2k,j}) \\ &= \sum_{k=1}^{n/2} (a_{i,2k-1} + b_{2k,j})(a_{i,2k} + b_{2k-1,j}) - \sum_{k=1}^{n/2} a_{i,2k-1} a_{i,2k} - \sum_{k=1}^{n/2} b_{2k-1,j} b_{2k,j} \end{aligned}$$

for $i = 1, \ldots, \ell$ and $j = 1, \ldots, m$.

This results in computational savings because the second term depends only on i and need not be recomputed for each j, and the third term depends only on j and need not be recomputed for each i. The total number of multiplications used to compute matrix $\boldsymbol{C}$ is $\frac{1}{2}n\ell m + \frac{1}{2}n(\ell + m)$, and the total number of additions is $\frac{3}{2}n\ell m + \ell m + \left(\frac{1}{2}n - 1\right)(\ell + m)$. For large matrices the number of multiplications is about half the direct method.

This last example may be a good place for a word of caution about numerical accuracy. Although the number of multiplications is reduced, this algorithm is more sensitive to roundoff error unless it is used with care. By proper scaling of intermediate steps,

Algorithm	Multiplications/pixel*	Additions/pixel
Direct computation of discrete Fourier transform 1000 x 1000	8000	4000
Basic Cooley–Tukey FFT 1024 x 1024	40	60
Hybrid Cooley–Tukey/Winograd FFT 1000 x 1000	40	72.8
Winograd FFT 1008 x 1008	6.2	91.6
Nussbaumer–Quandalle FFT 1008 x 1008	4.1	79

*1 pixel – 1 output grid point

Figure 1.1 Relative performance of some two-dimensional Fourier transform algorithms

however, one can obtain computational accuracy that is nearly the same as the direct method. Consideration of computational noise is always a practical factor in judging a fast algorithm, although we shall usually ignore it. Sometimes when the number of operations is reduced, the computational noise is reduced because fewer computations mean that there are fewer sources of noise. In other algorithms, though there are fewer sources of computational noise, the result of the computation may be more sensitive to one or more of them, and so the computational noise in the result may be increased.

Most of this book will be spent studying only a few problems: the problems of linear convolution, cyclic convolution, multidimensional linear convolution, multidimensional cyclic convolution, the discrete Fourier transform, the multidimensional discrete Fourier transforms, the solution of Toeplitz systems, and finding paths in a trellis. Some of the techniques we shall study deserve to be more widely used – multidimensional Fourier transform algorithms can be especially good if one takes the pains to understand the most efficient ones. For example, Figure 1.1 compares some methods of computing a two-dimensional Fourier transform. The improvements in performance come more slowly toward the end of the list. It may not seem very important to reduce the number of multiplications per output cell from six to four after the reduction has already gone from forty to six, but this can be a shortsighted view. It is an additional savings and may be well worth the design time in a large application. In power-limited applications, a potential of a significant reduction in power may itself justify the effort.

There is another important lesson contained in Figure 1.1. An entry, labeled the *hybrid Cooley–Tukey/Winograd FFT*, can be designed to compute a 1000 by 1000-point two-dimensional Fourier transform with forty real multiplications per grid point. This example may help to dispel an unfortunate myth that the discrete Fourier transform is practical only if the blocklength is a power of two. In fact, there is no need to insist

that one should use only a power of two blocklength; good algorithms are available for many values of the blocklength.

1.2 Applications of fast algorithms

Very large scale integrated circuits, or chips, are now widely available. A modern chip can easily contain many millions of logic gates and memory cells, and it is not surprising that the theory of algorithms is looked to as a way to efficiently organize these gates on special-purpose chips. Sometimes a considerable performance improvement, either in speed or in power dissipation, can be realized by the choice of algorithm. Of course, a performance improvement in speed can also be realized by increasing the size or the speed of the chip. These latter approaches are more widely understood and easier to design, but they are not the only way to reduce power or chip size.

For example, suppose one devises an algorithm for a Fourier transform that has only one-fifth of the computation of another Fourier transform algorithm. By using the new algorithm, one might realize a performance improvement that can be as real as if one increased the speed or the size of the chip by a factor of five. To realize this improvement, however, the chip designer must reflect the architecture of the algorithm in the architecture of the chip. A naive design can dissipate the advantages by increasing the complexity of indexing, for example, or of data flow between computational steps. An understanding of the fast algorithms described in this book will be required to obtain the best system designs in the era of very large-scale integrated circuits.

At first glance, it might appear that the two kinds of development – fast circuits and fast algorithms – are in competition. If one can build the chip big enough or fast enough, then it seemingly does not matter if one uses inefficient algorithms. No doubt this view is sound in some cases, but in other cases one can also make exactly the opposite argument. Large digital signal processors often create a need for fast algorithms. This is because one begins to deal with signal-processing problems that are much larger than before. Whether competing algorithms for some problem of interest have running times proportional to n^2 or n^3 may be of minor importance when n equals three or four; but when n equals 1000, it becomes critical.

The fast algorithms we shall develop are concerned with digital signal processing, and the applications of the algorithms are as broad as the application of digital signal processing itself. Now that it is practical to build a sophisticated algorithm for signal processing onto a chip, we would like to be able to choose such an algorithm to maximize the performance of the chip. But to do this for a large chip involves a considerable amount of theory. In its totality the theory goes well beyond the material that will be discussed in this book. Advanced topics in logic design and computer architecture, such as parallelism and pipelining, must also be studied before one can determine all aspects of practical complexity.

We usually measure the performance of an algorithm by the number of multiplications and additions it uses. These performance measures are about as deep as one can go at the level of the computational algorithm. At a lower level, we would want to know the area of the chip or the number of gates on it and the time required to complete a computation. Often one judges a circuit by the area–time product. We will not give performance measures at this level because this is beyond the province of the algorithm designer, and entering the province of the chip architecture.

The significance of the topics in this book cannot be appreciated without understanding the massive needs of some processing applications of the near future and the power limitations of other embedded applications now in widespread use. At the present time, applications are easy to foresee that require orders of magnitude more signal processing than current technology can satisfy.

Sonar systems have now become almost completely digital. Though they process only a few kilohertz of signal bandwidth, these systems can use hundreds of millions of multiplications per second and beyond, and even more additions. Extensive racks of digital equipment may be needed for such systems, and yet reasons for even more processing in sonar systems are routinely conceived.

Radar systems also have become digital, but many of the front-end functions are still done by conventional microwave or analog circuitry. In principle, radar and sonar are quite similar, but radar has more than one thousand times as much bandwidth. Thus, one can see the enormous potential for digital signal processing in radar systems.

Seismic processing provides the principal method for exploration deep below the Earth's surface. This is an important method of searching for petroleum reserves. Many computers are already busy processing the large stacks of seismic data, but there is no end to the seismic computations remaining to be done.

Computerized tomography is now widely used to synthetically form images of internal organs of the human body by using X-ray data from multiple projections. Improved algorithms are under study that will reduce considerably the X-ray dosage, or provide motion or function to the imagery, but the signal-processing requirements will be very demanding. Other forms of medical imaging continue to advance, such as those using ultrasonic data, nuclear magnetic resonance data, or particle decay data. These also use massive amounts of digital signal processing.

It is also possible, in principle, to enhance poor-quality photographs. Pictures blurred by camera motion or out-of-focus pictures can be corrected by signal processing. However, to do this digitally takes large amounts of signal-processing computations. Satellite photographs can be processed digitally to merge several images or enhance features, or combine information received on different wavelengths, or create stereoscopic images synthetically. For example, for meteorological research, one can create a moving three-dimensional image of the cloud patterns moving above the Earth's surface based on a sequence of satellite photographs from several aspects. The nondestructive testing of

manufactured articles, such as castings, is possible by means of computer-generated internal images based on the response to induced acoustic vibrations.

