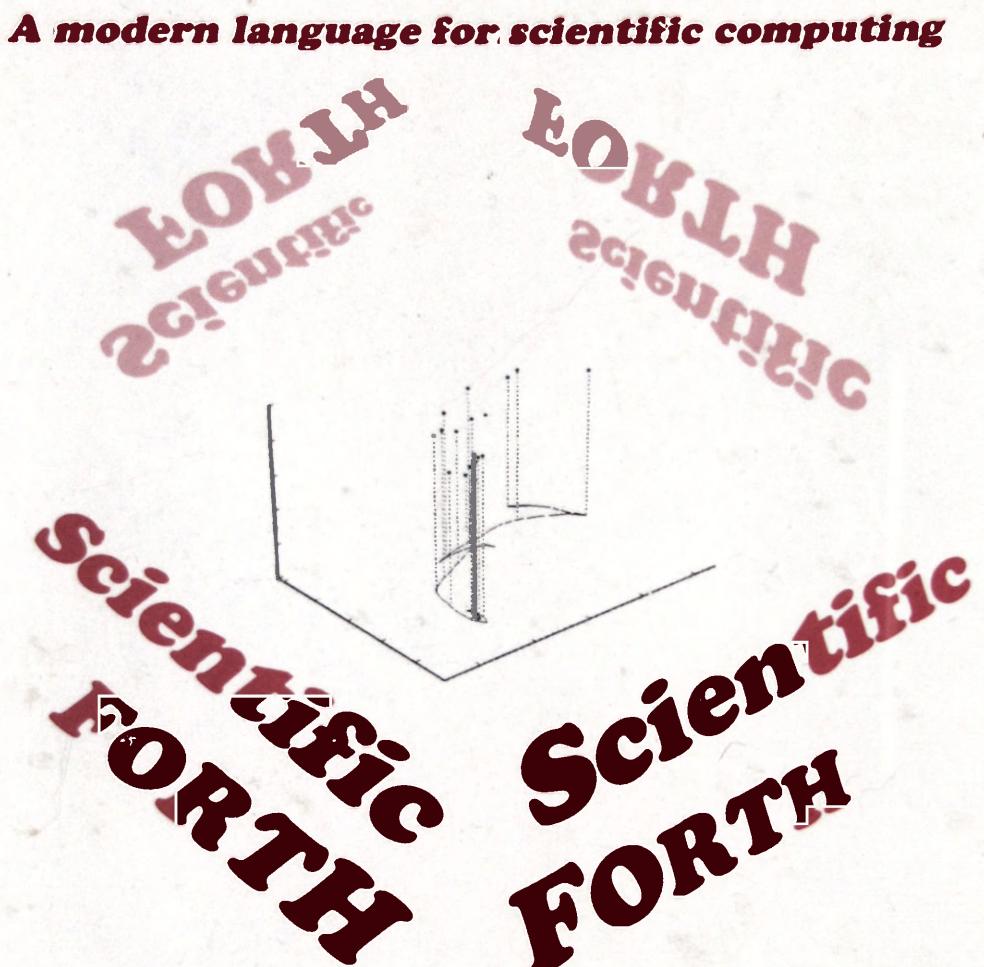


A modern language for scientific computing



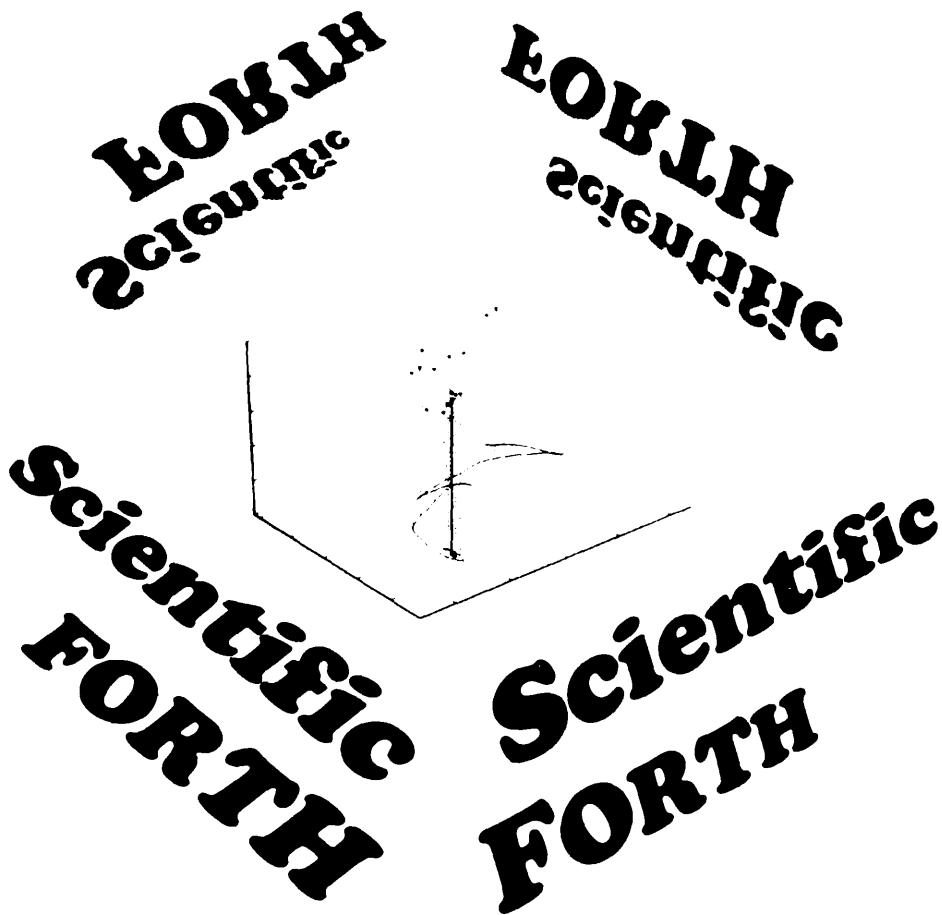
A modern language for scientific computing

Julian V. Noble
Professor of Physics, University of Virginia



Mechum Banks Publishing

A modern language for scientific computing



A modern language for scientific computing

Julian V. Noble
Professor of Physics, University of Virginia



Mechum Banks Publishing
P.O. Box 335
Ivy, Virginia 22945

Scientific FORTH

a modern language for scientific computing

Julian V. Noble

Published by:

Mechum Banks Publishing
P.O. Box 335
Charlottesville, VA 22945 USA

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system without written permission from the author, except for the inclusion of brief quotations in a review.

Copyright © 1992 by Julian V. Noble
Printed in the United States of America

Publication Data

Noble, Julian V.

Scientific FORTH: a modern language for scientific computing / by Julian V. Noble

Bibliography: p. 310

Includes Index.

1. FORTH (Computer Program Language).
2. Electronic digital computers—Programming.
3. Computer science. I. Title.

ISBN 0-9632775-0-2

Warning and Disclaimer of Warranty

FORTH is an extremely powerful language. It gives you complete control over every part of the computer. Therefore improperly used FORTH has the capability of causing damage, both to data and to devices. (This capacity is not unique to FORTH – recently a commercial program written in C crashed, wiping out the configuration settings on my computer and leaving the machine paralyzed and unable to boot.)

Because the components of FORTH programs can be tested as they are written, FORTH systems tend to omit many safety features – particularly bounds checking – found in more conventional languages. This omission is one of the ways FORTH achieves its execution speed. (Safety features are easy to add, however: FORTH programmers often do so during the debugging stages of a project.)

FORTH is easy to modify. The compiler is part of the language; FORTH systems always include an assembler for the target machine; hence FORTH dialects abound.

These aspects of FORTH –great power and lack of safety features, together with its mutability – preclude the author and publisher from being able to warrant the illustrative code and programs listed in this book. The author has used his best efforts in preparing this book, to provide code that works as intended.

However, the author and publisher make no warranty of any kind, express or implied, with regard to these programs or their suitability for any specific purpose.

The author and publisher shall not be liable in any event for incidental or consequential damages arising out of the furnishing, performance or use of these programs and examples.

II

Scientific FORTH

Scientific FORTH – Contents

Warning and Disclaimer of Warranty	I
Contents	III-VII
Preface	I-vii
Chapter 1 – Toward Scientific FORTH	1
§1 Overview of FORTRAN	2
§§1 Programs and sub-programs	2
§§2 Arithmetic statements	3
§§3 Function library	6
§2 What is FORTH ?	8
Chapter 2 – Programming In FORTH	11
§1 The structure of FORTH	13
§2 Extending the dictionary	15
§3 Stacks and reverse Polish notation (RPN)	17
§§1 Manipulating the parameter stack	19
§§2 The return stack and its uses	21
§4 Fetching and storing	23
§5 Arithmetic operations	24
§6 Comparing and testing	24
§7 Looping and structured programming	25
§8 The pearl of FORTH	26
§§1 Dummy words	27
§§2 Defining “defining” words	28
§§3 Run-time vs. compile-time actions	30
§§4 Advanced methods of controlling the compiler	34
§9 Strings	36
§10 FORTH programming style	38
§§1 Structure	38
§§2 “Top-down” design	39
§§3 Information hiding	40
§§4 Documenting and commenting FORTH code	42
§§5 Safety	44

Chapter 3 – Floating Point Arithmetic	45
§1 Organization of floating point arithmetic	46
§§1 Fstack manipulation	47
§§2 Special constants	48
§§3 Arithmetic operators	49
§§4 Example: evaluating a polynomial	50
§§5 Optimizing: FORTH vs. FORTRAN	51
§2 Testing floating point numbers	53
§3 Mathematical functions – the essential function library	53
§4 Library extensions	54
§§1 The FSGN function	54
§§2 Cosh, Sinh and their inverses	54
§5 Pseudo-random number generators (PRNG's)	56
§§1 Testing random number generators	58
§§2 Random data structures	61
Chapter 4 – The 80x87 Family	65
§1 Internal 80x87 stack manipulation	67
§§1 The FORTH assembler	67
§§2 Using MS-DOS DEBUG	68
§2 Memory usage (storage and retrieval)	70
§3 Arithmetic words	72
§4 Special constants	73
§5 Test words	74
§6 Mathematical functions	76
§7 Extending the intrinsic 80x87 stack	80
§8 Clone wars	84
Chapter 5 – Scientific Data Structures	91
§1 Typed data structures	92
§§1 Type descriptors	94
§§2 Typed scalars	94
§§3 Defining several scalars at once	95
§§4 Generic access	97
§§5 The intelligent floating point stack (ifstack)	99
§§6 Unary and binary generic operators	100