Other applications for the fast algorithms of signal processing could be given, but these should suffice to prove the point that a need exists and continues to grow for fast signal-processing algorithms.

All of these applications are characterized by computations that are massive but are fairly straightforward and have an orderly structure. In addition, in such applications, once a hardware module or a software subroutine is designed to do a certain task, it is permanently dedicated to this task. One is willing to make a substantial design effort because the design cost is not what matters; the operational performance, both speed and power dissipation, is far more important.

At the same time, there are embedded applications for which power reduction is of critical importance. Wireless handheld and desktop devices and untethered remote sensors must operate from batteries or locally generated power. Chips for these devices may be produced in the millions. Nonrecurring design time to reduce the computations needed by the required algorithm is one way to reduce the power requirements.

1.3 Number systems for computation

Throughout the book, when we speak of the complexity of an algorithm, we will cite the number of multiplications and additions, as if multiplications and additions were fundamental units for measuring complexity. Sometimes one may want to go a little deeper than this and look at how the multiplier is built so that the number of bit operations can be counted. The structure of a multiplier or adder critically depends on how the data is represented. Though we will not study such issues of number representation, a few words are warranted here in the introduction.

To take an extreme example, if a computation involves mostly multiplication, the complexity may be less if the data is provided in the form of logarithms. The additions will now be more complicated; but if there are not too many additions, a savings will result. This is rarely the case, so we will generally assume that the input data is given in its natural form either as real numbers, as complex numbers, or as integers.

There are even finer points to consider in practical digital signal processors. A number is represented by a binary pattern with a finite number of bits; both floating-point numbers and fixed-point numbers are in use. Fixed-point arithmetic suffices for most signal-processing tasks, and so it should be chosen for reasons of economy. This point cannot be stressed too strongly. There is always a temptation to sweep away many design concerns by using only floating-point arithmetic. But if a chip or an algorithm is to be dedicated to a single application for its lifetime – for example, a digital-processing chip to be used in a digital radio or television for the consumer market – it is not the design cost that matters; it is the performance of the equipment, the power dissapation,

and the recurring manufacturing costs that matter. Money spent on features to ease the designer's work cannot be spent to increase performance.

A nonnegative integer j smaller than q^m has an m-symbol fixed-point radix-q representation, given by

$$j = j_0 + j_1 q + j_2 q^2 + \cdots + j_{m-1} q^{m-1}, \qquad 0 \leq j_i < q.$$

The integer j is represented by the m-tuple of coefficients $(j_0, j_1, \ldots, j_{m-1})$. Several methods are used to handle the sign of a fixed-point number. These are sign-and-magnitude numbers, q-complement numbers, and $(q-1)$-complement numbers. The same techniques can be used for numbers expressed in any base. In a binary notation, q equals two, and the complement representations are called two's-complement numbers and one's-complement numbers.

The sign-and-magnitude convention is easiest to understand. The magnitude of the number is augmented by a special digit called the sign digit; it is zero – indicating a plus sign – for positive numbers and it is one – indicating a minus sign – for negative numbers. The sign digit is treated differently from the magnitude digits during addition and multiplication, in the customary way. The complement notations are a little harder to understand, but often are preferred because the hardware is simpler; an adder can simply add two numbers, treating the sign digit the same as the magnitude digits. The sign-and-magnitude convention and the $(q-1)$-complement convention each leads to the existence of both a positive and a negative zero. These are equal in meaning, but have separate representations. The two's-complement convention in binary arithmetic and the ten's-complement convention in decimal arithmetic have only a single representation for zero.

The $(q-1)$-complement notation represents the negative of a number by replacing digit j, including the sign digit, by $q-1-j$. For example, in nine's-complement notation, the negative of the decimal number $+62$, which is stored as 062, is 937; and the negative of the one's-complement binary number $+011$, which is stored as 0011, is 1100. The $(q-1)$-complement representation has the feature that one can multiply any number by minus one simply by taking the $(q-1)$-complement of each digit.

The q-complement notation represents the negative of a number by adding one to the $(q-1)$-complement notation. The negative of zero is zero. In this convention, the negative of the decimal number $+62$, which is stored as 062, is 938; and the negative of the binary number $+011$, which is stored as 0011, is 1101.

1.4 Digital signal processing

The most important task of digital signal processing is the task of filtering a long sequence of numbers, and the most important device is the digital filter. Normally, the data sequence has an unspecified length and is so long as to appear infinite to the

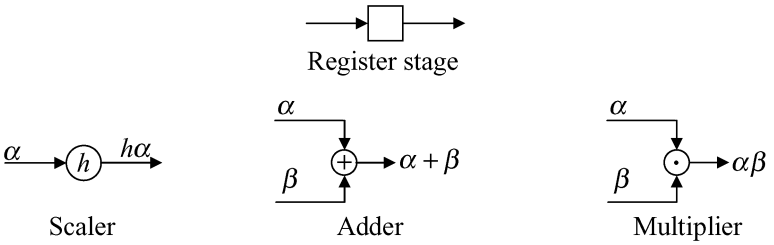


Figure 1.2 Circuit elements

Figure 1.3 A shift register

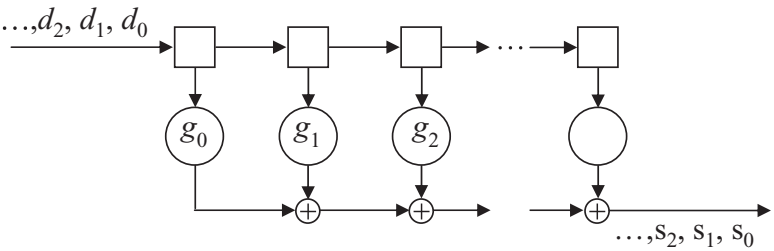


Figure 1.4 A finite-impulse-response filter

processing. The numbers in the sequence are usually either real numbers or complex numbers, but other kinds of number sometimes occur. A digital filter is a device that produces a new sequence of numbers, called the *output sequence*, from the given sequence, now called the *input sequence*. Filters in common use can be constructed out of those circuit elements, illustrated in Figure 1.2, called *shift-register stages*, *adders*, *scalers*, and *multipliers*. A shift-register stage holds a single number, which it displays on its output line. At discrete time instants called *clock times*, the shift-register stage replaces its content with the number appearing on the input line, discarding its previous content. A shift register, illustrated in Figure 1.3, is a number of shift-register stages connected in a chain.

The most important kinds of digital filter that we shall study are those known as *finite-impulse-response* (FIR) *filters* and *autoregressive filters*. A FIR filter is simply a tapped shift register, illustrated in Figure 1.4, in which the output of each stage is multiplied by a fixed constant and all outputs are added together to provide the filter output. The output of the FIR filter is a linear convolution of the input sequence and the sequence describing the filter tap weights. An autoregressive filter is also a tapped shift register, now with the output of the filter fed back to the input, as shown in Figure 1.5.