Scientific FORTH

§2 Arrays of typed data	104
§§1 Improved (FORTRAN-like) array notation	105
§§2 Large matrices	106
§§3 Using high memory	108
§§4 A general typed-array definition	110
§§5 2ARRAY and { }	113
§3 Tuning for speed	114
 Chapter 6 – Programming Examples	115
§1 Infinite series	116
§§1 Examples of infinite series	116
§§2 Numerical examples of convergent series	118
§§3 The infinite sum program	122
§2 Transcendental equations	127
§§1 Binary search	127
§§2 Regula falsi	128
§3 Ordinary differential equations	131
§§1 Runge-Kutta method	132
§§2 An implicit Runge-Kutta formula	136
 Chapter 7 – Complex arithmetic in FORTH	143
§1 The complex number system	144
§§1 Cartesian representation of complex numbers	144
§§2 Polar representation of complex numbers	147
§2 Load, store, manipulate fstack	148
§3 Arithmetic operations	149
§4 Roots of complex numbers	151
§§1 Complex square roots	152
§§2 The complex square root program	154
§5 Complex exponentials and trigonometric functions	154
§6 Logarithms	155

Chapter 8 – More Programming Examples	157
§1 Numerical integration	158
§§1 The integral of a function	158
§§2 The fundamental theorem of calculus	160
§§3 Monte-Carlo method	161
§§4 Adaptive methods	165
§§5 Adaptive integration on the real line	165
§§6 Adaptive integration in the Argand plane	179
§2 Fitting functions to data	181
§§1 Fast Fourier transform	184
§§2 Gram polynomials	193
§§3 Simplex algorithm	201
§3 Appendices	204
§§1 Gaussian quadrature	204
§§2 The trapezoidal rule	205
§§3 Richardson extrapolation	206
§§4 Linear least-squares: Gram polynomials	207
§§5 Non-linear least squares: simplex method	208
Chapter 9 – Linear Algebra	213
§1 Simultaneous linear equations	214
§§1 Theory of linear algebraic equations	214
§§2 Eigenvalue problems	215
§2 Solving linear equations	218
§§1 Cramer's rule	218
§§2 Pivotal Elimination	221
§§3 Testing	225
§§4 Implementing pivotal elimination	227
§§5 The program	227
§§6 Timing	238
§3 Matrix inversion	242
§§1 Linear transformations	242
§§2 Matrix multiplication	243
§§3 Matrix inversion	244
§§4 Why invert matrices, anyway?	245
§§5 An example	245
§4 LU decomposition	246

Scientific FORTH

Chapter 10 – Strings and I/O in FORTH	249
§1 Standard I/O in FORTH	249
§2 Standard I/O in Scientific FORTH	250
§§1 Disk file input	251
§§2 Disk file output	253
§3 Logging screen activity	254
§§1 Redirection by re-vectoring	254
§§2 Redirection using MS-DOS resources	255
§§3 Redirection while DEBUGging	257
Chapter 11 – Symbolic Programming	259
§1 Rules	260
§2 Tools	262
§§1 Pattern recognizers	262
§§2 Finite state machines	265
§§3 FSMs in FORTH	268
§§4 Automatic conversion tables	271
§3 Computer algebra	273
§§1 Stating the problem	273
§§2 The rules	275
§§3 The program	276
§4 FORmula TRANslator	284
§§1 Rules of FORTRAN	285
§§2 Details of the Problem	286
§§3 Parsing	289
§§4 Coding the FORmula TRANslator	296
Index	301
Bibliography	310

Preface

This book, *Scientific FORTH*, had its genesis in a sabbatical year in Europe, where I tried with extremely limited success to perform several calculations using a FORTRAN compiler on a (trans)portable PC clone that represented my main computing resource. The glacial pace of the multi-pass compiler, the inadequacy of its function library, and the balkiness of the machine, combined to make programming feel like a fly struggling in a spider web.

Luckily I had recently obtained a fairly complete FORTH system, HS/FORTH[†] and, at some point when my patience was exhausted, resorted to it in desperation. FORTH turned my sabbatical around and made it fruitful: once past the rudiments of the new language, I never looked back. For me, FORTRAN had joined the dinosaurs.

Apparently FORTH is much better known and more widely appreciated among European scientists than among Americans. On this side of the Atlantic FORTRAN remains the mainstay of scientific computation. While some argue that FORTRAN only retains its hold through its tremendous base (and investment) in existing well-debugged code, this cannot be the sole reason why new programs are written in FORTRAN; why new versions of FORTRAN are developed; or why FORTRAN (or a FORTRAN-like interface) is supplied with each new machine, even exotic ones like array- and parallel processors.

† © Harvard Softworks, P.O. Box 69, Springboro, OH 45066

FORTRAN remains popular among scientists and engineers because it provides most of the functionality needed for technical applications — capabilities simply not found in more modern and/or “rational” languages. Despite its manifold deficiencies **FORTRAN** suits the majority of scientific computation.

Post-**FORTRAN** languages embody precepts of portability, structure, modularity and information-hiding. They generally permit self-reference (recursion), dynamic storage allocation and definition of new data types. But languages like C, Pascal, Modula-2 and BASIC cannot deal with complex numbers, nor can they be extended easily to do so. Neither do these languages permit graceful machine-code programming of selected functions, something one must occasionally do to augment a function library, or when tuning for ultimate speed.

Scientific computation deserves a language that is easy to program, debug and maintain; that embodies the desirable features of modern structured languages without sacrificing the best parts of **FORTRAN**.

My ideal language would simplify linking with machine code and with compiled subroutines (from other languages).

It would execute quickly while using memory parsimoniously (clearly, the more memory devoted to program, the less available for data).

And finally, this new language would be extensible both to new data structures and to new operations, so it could grow and evolve without having to be redesigned.

Despite its small size and simple structure, **FORTH** has all but one of the characteristics of my ideal scientific programming language: In its pristine form **FORTH** lacks any resemblance to **FORTRAN** either in structure or functionality. Fortunately, **FORTH** is an easily extensible language. This book presents the **FORTH** extensions I have developed over the past six years, the sum being a dialect one might call “Scientific **FORTH**”.

FORTH is not usually encountered within the context of scientific or engineering computation, although most users of personal computers or workstations have unwittingly experienced it in one form or another. FORTH has been called "one of the best-kept secrets in computing". It lurks unseen in automatic bank teller machines, computer games, industrial control devices and robots. The specialized printing language, PostScript®, is a dialect of FORTH, as is the ASYST® language for laboratory automation. Commercial products such as the VP-Planner® spreadsheet and Peachtree Accounting® are written in FORTH.

Some scientists and engineers have gained familiarity with FORTH because it is fast, compact, and easily debugged; and because it simplifies interfacing microprocessors with machines and laboratory equipment (FORTH was invented for just that purpose).

FORTH has not found great favor in number-crunching applications, however. There are several reasons for this:

- FORTH's creator, Charles Moore, preferred integer to floating point arithmetic for speed of execution. His successors mostly followed this philosophy. However, the large dynamical range of many physical problems precludes integer arithmetic: **Scientific computation demands floating point arithmetic.**
- Systems designers have avoided porting FORTH to mainframes: FORTH permits such intimate hardware control that irresponsible users can easily misuse the machine. Operating systems that maintain the necessary discipline are hard to design without altering the very nature of the language. (Nevertheless FORTH is now available on many multiuser systems such as VAX's, PRIMES and AT&T's.)
- FORTH is a tool (metalanguage) for constructing applications languages. Thus the kernel supplied by vendors is standard (within the confines of a given dialect) but any extensions —whether vendor-supplied or user-created— tend to be wildly variable even when the end result is the same. This variability hinders software portability and interchange.

Recent developments have improved somewhat this state of affairs, removing many objections to FORTH:

- Floating point hardware (including transcendental functions) is now available even for microcomputers, thereby eliminating the speed advantage of integer arithmetic.
- Microcomputers now rival the speed and memory of mainframes, at far lower cost per user. Thus serious scientific calculations can be undertaken on single-user workstations and the problem of misuse becomes irrelevant.
- Efforts to produce an ANSI standard FORTH are underway; a final draft version should be available concurrently with this book.
- This book, *Scientific FORTH*, provides a clear notational and stylistic standard for scientific and technical computing, together with programming examples and useful programs.
- In conjunction with the new ANSI Standard FORTH, this book can simplify cooperative program development and program interchange.

Every operation that FORTRAN is capable of can be programmed easily in FORTH. For example, the EXTERNAL specification of FORTRAN has its FORTH analogue in “vectoring”.

But FORTH has the ability not only to reproduce all the functionality of FORTRAN —using less memory, compiling much faster and often executing faster also—but to do things that FORTRAN could not accomplish easily or even at all.

This power comes at a price, however—FORTH normally incorporates little error checking, especially bounds checking on arrays, hence it is easy to overwrite the operating system in most computers, causing a computer crash or worse. The responsibility to avoid such problems lies entirely with the programmer.

One reason FORTH has not yet realized its potential in scientific computing is that scientists and programmers tend to reside in orthogonal communities, so that no one has until now troubled

to write the necessary extensions. One aim of this book is to provide such extensions in a form I hope will prove appealing to current FORTRAN users.

Since time and chance happen[†] to everything, even FORTH, I have devoted considerable effort to explaining the algorithms and ideas behind these extensions, as well as their nuts and bolts.

This book is intended primarily for users with some familiarity with FORTH — perhaps gained in robotics, interfacing or microprocessor-controlled machinery — who had not suspected how useful the language could be for scientific problem-solving. I hope it will also prove useful to scientists who need to do things beyond FORTRAN's scope, but who find C restrictive, unesthetic and hard to learn.

Chapter 2 briefly reviews standard FORTH, so readers whose main motivation is curiosuty will be able to understand the program fragments given later on. It is not intended as a FORTH tutorial. To learn FORTH, get hold of a FORTH system and play with it while consulting one or more of the excellent references on the subject.

Chapter 3 is devoted to floating point arithmetic and the floating point stack. In it we develop standard library of functions that duplicates FORTRAN's capabilities. The functions include three random number generators, with some simple tests thereof. The problem of generating random numbers from an arbitrary distribution is made the framework of a new kind of data structure, akin to the object familiar from SMALLTALK and C++ programming languages.

Chapter 4 describes how to extend FORTH on a machine that can use a floating-point arithmetic co-processor, to produce a full lexicon of mathematical functions — equivalent to those found

†

Ecclesiastes, 9.11.

M. Kelly and N. Spies, *FORTH, a Text and Reference* (Prentice-Hall, Englewood Cliffs, N.J., 1986). L. Brodie, *Starting FORTH* (Prentice-Hall, Englewood Cliffs, N.J., 1981).

in the built-in libraries of other languages. We illustrate how the system debugger/assembler routines can be used to develop machine-code co-processor routines. While specific to the Intel iapx87 chips, these techniques could obviously be applied to newer families such as the MC68881 or NS32081 chips.

In Chapter 5 we devise data structures useful in scientific work, including typed data, arrays and user stacks. The object is to achieve a balance between ease of use *a la* FORTRAN, and making the operations so "smart" they slow up the machine or become inflexible. The chapter proposes several methods of making array addressing more readable, as well as a generic technique for "fetch" and "store" of scientifically useful data types.

Chapter 6 gives programming examples using floating point techniques. The examples are: summing infinite series; solving transcendental equations; and solving differential equations.

Because scientific work makes such heavy use of complex arithmetic, Chapter 7 extends the floating point functions and operations to the complex number field.

Chapter 8 is devoted to additional programming examples that illustrate the scientific data structures of Chapters 6 and 7. These include numerical quadrature, contour integration and fast Fourier transform (FFT). The quadrature programs are adaptive, and both recursive and non-recursive versions are given.

Linear algebra is such a standard problem in scientific work that we give it its own chapter (9). We illustrate the data structures developed in Chapter 7 with programs to solve linear equations and invert matrices. The optimized version of the program will solve 350 simultaneous linear equations (in single precision) in about 16 minutes on a 10 MHz 8086/8087 machine.

Chapter 9 further serves to illustrate the development-test-debug cycle in a FORTH system. The first-pass programs actually work, although not optimized for speed or elegance. Special cases for timing and precision are included to ensure that variants developed by the reader also work. We also discuss optimization

and illustrate the principles of inner-loop optimization with a detailed analysis of execution time.

Chapter 10 is a very brief look at file handling. This aspect of FORTH is non-standard, and each commercially available system will have its own methods of tackling the problem of reading data from a disk- (or other device-) file, and then of storing the output back in a file. The methods given in Chapter 10 assume a file-based version of FORTH (such as HS/FORTH) running under MS-DOS. The subroutines are written in high-level FORTH to maximize the chance they will be adaptable to FORTHS other than HS/FORTH.

Chapter 11 describes a simple FORmula TRANslator, that translates FORTRAN arithmetic assignment statements into corresponding FORTH. In developing the translator we shall explore more advanced programming techniques such as finite state machines and recursive-descent parsing. Chapter 11 also applies these techniques to computer algebra.

Toward Scientific FORTH

Contents

§1 Overview of FORTRAN	2
§§1 Programs and sub-programs	2
§§2 Arithmetic statements	3
§§3 Function library	7
§2 What is FORTH ?	8

This book presents extensions to the FORTH programming language suitable for scientific and technical computation. The aim is to retain FORTRAN's good points while taking advantage of the simplicity, flexibility, extensibility, and control offered by FORTH.

The resulting dialect has many advantages over more traditional languages, both for small, casual, throw-away programs as well as for large, complex projects. FORTH lends itself to many programming styles, including procedural, object-oriented or event-driven. Its speed and economical use of memory suit FORTH for real-time, on-line data pre-processing as well as off-line analysis and computation.

Because FORTH is a **threaded, interpretive language**¹ its structure and philosophy differ radically from those of traditional languages like FORTRAN and BASIC. This introductory chapter

1. This description refers to FORTH's compilation scheme. See, e.g., R.G. Loeliger, *Threaded Interpretive Languages* (Byte Publications, Inc., Peterborough, NH, 1981). We shall have more to say about it in Chapter 2.

explains some of the differences by contrasting FORTRAN with FORTH.

§1 Overview of FORTRAN

FORTRAN is a compiled high level language². The programmer writes a source code program using FORTRAN's grammatical rules, data structures and operators; a special computer program (the compiler) then translates the source into a (relocatable) machine language version (object code). Another machine language program (the linker) then links modules of object code into an executable program that can be run under the control of the operating system of the computer.

Compilation produces executable programs that run fast, without the tedium of writing them directly in machine code (or assembly language). The source code will run virtually the same on any machine for which a compiler exists. That is, the source code is portable. Among other things, portability makes possible the development of standard libraries of reusable code for performing standard tasks like solving linear equations or computing Bessel functions.

The chief disadvantage of compilation is its tedium. Testing small portions of a program in isolation is virtually impossible — either an “exercise” program must be written and compiled with the module being tested, or else the entire program must be compiled as a unit. This process is so time-consuming it discourages fine-grained decomposition of programs into small, comprehensible components.

§§1 Programs and sub-programs

A FORTRAN program consists of a master, or main program that either stands alone or can call (transfer control to) sub-programs. Sub-programs fall into two classes: subroutines and

2. Although some interpreted FORTRANs such as WATFOR have been developed.

functions. Both receive **arguments** (input) from the calling program; they differ in how **results** (output) are returned to the calling program. Subroutines are called by the phrase

CALL SUB1(A,B,RESULT)

where **A** and **B** are arguments and **RESULT** is the result (which is returned in the argument list). By contrast, a function is called by having its name placed in an arithmetic expression. When the expression is evaluated, the value of the function (at its given arguments) is inserted in the expression where the function name appeared. That is, we might have a phrase like

OPSIDE = HYPOT*SIN(3.14159*ANGLE/180.).

Here the argument of **SIN** is also an expression which must be evaluated before being passed to the **SIN** subroutine. When **SIN** is evaluated, its value is returned, multiplied by **HYPOT** and the product stored in the area labelled **OPSIDE**.

There is no specific calling hierarchy in FORTRAN – a function can call a subroutine or *vice-versa* and the called sub-program can call still further sub-programs.

§§2 Arithmetic statements

FORTRAN arithmetic is performed by “smart” operators acting on **typed variables** and **literals**. A variable is simply a name that refers to a specific location in memory. The type declaration is a way to let the compiler know how much memory to allot for that variable. A literal is an explicit number that appears in the program, such as the values 3.14159 and 180. in the preceding example.

FORTRAN arithmetic expressions can freely mix types. To make this possible, the arithmetic operators are **overloaded** in the sense that the plus sign –say– can add floating point numbers, integers or complex numbers in any combination, mixture or order.

Consider, e.g., the actions performed by the FORTRAN compiler in parsing the arithmetic assignment statement

$$A = B1*3 + B2*1.2E-5 - H(3)/3.14159265358979D14 + K$$

keeping in mind that in FORTRAN, data types can be declared explicitly or implicitly³:

- Define and reserve space for a floating-point single precision variable A (implicit type REAL) if A has not been defined previously (perhaps as something else);
- Convert the literal integer constant 3 to floating point and multiply it by the (implicit-REAL) variable B1's current value (fetch from memory), placing the product in temporary storage (TEMP).
- Fetch (implicit-REAL) variable B2 and multiply it by the REAL literal 1.2E-5;
- Add the second product to the contents of TEMP;
- Fetch the 3rd element of the (implicit-REAL) array H;
- Divide by the DREAL (double-precision) literal 3.14159265358979D-14 (=π), converting to and from DREAL format as necessary;
- Convert the dividend to REAL and subtract from TEMP;
- Convert (implicit) INTEGER variable K to REAL and add to TEMP;
- Move the result from TEMP to the memory reserved for A.

These actions can be over-ridden by explicit type declarations. For example, if the program had contained the following statements in its first few lines:

```
INTEGER A, H(15), B1, B2
REAL K
```

the conversions and assignments would have been floating point to integer, rather than *vice-versa*.

3. The original version of FORTRAN included naming conventions such that names beginning with letters I, J, K, L, M and N are assumed to be integers, while those beginning with other letters are assumed to be single-precision floating point numbers. Subsequent versions have maintained this convention for backward compatibility.

To achieve the simplicity of mixed-mode expressions, the FORTRAN compiler must be prepared for any eventuality. The operators "+", "-", "*", "/" and "=" must be "smart" (overloaded) – they must "know" (or at least be able to figure out) what kinds of numbers are going to be used and what kinds of arithmetic will be used to combine them. The FORTRAN exponentiation operator "**" must similarly "know" whether the base is INTEGER, REAL, DREAL or COMPLEX (some FORTRAN's even permit DCOMPLEX), and the same for the exponent. That is, it must be able to compile 16 (or 25) versions of **, depending on circumstances. The compiler must contain decision branches to handle every eventuality. Compilers for languages such as FORTRAN, PASCAL, C or Modula-2 are therefore complex and slow.

Smart operators benefit the user by simplifying source code. The benefit is only partial, however, since the programmer must still keep track of types in calling sequences for subroutines, and in declaring global variables with COMMON and EQUIVALENCE statements.

Since FORTRAN subroutines can be compiled separately, many a subtle bug has been introduced by omitting an argument from a long calling sequence, or by inverting arguments in a list (thereby, for example, telling a subroutine to interpret a REAL as a very large INTEGER). I can vouch for these problems from long, sad experience debugging FORTRAN.