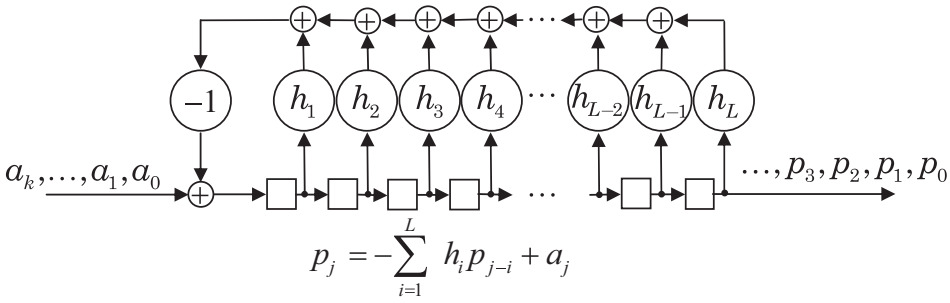


Figure 1.5 An autoregressive filter

Linear convolution is perhaps the most common computational problem found in signal processing, and we shall spend a great deal of time studying how to implement it efficiently. We shall spend even more time studying ways to compute a cyclic convolution. This may seem a little strange because a cyclic convolution does not often arise naturally in applications. We study it because there are so many good ways to compute a cyclic convolution. Therefore we will develop fast methods of computing long linear convolutions by patching together many cyclic convolutions.

Given the two sequences called the *data sequence*

$$\boldsymbol{d} = \{d_i \mid i = 0, \ldots, N-1\}$$

and the *filter sequence*

$$\boldsymbol{g} = \{g_i \mid i = 0, \ldots, L-1\},$$

where N is the data blocklength and L is the filter blocklength, the linear convolution is a new sequence called the *signal sequence* or the *output sequence*

$$\boldsymbol{s} = \{s_i \mid i = 0, \ldots, L+N-2\},$$

given by the equation

$$s_i = \sum_{k=0}^{N-1} g_{i-k} d_k, \qquad i = 0, \ldots, L+N-2,$$

and $L + N - 1$ is the output blocklength. The convolution is written with the understanding that $g_{i-k} = 0$ if $i - k$ is less than zero. Because each component of $\boldsymbol{d}$ multiplies, in turn, each component of $\boldsymbol{g}$, there are NL multiplications in the obvious implementation of the convolution.

There is a very large body of theory dealing with the design of a FIR filter in the sense of choosing the length L and the tap weights g_i to suit a given application. We are not concerned with this aspect of filter design; our concern is only with fast algorithms for computing the filter output $\boldsymbol{s}$ from the filter $\boldsymbol{g}$ and the input sequence $\boldsymbol{d}$.

A concept closely related to the convolution is the correlation, given by

$$r_i = \sum_{k=0}^{N-1} g_{i+k} d_k, \quad i = 0, \ldots, L + N - 2,$$

where $g_{i+k} = 0$ for $i + k \geq L$. The correlation can be computed as a convolution simply by reading one of the two sequences backwards. All of the methods for computing a linear convolution are easily changed into methods for computing the correlation.

We can also express the convolution in the notation of polynomials. Let

$$d(x) = \sum_{i=0}^{N-1} d_i x^i,$$

$$g(x) = \sum_{i=0}^{L-1} g_i x^i.$$

Then

$$s(x) = g(x)d(x),$$

where

$$s(x) = \sum_{i=0}^{L+N-2} s_i x^i.$$

This can be seen by examining the coefficients of the product $g(x)d(x)$. Of course, we can also write

$$s(x) = d(x)g(x),$$

which makes it clear that $\boldsymbol{d}$ and $\boldsymbol{g}$ play symmetric roles in the convolution. Therefore we can also write the linear convolution in the equivalent form

$$s_i = \sum_{k=0}^{L-1} g_k d_{i-k}.$$

Another form of convolution is the *cyclic convolution*, which is closely related to the linear convolution. Given the two sequences d_i for $i = 0, \ldots, n-1$ and g_i for $i = 0, \ldots, n-1$, each of blocklength n, a new sequence s'_i for $i = 0, \ldots, n-1$ of blocklength n now is given by the cyclic convolution

$$s'_i = \sum_{k=0}^{n-1} g_{((i-k))} d_k, \qquad i = 0, \ldots, n-1,$$

where the double parentheses denote modulo n arithmetic on the indices (see Section 2.6). That is,

$$((i-k)) = (i-k) \quad \text{modulo } n$$

and

$$0 \leq ((i-k)) < n.$$

Notice that in the cyclic convolution, for every i, every d_k finds itself multiplied by a meaningful value of $g_{((i-k))}$. This is different from the linear convolution where, for some i, d_k will be multiplied by a g_{i-k} whose index is outside the range of definition of $\boldsymbol{g}$, and so is zero.

We can relate the cyclic convolution outputs to the linear convolution as follows. By the definition of the cyclic convolution

$$s'_i = \sum_{k=0}^{n-1} g_{((i-k))} d_k, \qquad i = 0, \ldots, n = 1.$$

We can recognize two kinds of term in the sum: those with $i - k \geq 0$ and those with $i - k < 0$. Those occur when $k \leq i$ and $k > i$, respectively. Hence

$$s'_i = \sum_{k=0}^{i} g_{i-k} d_k + \sum_{k=i+1}^{n-1} g_{n+i-k} d_k.$$

But now, in the first sum, $g_{i-k} = 0$ if $k > i$; and in the second sum, $g_{n+i-k} = 0$ if $k < i$. Hence we can change the limits of the summations as follows:

$$\begin{aligned} s'_i &= \sum_{k=0}^{n-1} g_{i-k} d_k + \sum_{k=0}^{n-1} g_{n+i-k} d_k, \qquad i = 0, \ldots, n-1 \\ &= s_i + s_{n+i}, \qquad i = 0, \ldots, n-1, \end{aligned}$$

which relates the cyclic convolution outputs on the left to the linear convolution outputs on the right. We say that coefficients of $\boldsymbol{s}$ with index larger than $n - 1$ are "folded" back into terms with indices smaller than n.

The linear convolution can be computed as a cyclic convolution if the second term above equals zero. This is so if $g_{n+i-k} d_k$ equals zero for all i and k. To ensure this, one can choose n, the blocklength of the cyclic convolution, so that n is larger than $L + N - 2$ (appending zeros to $\boldsymbol{g}$ and $\boldsymbol{d}$ so their blocklength is n). Then one can compute the linear convolution by using an algorithm for computing a cyclic convolution and still get the right answer.

The cyclic convolution can also be expressed as a polynomial product. Let

$$d(x) = \sum_{i=0}^{n-1} d_i x^i,$$

$$g(x) = \sum_{i=0}^{n-1} g_i x^i.$$

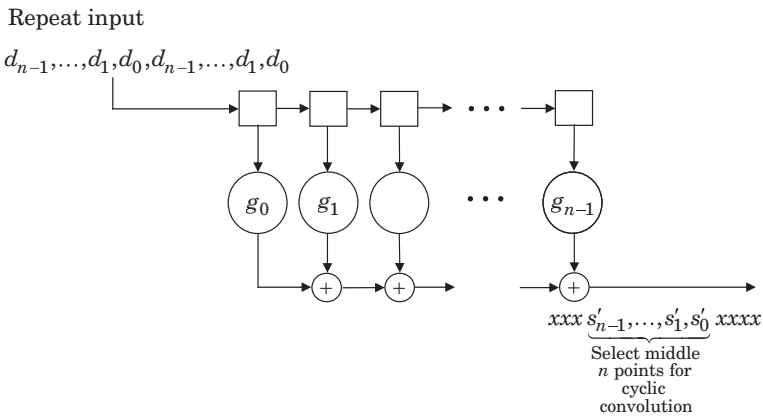


Figure 1.6 Using a FIR filter to form cyclic convolutions

Whereas the linear convolution is represented by

$$s(x) = g(x)d(x),$$

the cyclic convolution is computed by folding back the high-order coefficients of $s(x)$ by writing

$$s'(x) = g(x)d(x) \pmod{x^n - 1}.$$

By the equality modulo $x^n - 1$, we mean that $s'(x)$ is the remainder when $s(x)$ is divided by $x^n - 1$. To reduce $g(x)d(x)$ modulo $x^n - 1$, it suffices to replace x^n by one, or to replace x^{n+i} by x^i wherever a term x^{n+i} with i positive appears. This has the effect of forming the coefficients

$$s'_i = s_i + s_{n+i}, \qquad i = 0, \ldots, n-1$$

and so gives the coefficients of the cyclic convolution.

From the two forms

$$\begin{aligned} s'(x) &= d(x)g(x) \pmod{x^n - 1} \\ &= g(x)d(x) \pmod{x^n - 1}, \end{aligned}$$

it is clear that the roles of $\boldsymbol{d}$ and $\boldsymbol{g}$ are also symmetric in the cyclic convolution. Therefore we have the two expressions for the cyclic convolution

$$\begin{aligned} s'_i &= \sum_{k=0}^{n-1} g_{((i-k))} d_k, \qquad i = 0, \ldots, n-1 \\ &= \sum_{k=0}^{n-1} d_{((i-k))} g_k, \qquad i = 0, \ldots, n-1. \end{aligned}$$

Figure 1.6 shows a FIR filter that is made to compute a cyclic convolution. To do this, the sequence $\boldsymbol{d}$ is repeated. The FIR filter then produces $3n - 1$ outputs, and within

those $3n - 1$ outputs is a consecutive sequence of n outputs that is equal to the cyclic convolution.

A more important technique is to use a cyclic convolution to compute a long linear convolution. Fast algorithms for long linear convolutions break the input datastream into short sections of perhaps a few hundred samples. One section at a time is processed – often as a cyclic convolution – to produce a section of the output datastream. Techniques for doing this are called *overlap* techniques, referring to the fact that nonoverlapping sections of the input datastream cause overlapping sections of the output datastream, while nonoverlapping sections of the output datastream are caused by overlapping sections of the input datastream. Overlap techniques are studied in detail in Chapter 5.

The operation of an autoregressive filter, as was shown in Figure 1.5, also can be described in terms of polynomial arithmetic. Whereas the finite-impulse-response filter computes a polynomial product, an autoregressive filter computes a polynomial division. Specifically, when a finite sequence is filtered by an autoregressive filter (with zero initial conditions), the output sequence corresponds to the quotient polynomial under polynomial division by the polynomial whose coefficients are the tap weights, and at the instant when the input terminates, the register contains the corresponding remainder polynomial. In particular, recall that the output p_j of the autoregressive filter, by appropriate choice of the signs of the tap weights, h_i, is given by

$$p_j = -\sum_{j=1}^{L} h_i p_{j-i} + a_j,$$

where a_j is the jth input symbol and h_i is the weight of the ith tap of the filter. Define the polynomials

$$a(x) = \sum_{j=0}^{n} a_j x^j$$

and

$$h(x) = \sum_{j=0}^{L} h_j x^j,$$

and write

$$a(x) = Q(x)h(x) + r(x),$$

where $Q(x)$ and $r(x)$ are the quotient polynomial and the remainder polynomial under division of polynomials. We conclude that the filter output p_j is equal to the jth coefficient of the quotient polynomial, so $p(x) = Q(x)$. The coefficients of the remainder polynomial $r(x)$ will be left in the stages of the autoregressive filter after the n coefficients a_j are shifted in.

Another computation that is important in signal processing is that of the *discrete Fourier transform* (hereafter called simply the Fourier transform). Let

$\boldsymbol{v} = [v_i \mid i = 0, \ldots, n-1]$ be a vector of complex numbers or a vector of real numbers. The Fourier transform of $\boldsymbol{v}$ is another vector $\boldsymbol{V}$ of length n of complex numbers, given by

$$V_k = \sum_{i=0}^{n-1} \omega^{ik} v_i, \qquad k = 0, \ldots, n-1,$$

where $\omega = \mathrm{e}^{-\mathrm{j}2\pi/n}$ and $\mathrm{j} = \sqrt{-1}$.

Sometimes we write this computation as a matrix–vector product

$$\boldsymbol{V} = \boldsymbol{T}\boldsymbol{v}.$$

When written out, this becomes

$$\begin{bmatrix} V_0 \\ V_1 \\ V_2 \\ \vdots \\ V_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & \vdots & & \vdots & \\ 1 & \omega^{n-1} & \omega^{2(n-1)} & \cdots & \omega \end{bmatrix} \begin{bmatrix} v_0 \\ v_1 \\ v_2 \\ \vdots \\ v_{n-1} \end{bmatrix}.$$

If $\boldsymbol{V}$ is the Fourier transform of $\boldsymbol{v}$, then $\boldsymbol{v}$ can be recovered from $\boldsymbol{V}$ by the inverse Fourier transform, which is given by

$$v_i = \frac{1}{n} \sum_{k=0}^{n-1} \omega^{-ik} V_k.$$

The proof is as follows:

$$\begin{aligned} \sum_{k=0}^{n-1} \omega^{-ik} V_k &= \sum_{k=0}^{n-1} \omega^{-ik} \sum_{\ell=0}^{n-1} \omega^{\ell k} v_\ell \\ &= \sum_{\ell=0}^{n-1} v_\ell \left[\sum_{k=0}^{n-1} \omega^{k(\ell-i)} \right]. \end{aligned}$$

But the summation on k is clearly equal to n if ℓ is equal to i, while if ℓ is not equal to i the summation becomes

$$\sum_{k=0}^{n-1} (\omega^{(\ell-i)})^k = \frac{1 - \omega^{(\ell-i)n}}{1 - \omega^{(\ell-i)}}.$$

The right side equals zero because $\omega^n = 1$ and the denominator is not zero. Hence

$$\sum_{k=0}^{n-1} \omega^{-ik} V_k = \sum_{\ell=0}^{n-1} v_\ell (n\delta_{i\ell}) \;=\; n v_i,$$

where $\delta_{i\ell} = 1$ if $i = \ell$, and otherwise $\delta_{i\ell} = 0$.

There is an important link between the Fourier transform and the cyclic convolution. This link is known as the *convolution theorem* and goes as follows. The vector $\boldsymbol{e}$ is given by the cyclic convolution of the vectors $\boldsymbol{f}$ and $\boldsymbol{g}$:

$$e_i = \sum_{\ell=0}^{n-1} f_{((i-\ell))} g_\ell, \qquad i = 0, \ldots, n-1,$$

if and only if the Fourier transforms satisfy

$$E_k = F_k G_k, \qquad k = 0, \ldots, n-1.$$

This holds because

$$\begin{aligned} e_i &= \sum_{\ell=0}^{n-1} f_{((i-\ell))} \left[\frac{1}{n} \sum_{k=0}^{n-1} \omega^{-k\ell} G_k \right] \\ &= \frac{1}{n} \sum_{k=0}^{n-1} \omega^{-ik} G_k \left[\sum_{\ell=0}^{n-1} \omega^{(i-\ell)k} f_{((i-\ell))} \right] = \frac{1}{n} \sum_{k=0}^{n-1} \omega^{-ik} G_k F_k. \end{aligned}$$

Because $\boldsymbol{e}$ is the inverse Fourier transform of $\boldsymbol{E}$, we conclude that $E_k = G_k F_k$.