FORTRAN provides a limited suite of data types: INTEGER, LONG-INTEGER, REAL, DREAL, COMPLEX, DCOMPLEX, LOGICAL and CHARACTER. It provides no facilities for defining any new types (other than arrays of the above). Arrays must be declared according to a strict format – up to 3 indices are permitted.

FORTRAN's array notation is simple, logical and follows the conventions of algebra: parentheses replace subscripts via

$$A_j \Rightarrow A(I,J).$$

FORTRAN provides facilities for initializing constants and variables at run-time: the DATA statement within a program or subroutine, and the BLOCK DATA subprogram for initializing global variables in COMMON.

Limited control of memory allocation is provided: placed at the beginning of a program or subprogram, COMMON, BLOCK COMMON and EQUIVALENCE specification statements allow local variables to be made global or partially global, under the same or different names. DIMENSION allocates memory for arrays. (Dynamic re-allocation is not permitted.)

Finally, EXTERNAL directs the compiler (more precisely, the linker and loader) to search outside the subprogram for the specified name: for example, the usage

SUBROUTINE MYSUB(X,DUMMY,ANSWER)

permits the name of a function or subroutine to be inserted as an argument into the calling string at runtime. This facility is essential to separately compiled modules, of course.

Modern FORTRAN has evolved by accretion, with additions designed not to obsolesce older methods of accomplishing tasks. Thus FORTRAN has several ways to define functions, through external subprograms and through inline definitions; and several ways to allocate memory for arrays. Data types can be changed explicitly *via* functions and implicitly *via* replacement statements, leading to such redundancies as

A = FLOAT(K)
A = K

or

K = IFIX(A)
K = A .

§§3 Function library

Crucial to FORTRAN's utility in scientific programming is the mathematical function library, including REAL, DREAL and COMPLEX (at least!) versions of trigonometric functions, exponentials, logarithms, inverse trigonometric functions, some-

times hyperbolic functions and their inverses, and often a random number generator of uncertain quality.

FORTRAN supports modularity through separate compilation of functions and subroutines. For example, we can write a library function to compute complex Legendre polynomials:

```
COMPLEX FUNCTION CPLEG(Z,N)
COMPLEX Z, CP0, CP1, CMPLX
CPLEG = CMPLX( 1., 0. )
IF ( N .EQ. 0 ) RETURN
CP0 = CMPLX( 0., 0. )
K = 0
1 CP1 = CPLEG
K1 = K + 1
CPLEG = (( K + K1 ) * Z * CP1 - K * CP0 ) / K1
IF ( K .EQ. N ) RETURN
CP0 = CP1
GOTO 1
END
```

Because all the decisions as to which overloaded operator to use must be made when the function is compiled, a single-precision REAL Legendre polynomial routine will require a separate version from the above.

Worse, because the typical function or subroutine calling sequence wastes memory and execution time, there are severe penalties in efficiency that militate against fine-grained decomposition. That is, the code in one routine is unlikely to be re-used in another routine. Instead, it must be repeated, wasting memory.

§2 What Is FORTH ?

When I first encountered FORTH, it appeared to me as Looking Glass Land must have, to Alice. Twenty-five years' experience with FORTRAN colored my perceptions, making FORTH seem very strange indeed.

FORTH makes no essential distinctions between data structures, operators, functions or subroutines. *Everything* in FORTH is the *same* thing: a **word**. In appearance, words are strings of text separated by spaces. Functionally, words are **subroutines**. To execute a word, type its name, then a carriage return. No GOSUBs, CALLs or RETURNs are needed. This simple grammar is beautiful because it leaves nothing to remember.

Whereas FORTRAN imposes stringent naming conventions — names must begin with a letter, may be no longer than seven characters, and may use only letters and digits — FORTH has no such restrictions. FORTH names can be much more expressive than those in FORTRAN or even Pascal and C, for that matter.

For a preview of FORTH's flavor, consider the FORTH version of the Legendre polynomial function⁴:

```
\ Gx are generic operations (Real or Complex)
: S->FS  S->F  REAL*8  F>FS ;
: PLEG      ([z] n -- :: - - p[z, n])
    >R DUP >R >FS      ( -- :: - - z )
    R@ G=1 R> G=0 R> ( -- n :: - - z P1 P0 )
    ?DUP IF           \ loop n times, if n > 0
    0 DO             \ begin loop
        I S->FS G*      ( :: - - z P1 P0*I )
        FS>F GOVER GOVER
        G* I 2* 1+ S->FS
        G* F>FS G-
        I 1+ S->FS G/      ( :: - - z P1 P2 )
        GSWAP            ( :: - - z P2 P1 )
        LOOP             \ end loop
        THEN            \ end IF . . . THEN clause
        GDROP  GPLUCK ;   \ clean up stacks
```

4. The items between parentheses, (...), and following a backslash, "\", are comments.

We note the following similarities and differences between the FORTRAN and FORTH versions:

- They are of similar length. The FORTH version contains more explicit steps and looks more cryptic. The FORTRAN version looks more like algebraic formulae.
- The FORTH function lacks an argument list. Functions and subroutines generally look for arguments on stacks⁵ built into the system.
- The code uses both primitive words from the FORTH “kernel”, as well as advanced concepts from *Scientific FORTH*. In particular, the FORTH version employs generic operations with “run-time binding”, so one version works with REAL*4, REAL*8, COMPLEX*8 and COMPLEX*16 data types. By contrast, in FORTRAN one needs a separate Legendre function for each type desired.
- FORTH looks more cryptic than FORTRAN because it uses postfix (“reverse Polish”) notation, just like a Hewlett-Packard calculator. Thus, while FORTRAN lets us display the algorithm in almost-algebraic form, FORTH’s postfix arithmetic conceals the algorithm by decomposing it. This disadvantage can be overcome by suitable commenting, through telegraphic choices of names, or by employing the FORMula TRANslator from Chapter 11.

FORTH’s simple linguistic structure permits almost self-commenting code⁶, through clever naming of data structures and operations. In Chapter 2 we shall comment in detail on this and other differences between FORTRAN and FORTH.

Every operation that FORTRAN is capable of can be programmed easily in FORTH. For example, the EXTERNAL specification of FORTRAN has its analogue in “vectoring”.

5. A stack is a data structure like a pile of cards, each containing a number. New numbers are added by placing them atop the pile, numbers are also deleted from the top. In essence, a stack is a “last-in, first-out” buffer.
6. L. Brodie, *Thinking Forth* (Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1984). M. Ham, “Structured Programming”, *Dr. Dobb’s Journal*, July 1986.

But FORTH can not only imitate FORTRAN — using far less memory, compiling and debugging much faster, and often executing faster as well — it can perform tricks that FORTRAN accomplishes barely or not at all. The programming examples sprinkled throughout the book, and concentrated in Chapters 6, 8 and 11 offer repeated concrete proof for these assertions.

My experience with FORTH following 25 or so years in which FORTRAN (and sometimes BASIC) were my staple languages leads me to believe the chief advantage of FORTH over the more common procedural languages is its potential for directness and clarity of algorithmic expression.

One reason FORTH has not yet realized its potential in scientific computing may be that scientists and programmers tend to reside in orthogonal communities, so that no one has until now troubled to publicize the extensions that make FORTH convenient for scientific problem-solving. My sincere hope is that this book will in some measure mitigate this lack.

Programming in FORTH

Contents

§1 The structure of FORTH	13
§2 Extending the dictionary	15
§3 Stacks and reverse Polish notation (RPN)	17
§§1 Manipulating the parameter stack	19
§§2 The return stack and its uses	21
§4 Fetching and storing	23
§5 Arithmetic operations	24
§6 Comparing and testing	24
§7 Looping and structured programming	25
§8 The pearl of FORTH	26
§§1 Dummy words	27
§§2 Defining “defining” words	28
§§3 Run-time vs. compile-time actions	30
§§4 Advanced methods of controlling the compiler	34
§9 Strings	36
§10 FORTH programming style	38
§§1 Structure	38
§§2 “Top-down” design	39
§§3 Information hiding	40
§§4 Documenting and commenting FORTH code	42
§§5 Safety	44

This chapter briefly reviews the main ideas of FORTH to let the reader understand the program fragments and subroutines that comprise the meat of this book. We make no pretense to complete coverage of standard FORTH programming methods. **Chapter 2 is not a programmer's manual!**

Suppose the reader is stimulated to try FORTH — how can he proceed? Several excellent FORTH texts and references are available: *Starting FORTH*¹ and *Thinking FORTH*² by Leo Brodie; and *FORTH: a Text and Reference*³ by M.Kelly and N.Spies. I strongly recommend reading FTR or SF (or both) before trying to use the ideas from this book on a FORTH system. (Or at least read one concurrently.)

The (commercial) GENie information network maintains a session devoted to FORTH under the aegis of the Forth Interest Group (FIG).

FIG publishes a journal *Forth Dimensions* whose object is the exchange of programming ideas and clever tricks.

The Association for Computing Machinery (11 West 42nd St., New York, NY 10036) maintains a Special Interest Group on FORTH (SIGForth).

The Institute for Applied FORTH Research (Rochester, NY) publishes the refereed *Journal of FORTH Application and Research*, that serves as a vehicle for more scholarly and theoretical papers dealing with FORTH.

Finally, an attempt to codify and standardize FORTH is underway, so by the time this book appears the first draft of an ANS FORTH and extensions may exist.

-
1. L. Brodie, *Starting FORTH*, 2nd ed. (Prentice-Hall, NJ, 1986), referred to hereafter as SF.
 2. L. Brodie, *Thinking FORTH* (Prentice-Hall, NJ 1984), referred to hereafter as TF.
 3. M. Kelly and N. Spies, *FORTH: a Text and Reference* (Prentice-Hall, NJ, 1986), referred to hereafter as FTR.

§1 The structure of FORTH

The “atom” of FORTH is a **word** – a previously-defined operation (defined in terms of machine code or other, previously-defined words) whose definition is stored in a series of linked lists called the **dictionary**. The FORTH operating system is an endless loop (outer interpreter) that reads the console and interprets the input stream, consulting the dictionary as necessary. If the stream contains a word⁴ in the dictionary the interpreter immediately executes that word.