There are also two-dimensional Fourier transforms, which are useful for processing two-dimensional arrays of data, and multidimensional Fourier transforms, which are useful for processing multidimensional arrays of data. The two-dimensional Fourier transform on an n' by n'' array is

$$V_{k'k''} = \sum_{i'=0}^{n'-1} \sum_{i''=0}^{n''-1} \omega^{i'k'} \mu^{i''k''} v_{i'i''}, \qquad \begin{aligned} k' &= 0, \ldots, n'-1, \\ k'' &= 0, \ldots, n''-1, \end{aligned}$$

where $\omega = \mathrm{e}^{-\mathrm{j}2\pi/n'}$ and $\mu = \mathrm{e}^{-\mathrm{j}2\pi/n''}$. Chapter 12 is devoted to the two-dimensional Fourier transforms.

1.5 History of fast signal-processing algorithms

The telling of the history of fast signal-processing algorithms begins with the publication in 1965 of the fast Fourier transform (FFT) algorithm of Cooley and Tukey, although the history itself starts much earlier, indeed, with Gauss. The Cooley–Tukey paper appeared at just the right time and served as a catalyst to bring the techniques of signal processing into a new arrangement. Stockham (1966) soon noted that the FFT led to a good way to compute convolutions. Digital signal processing technology could immediately exploit the FFT, and so there were many applications, and the Cooley–Tukey paper was widely studied. A few years later, it was noted that there was an earlier FFT algorithm, quite different from the Cooley–Tukey FFT, due to Good (1960) and Thomas (1963). The Good–Thomas FFT algorithm had failed to attract much attention

at the time it was published. Later, a more efficient though more complicated algorithm was published by Winograd (1976, 1978), who also provided a much deeper understanding of what it means to compute the Fourier transform.

The radix-two Cooley–Tukey FFT is especially elegant and efficient, and so is very popular. This has led some to the belief that the discrete Fourier transform is practical only if the blocklength is a power of two. This belief tends to result in the FFT algorithm dictating the design parameters of an application rather than the application dictating the choice of FFT algorithm. In fact, there are good FFT algorithms for just about any blocklength.

The Cooley–Tukey FFT, in various guises, has appeared independently in other contexts. Essentially the same idea is known as the Butler matrix (1961) when it is used as a method of wiring a multibeam phased-array radar antenna.

Fast convolution algorithms of small blocklength were first constructed by Agarwal and Cooley (1977) using clever insights but without a general technique. Winograd (1978) gave a general method of construction and also proved important theorems concerning the nonexistence of better convolution algorithms in the real field or the complex field. Agarwal and Cooley (1977) also found a method to break long convolutions into short convolutions using the Chinese remainder theorem. Their method works well when combined with the Winograd algorithm for short convolutions.

The earliest idea of modern signal processing that we label as a fast algorithm came much earlier than the FFT. In 1947 the Levinson algorithm was published as an efficient method of solving certain Toeplitz systems of equations. Despite its great importance in the processing of seismic signals, the literature of the Levinson algorithm remained disjoint from the literature of the FFT for many years. Generally, the early literature does not distinguish carefully between the Levinson algorithm as a computational procedure and the filtering problem to which the algorithm might be applied. Similarly, the literature does not always distinguish carefully between the FFT as a computational procedure and the discrete Fourier transform to which the FFT is applied, nor between the Viterbi algorithm as a computational procedure and the minimum-distance pathfinding problem to which the Viterbi algorithm is applied.

Problems for Chapter 1

1.1 Construct an algorithm for the two-point real cyclic convolution

$$(s_1 x + s_0) = (g_1 x + g_0)(d_1 x + d_0) \pmod{x^2 - 1}$$

that uses two multiplications and four additions. Computations involving only g_0 and g_1 need not be counted under the assumption that g_0 and g_1 are constants, and these computations need be done only once off-line.

1.2 Using the result of Problem 1.1, construct an algorithm for the two-point complex cyclic convolution that uses only six real multiplications.

1.3 Construct an algorithm for the three-point Fourier transform

$$V_k = \sum_{i=0}^{2} \omega^{ik} v_i, \qquad k = 0, 1, 2$$

that uses two real multiplications.

1.4 Prove that there does not exist an algorithm for multiplying two complex numbers that uses only two real multiplications. (A throughful "proof" will struggle with the meaning of the term "multiplication.")

1.5 **a** Suppose you are given a device that computes the linear convolution of two fifty-point sequences. Describe how to use it to compute the cyclic convolution of two fifty-point sequences.

b Suppose you are given a device that computes the cyclic convolution of two fifty-point sequences. Describe how to use it to compute the linear convolution of two fifty-point sequences.

1.6 Prove that one can compute a correlation as a convolution by writing one of the sequences backwards, possibly padding a sequence with a string of zeros.

1.7 Show that any algorithm for computing x^{31} that uses only additions, subtractions, and multiplications must use at least seven multiplications, but that if division is allowed, then an algorithm exists that uses a total of six multiplications and divisions.

1.8 Another algorithm for complex multiplication is given by

$$e = ac - bd,$$
$$f = (a + b)(c + d) - ac - bd.$$

Express this algorithm in a matrix form similar to that given in the text. What advantages or disadvantages are there in this algorithm?

1.9 Prove the "translation property" of the Fourier transform. If $\{v_j\} \leftrightarrow \{V_k\}$ is a Fourier transform pair, then the following are Fourier transform pairs:

$$\{\omega^i v_i\} \leftrightarrow \{V_{((k+1))}\},$$
$$\{v_{((i-1))}\} \leftrightarrow \{\omega^k V_k\}.$$

1.10 Prove that the "cyclic correlation" in the real field satisfies the Fourier transform relationship

$$G_k D_k^* \leftrightarrow \sum_{k=0}^{n-1} g_{((i+k))} d_k.$$

1.11 Given two real vectors $\boldsymbol{v}'$ and $\boldsymbol{v}''$, show how to recover their individual Fourier transforms from the Fourier transform of the sum vector $\boldsymbol{v} = \boldsymbol{v}' + \mathrm{j}\boldsymbol{v}''$.

Notes for Chapter 1

A good history of the origins of the fast Fourier transform algorithms is given in a paper by Cooley, Lewis, and Welch (1967). The basic theory of digital signal processing can be found in many books, including the books by Oppenheim and Shafer (1975), Rabiner and Gold (1975), and Proakis and Manolakis (2006).

Algorithms for complex multiplication using three real multiplications became generally known in the late 1950s, but the origin of these algorithms is a little hazy. The matrix multiplication algorithm we have given is due to Winograd (1968).

2 Introduction to abstract algebra

Good algorithms are elegant algebraic identities. To construct these algorithms, we must be familiar with the powerful structures of number theory and of modern algebra. The structures of the set of integers, of polynomial rings, and of Galois fields will play an important role in the design of signal-processing algorithms. This chapter will introduce those mathematical topics of algebra that will be important for later developments but that are not always known to students of signal processing. We will first study the mathematical structures of groups, rings, and fields. We shall see that a discrete Fourier transform can be defined in many fields, though it is most familiar in the complex field. Next, we will discuss the familiar topics of matrix algebra and vector spaces. We shall see that these can be defined satisfactorily in any field. Finally, we will study the integer ring and polynomial rings, with particular attention to the euclidean algorithm and the Chinese remainder theorem in each ring.

2.1 Groups

A group is a mathematical abstraction of an algebraic structure that appears frequently in many concrete forms. The abstract idea is introduced because it is easier to study all mathematical systems with a common structure at once, rather than to study them one by one.

Definition 2.1.1 *A group G is a set together with an operation (denoted by $*$) satisfying four properties.*

1 **(*Closure*)** *For every a and b in the set, $c = a * b$ is in the set.*

2 **(*Associativity*)** *For every a, b, and c in the set,*

$$a * (b * c) = (a * b) * c.$$

3 **(*Identity*)** *There is an element e called the identity element that satisfies*

$$a * e = e * a = a$$

for every a in the set G.

$*$	e	g_1	g_2	g_3	g_4
e	e	g_1	g_2	g_3	g_4
g_1	g_1	g_2	g_3	g_4	e
g_2	g_2	g_3	g_4	e	g_1
g_3	g_3	g_4	e	g_1	g_2
g_4	g_4	e	g_1	g_2	g_3

$+$	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

Figure 2.1 Example of a finite group

4 (*Inverses*) *If a is in the set, then there is some element b in the set called an inverse of a such that*

$$a * b = b * a = e.$$

A group that has a finite number of elements is called a *finite group*. The number of elements in a finite group G is called the *order* of G. An example of a finite group is shown in Figure 2.1. The same group is shown twice, but represented by two different notations. Whenever two groups have the same structure but a different representation, they are said to be *isomorphic*.[1]

Some groups satisfy the property that for all a and b in the group

$$a * b = b * a.$$

This is called the *commutative property*. Groups with this additional property are called *commutative groups* or *abelian groups*. We shall usually deal with abelian groups.

In an abelian group, the symbol for the group operation is commonly written $+$ and is called addition (even though it might not be the usual arithmetic addition). Then the identity element e is called "zero" and is written 0, and the inverse element of a is written $-a$ so that

$$a + (-a) = (-a) + a = 0.$$

Sometimes the symbol for the group operation is written $*$ and is called multiplication (even though it might not be the usual arithmetic multiplication). In this case, the identity element e is called "one" and is written 1, and the inverse element of a is written a^{-1} so that

$$a * a^{-1} = a^{-1} * a = 1.$$

[1] In general, any two algebraic systems that have the same structure but are represented differently are called *isomorphic*.

Theorem 2.1.2 *In every group, the identity element is unique. Also, the inverse of each group element is unique, and* $(a^{-1})^{-1} = a$.

Proof Let e and e' be identity elements. Then $e = e * e' = e'$. Next, let b and b' be inverses for element a; then

$$b = b * (a * b') = (b * a) * b' = b'$$

so $b = b'$. Finally, for any a, $a^{-1} * a = a * a^{-1} = e$, so a is an inverse for a^{-1}. But because inverses are unique, $(a^{-1})^{-1} = a$. □

Many common groups have an infinite number of elements. Examples are the set of integers, denoted $\boldsymbol{Z} = \{0, \pm 1, \pm 2, \pm 3, \ldots\}$, under the operation of addition; the set of positive rationals under the operation of multiplication;[2] and the set of two by two, real-valued matrices under the operation of matrix addition. Many other groups have only a finite number of elements. Finite groups can be quite intricate.

Whenever the group operation is used to combine the same element with itself two or more times, an exponential notation can be used. Thus $a^2 = a * a$ and

$$a^k = a * a * \cdots * a,$$

where there are k copies of a on the right.

A *cyclic group* is a finite group in which every element can be written as a power of some fixed element called a *generator* of the group. Every cyclic group has the form

$$G = \{a^0, a^1, a^2, \ldots, a^{q-1}\},$$

where q is the order of G, a is a generator of G, a^0 is the identity element, and the inverse of a^i is a^{q-i}. To actually form a group in this way, it is necessary that $a^q = a^0$. Because, otherwise, if $a^q = a^i$ with $i \neq 0$, then $a^{q-1} = a^{i-1}$, and there are fewer than q distinct elements, contrary to the definition.

An important cyclic group with q elements is the group denoted by the label $\boldsymbol{Z}/\langle q \rangle$, by $\boldsymbol{Z}_q$, or by $\boldsymbol{Z}/q\boldsymbol{Z}$, and given by

$$\boldsymbol{Z}/\langle q \rangle = \{0, 1, 2, \ldots, q-1\},$$

and the group operation is modulo q addition. In formal mathematics, $\boldsymbol{Z}/\langle q \rangle$ would be called a *quotient group* because it "divides out" multiples of q from the original group $\boldsymbol{Z}$.

For example,

$$\boldsymbol{Z}/\langle 6 \rangle = \{0, 1, 2, \ldots, 5\},$$

and $3 + 4 = 1$.

[2] This example is a good place for a word of caution about terminology. In a general abelian group, the group operation is usually called "addition" but is not necessarily ordinary addition. In this example, it is ordinary multiplication.

The group $\mathbf{Z}/\langle q\rangle$ can be chosen as a standard prototype of a cyclic group with q elements. There is really only one cyclic group with q elements; all others are isomorphic copies of it differing in notation but not in structure. Any other cyclic group G with q elements can be mapped into $\mathbf{Z}/\langle q\rangle$ with the group operation in G replaced by modulo q addition. Any properties of the structure in G are also true in $\mathbf{Z}/\langle q\rangle$, and the converse is true as well.

Given two groups G' and G'', it is possible to construct a new group G, called the *direct product*,[3] or, more simply, the *product* of G' and G'', and written $G = G' \times G''$. The elements of G are pairs of elements (a', a''), the first from G' and the second from G''. The group operation in the product group G is defined by

$$(a', a'') * (b', b'') = (a' * b', a'' * b'').$$

In this formula, $*$ is used three times with three meanings. On the left side it is the group operation in G, and on the right side it is the group operation in G' or in G'', respectively.

For example, for $\mathbf{Z}_2 \times \mathbf{Z}_3$, we have the set

$$\mathbf{Z}_2 \times \mathbf{Z}_3 = \{(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2)\}.$$

A typical entry in the addition table for $\mathbf{Z}_2 \times \mathbf{Z}_3$ is

$$(1, 2) + (0, 2) = (1, 1)$$

with $\mathbf{Z}_2$ addition in the first position and $\mathbf{Z}_3$ addition in the second position. Notice that $\mathbf{Z}_2 \times \mathbf{Z}_3$ is itself a cyclic group generated by the element (1, 1). Hence $\mathbf{Z}_2 \times \mathbf{Z}_3$ is isomorphic to $\mathbf{Z}_6$. The reason this is so is that two and three have no common integer factor. In contrast, $\mathbf{Z}_3 \times \mathbf{Z}_3$ is not isomorphic to $\mathbf{Z}_9$.

Let G be a group and let H be a subset of G. Then H is called a *subgroup* of G if H is itself a group with respect to the restriction of $*$ to H. As an example, in the set of integers (positive, negative, and zero) under addition, the set of even integers is a subgroup, as is the set of multiples of three.

One way to get a subgroup H of a finite group G is to take any element h from G and let H be the set of elements obtained by multiplying h by itself an arbitrary number of times to form the sequence of elements $h, h^2, h^3, h^4, \ldots$. The sequence must eventually repeat because G is a finite group. The first element repeated must be h itself, and the element in the sequence just before h must be the group identity element because the construction gives a cyclic group. The set H is called the *cyclic subgroup* generated by h. The number q of elements in the subgroup H satisfies $h^q = 1$, and q is called the *order* of the element h. The set of elements $h, h^2, h^3, \ldots, h^q = 1$ is called a *cycle* in the group G, or the *orbit* of h.

[3] If the group G is an abelian group, the direct product is often called the *direct sum*, and denoted $\oplus$. For this reason, one may also use the notation $\mathbf{Z}_2 \oplus \mathbf{Z}_3$.

To prove that a nonempty subset H of G is a subgroup of G, it is necessary only to check that $a * b$ is in H whenever a and b are in H and that the inverse of each a in H is also in H. The other properties required of a group will then be inherited from the group G. If the group is finite, then even the inverse property is satisfied automatically, as we shall see later in the discussion of cyclic subgroups.