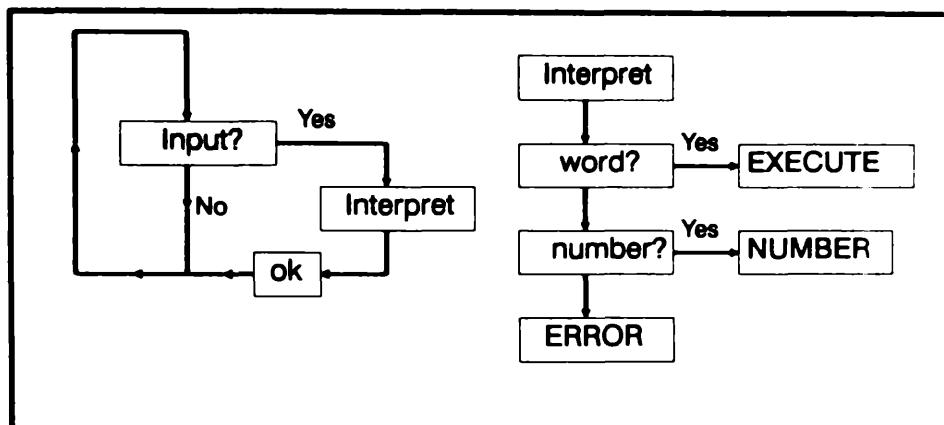


Fig. 2-1 Overview of FORTH outer interpreter

In general, because FORTH is interpretive as well as compiled, the best way to study something new is in front of a computer running FORTH. Therefore we explain with illustrations, expecting the reader to try them out.

In what follows, anything the user types in will be set in Helvetica, such as DECIMAL below.

Machine responses appear in ordinary type.

We now give a trivial illustration:

DECIMAL <cr> ok

4. Successive words in the input stream are separated from each other by blank spaces, ASCII 20hex, the standard FORTH delimiter.

Notes:

- <cr> means “the user pushes the ENTER or ↵ button”.
- ok is what FORTH says in response to an input line, if nothing has gone wrong.
- DECIMAL is an instruction to use base 10 arithmetic. FORTH will use any base you tell it, within reason, but usually only DECIMAL and HEX (hexadecimal) are predefined.

When the outer interpreter (see Fig. 2.1 on p. 13) encounters text with no dictionary entry, it tries to interpret it as a **NUMBER**.

It places the number in a special memory location called “the top of the stack” (TOS)⁵

```
2 17 + . <cr> 19 ok
```

Notes:

- FORTH interprets 2 and 17 as numbers, and pushes them onto the stack. “+” is a word and so is “.” so they are EXECUTEd.
- + adds 2 to 17 and leaves 19 on the stack.
- The word . (called “emit”) removes 19 from the stack and displays it on the screen.

We might also have said⁶

```
HEX 0A 14 * . <cr> C8 ok
```

(Do you understand this? Hint: HEX stands for “switch to hexadecimal arithmetic”.)

5. We will explain about the stack in §2.3.
6. since FORTH uses words, when we enter an input line we say the corresponding phrase.

If the incoming text can neither be located in the dictionary nor interpreted as a number, FORTH issues an error message.

§2 Extending the dictionary

The compiler is one of FORTH's most endearing features. It is elegant, simple, and mostly written in FORTH. Although the technical details of the FORTH compiler are generally more interesting to systems developers than to scientists, its components can often be used to solve programming problems. When this is the case, we necessarily discuss details of the compiler. In this section we discuss how the compiler extends the dictionary. In §2§§8 below we examine the parts of the compiler in greater detail.

FORTH has special words that allow the creation of new dictionary entries, *i.e.*, new words. The most important are “:” (“start a new definition”) and “;” (“end the new definition”).

Consider the phrase

: NEW-WORD WORD1 17 WORD2 . . . WORDn ; ok

The initial “:” is EXECUTEd because it is already in the dictionary. Upon execution, “:” does the following:

- Creates a new dictionary entry, **NEW-WORD**, and switches from interpret- to compile mode.
- In compile mode, the interpreter looks up words and – rather than executing them – installs pointers to their code. If the text is a number (17 above), FORTH builds the literal number into the dictionary space allotted for **NEW-WORD**.
- The action of **NEW-WORD** will be to EXECUTE sequentially the previously-defined words **WORD1**, **WORD2**, ... **WORDn**, placing any built-in numbers on the stack as they occur.

- The FORTH compiler **EXECUTEs** the last word “ ; ” of the definition, by installing code (to return control to the next outer level of the interpreter⁷) then switching back from compile to interpret mode. Most other languages treat tokens like “ ; ” as flags (in the input stream) that *trigger* actions, rather than actions in their own right. FORTH lets components execute themselves.

In FORTH *all* subroutines are words that are invoked when they are named. No explicit CALL or GOSUB statement is required.

The above definition of **NEW-WORD** is extremely structured compared with FORTRAN or BASIC. Its definition is just a series of subroutine calls.

We now illustrate how to define and use a new word using the previously defined words “ : ” and “ ; ”. Enter the phrase (this new word *** +** expects 3 numbers, *a*, *b*, and *c* on the stack)

```
: *+ * + ; ok
```

Notes:

- ***** multiplies *b* with *c*, leaving *b*c*.
- **+** then adds *b*c* to *a*, leaving *a + b*c* behind.

Now we actually try out *** +** :

```
DECIMAL 5 6 7 *+ . 47 ok
```

Notes:

- The period **.** is not a typo, it EMITs the result.
- FORTH’s response to **a b c *+ .** is **a + b*c ok**.

7. This level could be either the outer interpreter or a word that invokes **NEW-WORD**.

What if we were to enter `* +` with nothing on the stack ? Let's try it and see (`.S` is a word that displays the stack without changing its contents):

`.S` empty stack ok

`* +` empty stack ok

Exercise:

Suppose you entered the input line

HEX 5 6 7 *+ . <cr> xxx ok

What would you expect the response `xxx` to be?

Answer: 2F

§3 Stacks and reverse Polish notation (RPN)

We now discuss the stack and the “reverse Polish” or “postfix” arithmetic based on it. (Anyone who has used one of the Hewlett-Packard calculators should already be familiar with the basic concepts.)

A Polish mathematician (J.Lukasewicz) showed that numerical calculations require an irreducible minimum of elementary operations (fetching and storing numbers as well as addition, subtraction, multiplication and division). The minimum is obtained when the calculation is organized by “stack” arithmetic.

Thus virtually all central processors (CPU's) intended for arithmetic operations are designed around stacks. FORTH makes efficient use of CPU's by reflecting this underlying stack architecture in its syntax, rather than translating algebraic-looking program statements (“infix” notation) into RPN-based machine operations as FORTRAN, BASIC, C and Pascal do.

But what is a stack? As the name implies, a stack is the machine analog of a pile of cards with numbers written on them. Numbers are always added to, and removed from, the top of the pile. (That is, a stack resembles a job where layoffs follow seniority: last in, first out.) Thus, the FORTH input line

DECIMAL 2 5 73 -16 ok

followed by the line

+ - * . yyy ok

leaves the stack in the successive states shown in Table 2-1 below

Cell #	Initial	Ops →	+	-	*	.
0	-16	Result	57	-52	-104	...
1	73	→	5	2
2	5		2	
3	2			

Table 2-1 Picture of the stack during operations

We usually employ zero-based relative numbering in FORTH data structures —stacks, arrays, tables, etc.— so TOS (“top of stack”) is given relative #0, NOS (“next on stack”) #1, etc.

The operation “.” (“emit”) displays -104 to the screen, leaving the stack empty. That is, **yyy** above is **-104**.

§§1 Manipulating the parameter stack

FORTH systems incorporate (at least) two stacks: the parameter stack which we now discuss, and the return stack which we defer to §2.3.2.

In order to use a stack-based system, we must be able to put numbers on the stack, remove them, and rearrange their order. FORTH includes standard words for this purpose.

Putting numbers on the stack is easy: one simply types the number (or it appears in the definition of a FORTH word).

To remove a number we have the word **DROP** that drops the number from TOS and moves up all the other numbers.

To exchange the top 2 numbers we have .

DUP duplicates the TOS into NOS, pushing down all the other numbers.

ROT rotates the top 3 numbers.

Cell #	Initial	Ops → DROP SWAP ROT DUP			
0	-16	Result	73	73	5 -16
1	73	→	5	-16	-16
2	5		2	5	73
3	2		...	2	2
4

Table 2-2 Stack manipulation operators

These actions are shown on page 19 above in Table 2-2 (we show what each word does to the initial stack).

In addition the words **OVER**, **UNDER**, **PICK** and **ROLL** act as shown in Table 2-3 below (note **PICK** and **ROLL** must be

Cell #	Initial	Ops→	OVER	UNDER	4 PICK	4 ROLL
0	-16	Result	73	-16	2	
1	73	→	-16	73	-16	
2	5		73	-16	73	
3	2		5	5	5	
4	...		2	2	2	...

Table 2-3 *More stack manipulation operators*

preceded by an integer that says where on the stack an element gets **PICKed** or **ROLLED**).

Clearly, **1 PICK** is the same as **DUP**, **2 PICK** is a synonym for **OVER**, **2 ROLL** means **SWAP**, and **3 ROLL** means **ROT**.

As Brodie has noted (TF), it is rarely advisable to have a word use a stack so deep that **PICK** or **ROLL** is needed. It is generally better to keep word definitions short, using only a small number of arguments on the stack and consuming them to the extent possible. On the other hand, **ROT** and its opposite, **-ROT**⁸, are often useful.

8. defined as :**-ROT ROT ROT**;

§§2 The return stack and its uses

We have remarked above in §2§2 that compilation establishes links from the calling word to the previously-defined word being invoked. Part of the linkage mechanism – during actual execution – is the **return stack** (rstack): the address of the next word to be invoked after the currently executing word is placed on the rstack, so that when the current word is done, the system jumps to the next word. Although it might seem logical to call the address on the rstack the **next address**, it is actually called the **return address** for historical reasons.

In addition to serving as a reservoir of return addresses (since words can be nested, the return addresses need a stack to be put on) the rstack is where the limits of a **DO ... LOOP** construct are placed⁹.