Given a finite group G and a subgroup H, there is an important construction known as the *coset decomposition* of G that illustrates certain relationships between H and G. Let the elements of H be denoted by $h_1, h_2, h_3, \ldots$, and choose h_1 to be the identity element. Construct the array as follows. The first row consists of the elements of H, with the identity at the left and every other element of H appearing once. Choose any element of G not appearing in the first row. Call it g_2 and use it as the first element of the second row. The rest of the elements of the second row are obtained by multiplying each element in the first row by g_2. Then construct a third, fourth, and fifth row similarly, each time choosing a previously unused group element for the first element in the row. Continue in this way until, at the completion of a row, all the group elements appear somewhere in the array. This occurs when there is no unused element remaining. The process must stop, because G is finite. The final array is

$$\begin{array}{cccccc}
h_1 = 1 & h_2 & h_3 & h_4 & \cdots & h_n \\
g_2 * h_1 = g_2 & g_2 * h_2 & g_2 * h_3 & g_2 * h_4 & \cdots & g_2 * h_n \\
g_3 * h_1 = g_3 & g_3 * h_2 & g_3 * h_3 & g_3 * h_4 & \cdots & g_3 * h_n \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
g_m * h_1 = g_m & g_m * h_2 & g_m * h_3 & g_m * h_4 & \cdots & g_m * h_n.
\end{array}$$

The first element on the left of each row is known as a *coset leader*. Each row in the array is known as a *left coset*, or simply as a *coset* when the group is abelian. If the coset decomposition is defined instead with the new elements of G multiplied on the right, the rows are known as *right cosets*. The coset decomposition is always rectangular with all rows completed because it is constructed that way. The next theorem says that every element of G appears exactly once in the final array.

Theorem 2.1.3 *Every element of G appears once and only once in a coset decomposition of G.*

Proof Every element appears at least once because, otherwise, the construction is not halted. We now prove that an element cannot appear twice in the same row and then prove that an element cannot appear in two different rows.

Suppose that two elements in the same row, $g_i * h_j$ and $g_i * h_k$, are equal. Then multiplying each by g_i^{-1} gives $h_j = h_k$. This is a contradiction because each subgroup element appears only once in the first row.

Suppose that two elements in different rows, $g_i * h_j$ and $g_k * h_\ell$, are equal with k less than i. Multiplying on the right by h_j^{-1} gives $g_i = g_k * h_\ell * h_j^{-1}$. Then g_i is in the kth

coset, because $h_\ell * h_j^{-1}$ is in the subgroup H. This contradicts the rule of construction that coset leaders must be previously unused. Thus the same element cannot appear twice in the array. □

Corollary 2.1.4 *If H is a subgroup of G, then the number of elements in H divides the number of elements in G. That is,*

(Order of H) (Number of cosets of G with respect to H) = (Order of G).

Proof Follows immediately from the rectangular structure of the coset decomposition. □

Theorem 2.1.5 (Lagrange) *The order of a finite group is divisible by the order of any of its elements.*

Proof The group contains the cyclic subgroup generated by any element; then Corollary 2.1.4 proves the theorem. □

Corollary 2.1.6 *For any a in a group with q elements, $a^q = 1$.*

Proof By Theorem 2.1.5, the order of a divides q, so $a^q = 1$. □

2.2 Rings

The next algebraic structure we will need is that of a ring. A ring is an abstract set that is an abelian group and also has an additional operation.

Definition 2.2.1 *A ring R is a set with two operations defined – the first called addition (denoted by $+$) and the second called multiplication (denoted by juxtaposition) – and the following axioms are satisfied.*

1 R is an abelian group under addition.

2 (Closure) For any a, b in R, the product ab is in R.

3 (Associativity)

$$a(bc) = (ab)c.$$

4 (Distributivity)

$$a(b + c) = ab + ac$$
$$(b + c)a = ba + ca.$$

The distributivity property in the definition of a ring links the addition and multiplication operations. The addition operation is always required to be commutative in a

ring, but the multiplication operation need not be commutative. A *commutative ring* is one in which multiplication is commutative; that is, $ab = ba$ for all a, b in R.

Some important examples of rings are the ring of integers, denoted $\boldsymbol{Z}$, and the ring of integers under modulo q arithmetic, denoted $\boldsymbol{Z}/\langle q\rangle$. The ring $\boldsymbol{Z}/\langle q\rangle$ is an example of a quotient ring because it uses modulo q arithmetic to "divide out" q from $\boldsymbol{Z}$. We have already seen the various $\boldsymbol{Z}/\langle q\rangle$ as examples of groups under addition. Because there is also a multiplication operation that behaves properly on these sets, they are also rings. Another example of a ring is the set of all polynomials in x with rational coefficients. This ring is denoted $\boldsymbol{Q}[x]$. It is easy to verify that $\boldsymbol{Q}[x]$ has all the properties required of a ring. Similarly, the set of all polynomials with real coefficients is a ring, denoted $\boldsymbol{R}[x]$. The ring $\boldsymbol{Q}[x]$ is a subring of $\boldsymbol{R}[x]$. The set of all two by two matrices over the reals is an example of a noncommutative ring, as can be easily checked.

Several consequences that are well-known properties of familiar rings are implied by the axioms as can be proved as follows.

Theorem 2.2.2 *For any elements a, b in a ring R,*
(i) $a0 = 0a = 0$*;*
(ii) $a(-b) = (-a)b = -(ab)$.

Proof
(i) $a0 = a(0 + 0) = a0 + a0$. Hence adding $-a0$ to both sides gives $0 = a0$. The second half of (i) is proved the same way.
(ii) $0 = a0 = a(b - b) = ab + a(-b)$. Hence

$$a(-b) = -(ab).$$

The second half of (ii) is proved the same way. □

The addition operation in a ring has an identity called "zero." The multiplication operation need not have an identity, but if there is an identity, it is unique. A ring that has an identity under multiplication is called a *ring with identity*. The identity is called "one" and is denoted by 1. Then

$$1a = a1 = a$$

for all a in R.

Every element in a ring has an inverse under the addition operation. Under the multiplication operation, there need not be any inverses, but in a ring with identity, inverses may exist. That is, given an element a, there may exist an element b with $ab = 1$. If so, b is called a *right inverse* for a. Similarly, if there is an element c such that $ca = 1$, then c is called a *left inverse* for a.

Theorem 2.2.3 *In a ring with identity:*

(*i*) *The identity is unique.*

(*ii*) *If an element a has both a right inverse b and a left inverse c, then* $b = c$. *In this case the element a is said to have an inverse* (*denoted* a^{-1}). *The inverse is unique.*

(*iii*) $(a^{-1})^{-1} = a$.

Proof The proof is similar to that used in Theorem 2.1.2. □

The identity of a ring, if there is one, can be added to itself or subtracted from itself any number of times to form the doubly infinite sequence

$$\ldots, -(1+1+1), -(1+1), -1, 0, 1, (1+1), (1+1+1), \ldots.$$

These elements of the ring are called the *integers* of the ring and are sometimes denoted more simply as $0, \pm 1, \pm 2, \pm 3, \pm 4, \ldots$. There may be a finite number or an infinite number of integers. The number of integers in a ring with identity, if this is finite, is called the *characteristic* of the ring. If the characteristic of a ring is not finite then, by definition, the ring has characteristic zero. If the characteristic is the finite number q, then we can write the integers of the ring as the set

$$\{1, 1+1, 1+1+1, \ldots\},$$

denoting these more simply with the notation of the usual integers

$$\{1, 2, 3, \ldots, q-1, 0\}.$$

This subset is a subgroup of the additive group of the ring; in fact, it is the cyclic subgroup generated by the element one. Hence addition of the integers of a ring is modulo q addition whenever the characteristic of the ring is finite. If the characteristic is infinite, then the integers of the ring add as integers. Hence every ring with identity contains a subset that behaves under addition either as $\boldsymbol{Z}$ or as $\boldsymbol{Z}/\langle q \rangle$. In fact, it also behaves in this way under multiplication, because if α and β are each a finite sum of the ring identity one, then

$$\alpha * \beta = \beta + \beta + \cdots + \beta,$$

where there are α copies of β on the right. Because addition of integers in R behaves like addition in $\boldsymbol{Z}$ or in $\boldsymbol{Z}/\langle q \rangle$, then so does multiplication of integers in R.