The user can also store/retrieve to/from the rstack. This is an example of using a component for a purpose other than the one it was designed for. Such use is not encouraged by every FORTH text, needless to say, since it introduces the spice of danger. To store to the rstack we say **>R**, and to retrieve we say **R>**. **DUP >R** is a speedup of the phrase **DUP >R**. The words **D>R DR>**, for moving double-length integers, also exist on many systems. The word **R@** copies the top of the rstack to the TOS.

The danger is this: anything put on the rstack during a word's execution must be removed before the word terminates. If the **>R** and the **R>** do not balance, then a **wrong next address** will be jumped to and **EXECUTEd**. Since this could be the address of data, and since it is being interpreted as machine instructions, the results will be **always unpredictable**, but seldom amusing.

Why would we want to use the rstack for storage when we have a perfectly good parameter stack to play with? Sometimes it becomes simply impossible to read code that performs complex gymnastics on the parameter stack, even though FORTH permits such gymnastics.

9. We discuss looping in §2.7 below.

Consider a problem — say, drawing a line on a bit-mapped graphics output device from (x,y) to (x',y') — that requires 4 arguments. We have to turn on the appropriate pixels in the memory area representing the display, in the ranges from the origin to the end coordinates of the line. Suppose we want to work with x and y first, but they are 3rd and 4th on the stack. So we have to **ROLL** or **PICK** to get them to TOS where they can be worked with conveniently. We probably need them again, so we use

4 PICK 4 PICK (- - x y x' y' x y)

Now 6 arguments are on the stack! See what I mean? A better way stores temporarily the arguments x' and y' , leaving only 2 on the stack. If we need to duplicate them, we can do it with an already existing word, **DDUP**.

Complex stack manipulations can be avoided by defining **VARIABLEs** — named locations — to store numbers. Since FORTH variables are typically *global* — any word can access them — their use can lead to unfortunate and unexpected interactions among parts of a large program. Variables should be used sparingly.

While FORTH permits us to make variables local to the subroutines that use them¹⁰, for many purposes the rstack can advantageously replace local variables:

- The rstack already exists, so it need not be defined anew.
- When the numbers placed on it are removed, the rstack shrinks, thereby reclaiming some memory.

Suppose, in the previous example, we had put x' and y' on the rstack via the phrase

>R >R DDUP .

Then we could duplicate and access x and y with no trouble.

10. See **FTR**, p. 325ff for a description of beheading — a process to make variables local to a small set of subroutines. Another technique is to embed variables within a data structure so they cannot be referenced inadvertently. Chapters 2§§§3-2, 3§5§§2, 5§1§§2 and 11§2 offer examples.

A note of caution: since the rstack is a critical component of the execution mechanism, we mess with it at our peril. If we want to use it, we must clean up when we are done, so it is in the same state as when we found it. A word that places a number on the rstack must get it off again – using **R>** or **RDROP** – before exiting that word¹¹. Similarly, since **DO ... LOOP** uses the rstack also, for each **>R** in such a loop (after **DO**) there must be a corresponding **R>** or **RDROP** (before **LOOP** is reached). Otherwise the results will be unpredictable and probably will crash the system.

§4 Fetching and storing

Ordinary (16-bit) numbers are fetched from memory to the stack by “@” (“fetch”), and stored by “!” (“store”). The word **@** expects an address on the stack and replaces that address by its contents using, e.g., the phrase **X @**. The word “!” expects a number (NOS) and an address (TOS) to store it in, and places the number in the memory location referred to by the address, consuming both arguments in the process, as in the phrase **32 X !**

Double length (32-bit) numbers can similarly be fetched and stored, by **D@** and **D!**. (FORTH systems designed for the newer 32-bit machines sometimes use a 32-bit-wide stack and may not distinguish between single- and double-length integers.)

Positive numbers smaller than 255 can be placed in single bytes of memory using **C@** and **C!**. This is convenient for operations with strings of ASCII text, for example screen, file and keyboard I/O.

In Chapters 3, 4, 5 and 7 we shall extend the lexicon of **@** and **!** words to include floating point and complex numbers.

11. **RDROP** is a handy way to exit from a word before reaching the final “;”. See **TF**.

§5 Arithmetic operations

The 1979 or 1983 standards, not to mention the forthcoming ANSII standard, require that a conforming FORTH system contain a certain minimum set of predefined words. These consist of arithmetic operators + - * / MOD /MOD */ for (usually) 16-bit *signed-integer* (-32767 to +32767) arithmetic, and equivalents for *unsigned* (0 to 65535), double-length and mixed-mode (16- mixed with 32-bit) arithmetic. The list will be found in the glossary accompanying your system, as well as in SF and FTR.

§6 Comparing and testing

In addition to arithmetic, FORTH lets us **compare** numbers on the stack, using relational operators > < = . These operators work as follows: the phrase

2 3 > <cr> ok

will leave 0 (“false”) on the stack, because 2 (NOS) is not greater than 3 (TOS). Conversely, the phrase

2 3 < <cr> ok

will leave -1 (“true”) because 2 is less than 3. Relational operators typically consume their arguments and leave a “flag” to show what happened¹². Those listed so far work with signed 16-bit integers. The operator U< tests *unsigned* 16-bit integers (0-65535).

FORTH offers unary relational operators 0= 0> and 0< that determine whether the TOS contains a (signed) 16-bit integer that is 0, positive or negative. Most FORTHS offer equivalent relational operators for use with double-length integers.

The relational words are used for branching and control. The usual form is

```
: MAYBE 0> IF WORD1 WORD2 ...
WORDn THEN ;
```

12. The original FORTH-79 used +1 for “true”, 0 for “false”; many newer systems that mostly follow FORTH-79 use -1 for “true”. HS/FORTH is one such. Both FORTH-83 and ANSII FORTH require -1 for “true”, 0 for “false”.

The word **MAYBE** expects a number on the stack, and executes the words between **IF** and **THEN** if the number on the stack is positive, but not otherwise. If the number initially on the stack were negative or zero, **MAYBE** would do nothing.

An alternate form including **ELSE** allows two mutually exclusive actions:

```
: CHOOSE      0> IF WORD1 ... WORDn
              ELSE WORD1' ... WORDn'
              THEN ;      (n--)
```

If the number on the stack is positive, **CHOOSE** executes **WORD1 WORD2 ... WORD**, whereas if the number is negative or 0, **CHOOSE** executes **WORD1' ... WORDn'**.

In either example, **THEN** marks the end of the branch, rather than having its usual logical meaning¹³.

§7 Looping and structured programming

FORTH contains words for setting up loops that can be definite or indefinite:

```
BEGIN xxx flag UNTIL
```

The words represented by **xxx** are executed, leaving the TOS (flag) set to 0 (F) – at which point **UNTIL** leaves the loop – or -1 (T) – at which point **UNTIL** makes the loop repeat from **BEGIN**.

A variant is

```
BEGIN xxx flag WHILE yyy REPEAT
```

Here **xxx** is executed, **WHILE** tests the flag and if it is 0 (F) leaves the loop; whereas if **flag** is -1 (T) **WHILE** executes **yyy** and

13. This has led some FORTH gurus to prefer the synonymous word **ENDIF** as clearer than **THEN**.

REPEAT then branches back to **BEGIN**. These forms can be used to set up loops that repeat until some external event (pressing a key at the keyboard, e.g.) sets the flag to exit the loop. They can also be used to make endless loops (like the outer interpreter of FORTH) by forcing flag to be 0 in a definition like

```
: ENDLESS BEGIN xxx 0 UNTIL ;
```

FORTH also implements indexed loops using the words **DO**, **LOOP**, **+LOOP**, **/LOOP**. These appear within definitions, e.g.

```
: LOOP-EXAMPLE 100 0 DO xxx LOOP ;
```

The words **xxx** will be executed 100 times as the lower limit, 0, increases in unit steps to 99. To step by -2's, we use the phrase

-2 +LOOP

to replace **LOOP**, as in

```
: DOWN-BY-2's 0 100 DO xxx -2 +LOOP ;
```

The word **/LOOP** is a variant of **+LOOP** for working with unsigned limits¹⁴ and increments (to permit the loop index to go up to 65535 in 16-bit systems).

§8 The pearl of FORTH

An unusual construct, **CREATE ... DOES >**, has been called “the pearl of FORTH”¹⁵. This is more than poetic license.

CREATE is a component of the compiler that makes a new dictionary entry with a given name (the next name in the input stream) and has no other function.

DOES > assigns a specific run-time action to a newly **CREATED** word (we shall see this in §2§§8-3 below).

14. Signed 16-bit integers run from -32768 to +32767, unsigned from 0 to 65535. See **FTR**.

15. Michael Ham, “Structured Programming”, *Dr. Dobb’s Journal of Software Tools*, October, 1986.

§§1 Dummy words

Sometimes we use **CREATE** to make a dummy entry that we can later assign to some action:

```
CREATE DUMMY
CA' * DEFINES DUMMY
```

The second line translates as "The code address of * defines **DUMMY**". Entry of the above phrase would let **DUMMY** perform the job of * just by saying **DUMMY**. That is, FORTH lets us first define a dummy word, and then give it any other word's meaning¹⁶.

Here is one use of this power: Suppose we have to define two words that are alike except for some piece in the middle:

```
: *_WORD WORD1 WORD2 */ WORD3 WORD4 ;
: */WORD WORD1 WORD2 */ WORD3 WORD4 ;
```

we could get away with 1 word, together with **DUMMY** from above,

```
: *_or_*/WORD
WORD1 WORD2
DUMMY
WORD3 WORD4 ;
```

by saying

```
CA' * DEFINES DUMMY *_or_*/WORD
or
```

```
CA' */ DEFINES DUMMY *_or_*/WORD .
```

16. This usage is a non-standard construct of HS/FORTH.

This technique, a rudimentary example of **vectoring**, saves memory and saves programming time by letting us vary something in the middle of a definition *after the definition has been entered in the dictionary*. However, this technique must be used with caution as it is akin to **self-modifying code**¹⁷.

A similar procedure lets a subroutine call itself recursively, an enormous help in coding certain algorithms.

§§2 Defining “defining” words

The title of this section is neither a typo nor a stutter: **CREATE** finds its most important use in extending the powerful class of FORTH words called “defining” words. The colon compiler “:” is such a word, as are **VARIABLE** and **CONSTANT**. The definition of **VARIABLE** is simple

```
: VARIABLE      CREATE 2 ALLOT ;
```

Here is how we use it:

```
VARIABLE X <cr> ok
```

The inner workings of **VARIABLE** are these:

- **CREATE** makes a dictionary entry with the next name in the input stream — in this case, **X**.
- Then the number 2 is placed on the stack, and the word **ALLOT** increments the pointer that represents the current location in the dictionary by 2 bytes.

17. Self-modifying machine code is considered a serious “no-no” by modern structured programming standards. Although it is sometimes valuable, few modern cpu’s are capable of handling it safely. More often, because cpu’s tend to use pipelining and parallelism to achieve speed, a piece of code might be modified in memory, but — having been pre-fetched before modification — actually execute in unmodified form.

- This leaves a 2-byte vacancy to store the value of the variable (that is, the next dictionary header begins 2 bytes above the end of the one just defined).

When the outer interpreter loop encounters a new **VARIABLE**'s name in the input stream, that name's address is placed on the stack. But this is also the location where the 2 bytes of storage begins. Hence when we type in **X**, the TOS will contain the storage address named **X**.

As noted in §2.4 above, the phrase **X @** (pronounced "X fetch") places the contents of address **X** on the stack, dropping the address in the process. Conversely, to store a value in the named location **X**, we use **!** ("store"): thus

```
4 X !      <cr> ok
X @ .      <cr> 4 ok
```

Double-length variables are defined via **DVARIABLE**, whose definition is

```
: DVARIABLE CREATE 4 ALLOT ;
```

FORTH has a method for defining words initialized to contain specific values: for example, we might want to define the number 17 to be a word. **CREATE** and **,** ("comma") let us do this as follows:

```
17 CREATE SEVENTEEN , <cr> ok
```

Now test it via

```
SEVENTEEN @ . <cr> 17 ok
```

Note: The word “ , ” (“comma”) puts TOS into the next 2 bytes of the dictionary and increments the dictionary pointer by 2.

A word **C**, (“see-comma”) puts a byte-value into the next byte of the dictionary and increments the pointer by 1 byte.

§§3 Run-time vs. compile-time actions

In the preceding example, we were able to initialize the variable **SEVENTEEN** to 17 when we **CREATED** it, but we still have to fetch it to the stack via **SEVENTEEN @** whenever we want it. This is not quite what we had in mind: we would like to find 17 in TOS when we say **SEVENTEEN**. The word **DOES >** gives us precisely the tool to do this.

As noted above, the function of **DOES >** is to specify a run-time action for the “child” words of a defining word. Consider the defining word **CONSTANT**, defined in high-level¹⁸ FORTH by

```
: CONSTANT CREATE , DOES > @ ;
```

and used as

```
53 CONSTANT PRIME ok
```

Now test it:

```
PRIME . <cr> 53 ok
```

What happened?

- **CREATE** (hidden in **CONSTANT**) made an entry (named **PRIME**, the first word in the input stream following **CONSTANT**). Then “ , ” placed the TOS (the number 53) in the next two bytes of the dictionary.

18. Of course **CONSTANT** is usually a machine-code primitive, for speed.

- **DOES >** (inside **CONSTANT**) then appended the actions of all words between it and “ ; ” (the end of the definition of **CONSTANT**) to the child word(s) defined by **CONSTANT**.
- In this case, the only word between **DOES >** and ; was **@** , so all FORTH constants defined by **CONSTANT** perform the action of placing their address on the stack (anything made by **CREATE** does this) and fetching the contents of this address.

§§3-1 Klingons

Let us make a more complex example. Suppose we had previously defined a word **BOX** (n x y - -) that draws a small square box of n pixels to a side centered at (x, y) on the graphics display. We could use this to indicate the instantaneous location of a moving object — say a Klingon space-ship in a space-war game.

So we define a defining word that creates (not very realistic looking) space ships as squares n pixels on a side:

```
: SPACE-SHIP CREATE , DOES >
    @ -ROT ( --n x y )      BOX ;
: SIZE ;           \ do-nothing word
```

Now, the usage would be (**SIZE** is included merely as a reminder of what 5 means — it has no function other than to make the definition look like an English phrase)

```
SIZE 5 SPACE-SHIP KLINGON <cr> ok
71 35 KLINGON <cr> ok
```

Of course, **SPACE-SHIP** is a poorly constructed defining word because it does not do what it is intended to do. Its child-word **KLINGON** simply draws itself at (x, y).

What we really want is for **KLINGON** to *undraw* itself from its old location, compute its new position according to a set of rules, and then *redraw* itself at its new position. This sequence of operations would require a definition more like

```
: OLD.POS@ ( adr -- adr n x y ) DUP @ OVER
    2+ D@ ;
```

```
: SPACE-SHIP CREATE , 4 ALLOT DOES>
OLD.POS@ UNBOX NEW.POS!
OLD.POS@ BOX DROP ;
```

where the needed specialized operation **UNBOX** would be defined previously along with **BOX**.

§§3-2 Dimensioned data (with intrinsic units)

Here is another example of the power of defining words and of the distinction between compile-time and run-time behaviors.

Physical problems generally work with quantities that have dimensions, usually expressed as mass (M), length (L) and time (T) or products of powers of these. Sometimes there is more than one system of units in common use to describe the same phenomena.

For example, traffic police reporting accidents in the United States or the United Kingdom might use inches, feet and yards; whereas Continental police would use the metric system. Rather than write different versions of an accident analysis program it is simpler to write one program and make unit conversions part of the grammar. This is easy in FORTH; impossible in FORTRAN, BASIC, Pascal or C; and possible, but exceedingly cumbersome in Ada¹⁹.

We simply keep all internal lengths in millimeters, say, and convert as follows²⁰:

- 19. An example (and its justification) of dimensioned data types in Ada is given by Do-While Jones, *Dr. Dobb's Journal*, March 1987. The FORTH solution below is much simpler than the Ada version.
- 20. This example is based on 16-bit integer arithmetic. The word ***/** means “multiply the third number on the stack by NOS, keeping 32 bits of precision, and divide by TOS”. That is, the stack comment for ***/** is **(a b c -- a*b/c)**.

```
: INCHES 254 10 */ ;
: FEET [ 254 12 * ] LITERAL 10 */ ;
: YARDS [ 254 36 * ] LITERAL 10 */ ;
: CENTIMETERS 10 * ;
: METERS 1000 * ;
```

The usage would be

```
10 FEET . <cr> 3048 ok
```

These are more definitions than necessary, of course, and the technique generates unnecessary code. A more compact approach uses a *defining word*, **UNITS** :

```
: D, SWAP , , ; \! double-length # in next cells
: UNITS CREATE D, DOES> D@ */ ;
```

Then we could make the table

```
254 10      UNITS INCHES
254 12 * 10 UNITS FEET
254 36 * 10 UNITS YARDS
10 1        UNITS CENTIMETERS
1000 1      UNITS METERS
\ Usage:
\ 10 FEET . <cr> 3048 ok
\ 3 METERS . <cr> 3000 ok
\ .....
\ etc.
```

This is an improvement, but FORTH lets us do even better: here is a simple extension that allows conversion back to the input units, for use in output:

VARIABLE <AS>	\ new variable
0 <AS> !	\ initialize to "F"
: AS -1 <AS> ! ;	\ set <AS> = "T"

```

: UNITS CREATE D, DOES>
  D@          \ get 2 #s
  <AS> @      \ get current val.
  IF SWAP THEN \ flip if "true"
  */    0 <AS> ! ; \ convert, reset <AS>

BEHEAD' <AS>      \ make it local for security21

\ unit definitions remain the same
\ Usage:
\ 10 FEET . <cr> 3048 ok
\ 3048 AS FEET . <cr> 10 ok

```

§§4 Advanced methods of controlling the compiler

FORTH includes a technique for switching from compile mode to interpret mode while compiling or interpreting. This is done using the words] and [. (Contrary to intuition,] turns the compiler on, [turns it off.)

One use of] and [is to create an “action table” that allows us to choose which of several actions we would like to perform²².

For example, suppose we have a series of push-buttons numbered 1-6, and a word **WHAT** to read them.

That is, **WHAT** waits for input from a keypad; when button #3 is pushed, e.g., **WHAT** leaves 3 on the stack.

We would like to use the word **BUTTON** in the following way:

WHAT BUTTON

- 21. Headerless words are described in FTR, p. 325ff. The word **BEHEAD'** is HS/FORTH's method for making a normal word into a headerless one. See Ch. §§1§§3 for further details.
- 22. Better methods will be described in Chapter 5.

BUTTON can be defined to choose its action from a table of actions called **BUTTONS**. We define the words as follows:

```
CREATE BUTTONS ] RING-BELL OPEN-DOOR
    ENTER LAUGH CRY SELF-DESTRUCT [
: BUTTON 1- 2* BUTTONS + @ EXECUTE ;
```

If, as before, I push #3, then the action **ENTER** will be executed. Presumably button #7 is a good one to avoid²³.

How does this work?

- **CREATE BUTTONS** makes a dictionary entry **BUTTONS**.
- **]** turns on the compiler: the previously-defined word-names **RING-BELL**, etc. are looked up in the dictionary and compiled into the table (as though we had begun with **:**), rather than being executed.
- **[** returns to interactive mode (as if it were **;**), so that the next colon definition (**BUTTON**) can be processed.
- The table **BUTTONS** now contains the code-field addresses (CFA's) of the desired actions of **BUTTON**.
- **BUTTON** first uses **1-** to subtract 1 from the button number left on the stack by **WHAT** (so we can use 0-based numbering into the table — if the first button were #0, this would be unneeded).
- **2*** then multiplies by 2 to get the offset (from the beginning of **BUTTONS**) of the CFA representing the desired action.
- **BUTTONS +** then adds the base address of **BUTTONS** to get the absolute address where the desired CFA is stored.
- **@** fetches the CFA for **EXECUTE** to execute.
- **EXECUTE** executes the word corresponding to the button pushed. Simple!