Within a ring R, any element α can be raised to an integer power; the notation α^m simply means the product of m copies of α. If the ring has an identity and the number of integers of the ring is a prime, then the following theorem is sometimes useful for simplifying powers of sums.

Theorem 2.2.4 *Let p be a prime, and let R be a commutative ring with p integers. Then for any positive integer m and for any two elements* α *and* β *in R,*

$$(a \pm \beta)^{p^m} = \alpha^{p^m} \pm \beta^{p^m},$$

and by direct extension,

$(\Sigma_i \alpha_i)^{p^m} = \Sigma_i \alpha_i^{p^m}$

for any set of α_i *in* R.

Proof The binomial theorem,

$$(a \pm \beta)^p = \sum_{i=0}^{p} \binom{p}{i} \alpha^i (\pm\beta)^{p-i}$$

holds in the ring of real numbers, so it must also hold in any commutative ring with identity because the coefficient $\binom{p}{i}$ merely counts how many terms of the expansion have i copies of α and $p - i$ copies of β. This does not depend on the specific ring. However, $\binom{p}{i}$ is interpreted in R as a sum of this many copies of the ring identity. Recall that in a ring with p integers, all integer arithmetic is modulo p. Next, observe that

$$\binom{p}{i} = \frac{p!}{i!(p-i)!} = \frac{p(p-1)!}{i!(p-i)!}$$

is an integer and p is a prime. Hence, as integers, the denominator divides $(p - 1)!$ for $i = 1, \ldots, p - 1$, and so $\binom{p}{i}$ must be a multiple of p. That is, $\binom{p}{i} = 0 \pmod{p}$ for $i = 1, \ldots, p - 1$. Then $(\alpha \pm \beta)^p = \alpha^p + (\pm\beta)^p$. Finally, either $p = 2$ and $-\beta = \beta$ so that $(\pm\beta)^2 = \pm\beta^2$, or p is odd and $(\pm\beta)^p = \pm\beta^p$. Then

$$(a \pm \beta)^p = \alpha^p \pm \beta^p,$$

which proves the theorem for $m = 1$.

This now can again be raised to the pth power,

$$((\alpha \pm \beta)^p)^p = (\alpha^p \pm \beta^p)^p,$$

and because the statement is true for $m = 1$, we have

$$(\alpha^p \pm \beta^p)^p = \alpha^{p^2} \pm \beta^{p^2}.$$

This can be repeated multiple times to get

$$(\alpha \pm \beta)^{p^m} = \alpha^{p^m} \pm \beta^{p^m},$$

which completes the proof of the theorem. □

An element that has an inverse under multiplication in a ring with identity is called a *unit* of the ring. The set of all units is closed under multiplication, because if a and b are units, then $c = ab$ has the inverse $c^{-1} = b^{-1}a^{-1}$.

Theorem 2.2.5

(*i*) *Under ring multiplication, the set of units of a ring forms a group.*

(*ii*) *If* $c = ab$ *and* c *is a unit, then* a *has a right inverse and* b *has a left inverse.*

(*iii*) *If* $c = ab$ *and* a *has no right inverse or* b *has no left inverse, then* c *is not a unit.*

Proof Exercise. □

There are many familiar examples of rings such as the following.

1 The set of all real numbers under the usual addition and multiplication is a commutative ring with identity. Every nonzero element is a unit.
2 The set $\boldsymbol{Z}$ of all integers under the usual addition and multiplication is a commutative ring with identity. The only units are ± 1.
3 The set of all n by n matrices with real-valued elements under matrix addition and matrix multiplication is a noncommutative ring with identity. The identity is the n by n identity matrix. The units are the nonsingular matrices.
4 The set of all n by n matrices with integer-valued elements under matrix addition and matrix multiplication is a noncommutative ring with identity.
5 The set of all polynomials in x with real-valued coefficients is a commutative ring with identity under polynomial addition and polynomial multiplication. The identity is the zero-degree polynomial $p(x) = 1$. The units are the nonzero polynomials of degree zero.

2.3 Fields

Loosely speaking, an abelian group is a set in which one can "add" and "subtract," and a ring is a set in which one can "add," "subtract," and "multiply." A more powerful algebraic structure, known as a field, is a set in which one can "add," "subtract," "multiply," and "divide."

Definition 2.3.1 *A field F is a set containing at least two elements that has two operations defined on it – addition and multiplication – such that the following axioms are satisfied:*

1 the set is an abelian group under addition;
2 the set is closed under multiplication, and the set of nonzero elements is an abelian group under multiplication;
3 the distributive law,

$$(a + b)c = ac + bc,$$

holds for all a, b, and c in the field.

In a field, it is conventional to denote the identity element under addition by 0 and to call it zero; to denote the additive inverse of a by $-a$; to denote the identity element under multiplication by 1 and to call it one; and to denote the multiplicative inverse of a by a^{-1}. By subtraction $(a - b)$, one means $a + (-b)$; by division (a/b), one means $b^{-1}a$.

The following examples of fields are well known.

1 $\boldsymbol{R}$: the set of real numbers.

2 $\boldsymbol{C}$: the set of complex numbers.

3 $\boldsymbol{Q}$: the set of rational numbers.

These fields all have an infinite number of elements. There are many other, less-familiar fields with an infinite number of elements. One that is easy to describe is known as the field of *complex rationals*, denoted $\boldsymbol{Q}(\mathrm{j})$. It is given by

$$\boldsymbol{Q}(\mathrm{j}) = \{a + \mathrm{j}b\},$$

where a and b are rationals. Addition and multiplication are as complex numbers. With this definition, $\boldsymbol{Q}(\mathrm{j})$ satisfies the requirements of Definition 2.3.1, and so it is a field.

There are also fields with a finite number of elements, and we have uses for these as well. A field with q elements, if there is one, is called a *finite field* or a *Galois field* and is denoted by the label $GF(q)$.

Every field must have an element zero and an element one, thus every field has at least two elements. In fact, with the addition and multiplication tables,

$+$	0	1
0	0	1
1	1	0

$\cdot$	0	1
0	0	0
1	0	1

,

these elements suffice to form a field. This is the field known as $GF(2)$. No other field with two elements exists.

Finite fields can be described by writing out their addition and multiplication tables. Subtraction and division are defined implicitly by the addition and multiplication tables. Later, we shall study finite fields in detail. Here we give three more examples.

The field $GF(3) = \{0, 1, 2\}$ with the operations

$+$	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

$\cdot$	0	1	2
0	0	0	0
1	0	1	2
2	0	2	1

.

The field $GF(4) = \{0, 1, 2, 3\}$ with the operations

$+$	0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

$\cdot$	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	3	1
3	0	3	1	2

.

Notice that multiplication in $GF(4)$ is *not* modulo 4 multiplication, and addition in $GF(4)$ is *not* modulo 4 addition. The field $GF(5) = \{0, 1, 2, 3, 4\}$ with the