23. The safety of an execution table can be increased by making the first (that is, the zero'th) action **WARNING**, and making the first step of **BUTTON** a word **CHECK-DATA** that maps any number not in the range 1-6 into 0. Then a wrong button number causes a **WARNING** to be issued and the system resets.

You may well ask “Why bother with all this indirection, pointers, pointers to pointers, tables of pointers to tables of pointers, and the like?” Why not just have nested **IF ... ELSE ... THEN** constructs, as in Pascal?

There are three excellent reasons for using pointers:

- Nested **IF...THEN**'s quickly become cumbersome and difficult to decipher (TF). They are also slow (see Ch. 11).
- Changing pointers is generally much faster than changing other kinds of data — for example reading in code overlays to accomplish a similar task.
- The unlimited depth of indirection possible in FORTH permits arbitrary levels of abstraction. This makes the computer behave more “intelligently” than might be possible with more restrictive languages.

A similar facility with pointers gives the C language its abstractive power, and is a major factor in its popularity.

§9 Strings

By now it should be apparent that FORTH can do anything any other language can do. One feature we need in any sort of programming — scientific or otherwise — is the ability to handle alphanumeric strings. We frequently want to print messages to the console, or to put captions on figures, even if we have no interest in major text processing.

While every FORTH system must include words to handle strings (see, e.g., FTR, Ch. 9) — the very functioning of the outer interpreter, compiler, etc., demands this — there is little unanimity in defining extensions. BASIC has particularly good string-handling features, so HS/FORTH and others provide extensions designed to mimic BASIC's string functions.

Typical FORTH strings are limited to 255 characters because they contain a count in their first byte²⁴. The word **COUNT**

```
: COUNT DUP 1+ SWAP C@ ; ( adr -- n adr + 1)
```

expects the address of a counted string, and places the count and the address of the first character of the string on the stack. **TYPE**, a required '79 or '83 word, prints the string to the console.

It is straightforward to employ words that are part of the system (such as **KEY** and **EXPECT**) to define a word like **\$“** that takes all characters typed at the keyboard up to a final " (close-quote — not a word but a string-terminator), makes a counted string of them, and places the string in a buffer beginning at an address returned by **PAD**²⁵.

The word **\$.** ("string-emit") could then be defined as

```
: $. COUNT TYPE ; ( adr -- )
```

and would be used with **\$“** like this:

```
$“ The quick brown fox” <cr> ok
$. The quick brown fox ok
```

Since this book is not an attempt to paraphrase **FTR**, it is strongly recommended that the details of using the system words to devise a string lexicon be studied there.

One might contemplate modifying the **FTR** lexicon by using a full 16-bit cell for the count. This would permit strings of up to 64k bytes (using unsigned integers²⁶), wasting 1 byte of memory per short (255 bytes) string. Although few scientific applications need to manipulate such long strings, the program that generated the index to this book needed to read a page at a time, and thus to handle strings about 3–5 kbytes long.

24. A single byte can represent positive numbers 0-255.

25. A system variable that returns the current starting address of the "scratchpad".

26. **FTR**, Ch. 3.

§10 FORTH programming style

A FORTH program typically looks like this

```
\ Example of FORTH program
: WORD1 ... ;
: WORD2 OTHER-WORDS ;
: WORD3 YET-OTHER-WORDS ;
...
: LAST-WORD WORDn ... WORD3
    WORD2 WORD1 ;
LAST-WORD <cr> \ run program
```

Note: The word \ means “disregard the rest of this line”. It is a convenient method for commenting code.

In other words, a FORTH program consists of a series of word definitions, culminating in a definition that invokes the whole shebang. This aspect gives FORTH programming a somewhat different flavor from programming in more conventional languages.

Brodie notes in TF that high-level programming languages are considered *good* if they require structured, top-down programming, and *wonderful* if they impose *information hiding*. Languages such as FORTRAN, BASIC and assembler that permit direct jumps and do not impose structure, top-down design and data-hiding are considered *primitive* or *bad*. To what extent does FORTH follow the norms of *good* or *wonderful* programming practice?

§§1 Structure

The philosophy of “structured programming” entered the general consciousness in the early 1970’s. The idea was to make the logic of program control flow immediately apparent,

thereby aiding to produce correct and maintainable programs. The language Pascal was invented to impose by fiat the discipline of structure. To this end, direct jumps (GOTOs) were omitted from the language²⁷.

FORTH programs are *automatically* structured because word definitions are nothing but subroutine calls. The language contains no direct jump statements (that is, no GOTO's) so it is *impossible* to write "spaghetti" code.

A second aspect of structure that FORTH imposes (or at least encourages) is *short* definitions. There is little speed penalty incurred in breaking a long procedure into many small ones, unlike more conventional languages. Each of the short words has one entry and one exit point, and does one job. This is the beaux ideal of structured programming!

§§2 “Top-down” design

Most authors of “how to program” books recommend designing the entire program from the general to the particular. This is called “top-down” programming, and embodies these steps:

- Make an outline, flow chart, data-flow diagram or whatever, taking a broad overview of the whole problem.
- Break the problem into small pieces (decompose it).
- Then code the individual components.

The natural programming mode in FORTH is “bottom-up” rather than “top-down” — the most general word appears last, whereas the definitions necessarily progress from the primitive to the complex. It is possible — and sometimes vital — to invoke a word before it is defined (“forward referencing”²⁸). The dictionary and threaded compiler mechanisms make this nontrivial.

-
- 27. Ironically, most programmers refuse to get along without spaghetti code, so commercial Pascal's now include GOTO. Only FORTH among major languages completely eschews both line labels and GOTOs, making it the most structured language available.
 - 28. The FORmula TRANslator in Ch. 11&4 uses this method to implement its recursive structure.

The naturalness of bottom-up programming encourages a somewhat different approach from more familiar languages:

- In FORTH, components are specified roughly, and then as they are coded they are immediately tested, debugged, redesigned and improved.
- The evolution of the components guides the evolution of the outer levels of the program.

We will observe this evolutionary style in later chapters as we design actual programs.

§§3 Information hiding

Information (or data) “hiding” is another doctrine of structured programming. It holds that no subroutine should have access to, or be able to alter (corrupt!) data that it does not absolutely require for its own functioning²⁹.

Data hiding is used both to prevent unforeseen interactions between pieces of a large program; and to ease designing and debugging a large program. The program is broken into small, manageable chunks (“black boxes”) called **modules** or **objects** that communicate by sending messages to each other, but are otherwise mutually impenetrable. Information hiding and modularization are now considered so important that special languages – Ada, MODULA-2, C++ and Object Pascal – have been devised with it in mind.

To illustrate the problem information hiding is intended to solve, consider a FORTRAN program that calls a subroutine

29. Rather like the “cell” system in revolutionary conspiracies, where members of a cell know only each other but not the members of other cells. Mechanisms for receiving and transmitting messages between cells are double-blind. Hence, if an individual is captured or is a spy, he can betray at most his own cell and the damage is limited.

```
PROGRAM MAIN
    some lines
    CALL SUB1(arg1, arg2, ..., argn, answer)
    some lines
    END

    SUB1(X1, ..., Xn, Y)
        some lines
        Y = something
    RETURN
    END
```

There are two ways to pass the arguments from MAIN to SUB1, and FORTRAN can use both methods.

- Copy the arguments from where they are stored in MAIN into locations in the address space of SUB1 (set aside for them during compilation). If the STATEMENTS change the values X1,...,Xn during execution of SUB1, the original values in the calling program will not be affected (because they are stored elsewhere and were copied during the CALL).
- Let SUB1 have the addresses of the arguments where they are stored in MAIN. This method is dangerous because if the arguments are changed during execution of SUB1, they are changed in MAIN and are forever corrupted. If these changes were unintended, they can produce remarkable bugs.

Although copying arguments rather than addresses seems safer, sometimes this is impossible either because the increased memory overhead may be infeasible in problems with large amounts of data, or because the extra overhead of subroutine calls may unacceptably slow execution.

What has this to do with FORTH?

- FORTH uses linked lists of addresses, compiled into a dictionary to which all words have equal right of access.
- Since everything in FORTH is a word –constants, variables, numerical operations, I/O procedures– it might seem impossible to hide information in the sense described above.
- Fortunately, word-names can be erased from the dictionary after their CFAs have been compiled into words that call them. (This erasure is called “beheading”.)

- Erasing the names of variables guarantees they can be neither accessed nor corrupted by unauthorized words (except through a calamity so dreadful the program crashes).

§§4 Documenting and commenting FORTH code

FORTH is sometimes accused of being a “write-only” language.

In other words, some complain that FORTH is cryptic. I feel this is basically a complaint against poor documentation and unhelpful word names, as Brodie and others have noted.

Unreadability is equally a flaw of poorly written FORTRAN, Pascal, C, *et al.*

FORTH offers a programmer who takes the trouble a goodly array of tools for adequately documenting code.

§§4-1 Parenthesized remarks

The word (— a left parenthesis followed by a space — says “disregard all following text up until the next right parenthesis³⁰ in the input stream. Thus we can intersperse explanatory remarks within colon definitions. This method was used to comment the Legendre polynomial example program in Ch. 1.

§§4-2 Stack comments

A particular form of parenthesized remark describes the effect of a word on the stack (or on the floating point fstack in Ch. 3). For example, the stack-effect comment (stack comment, for short)

(adr - - n)

would be appropriate for the word @ (“fetch”): it says @ expects to find an address (adr) on the stack, and to leave its contents (n) upon completion.

The corresponding comment for ! would be

30. The right parenthesis,), is not a word but a delimiter.

(n adr --) .

An fstack comment is prefaced by a double colon :: as

(:: x -- ff[x]) .

Note that to replace parentheses within the comment we use brackets [], since parentheses would be misinterpreted. Since the brackets appear to the right of the word (, they cannot be (mis-) interpreted as the FORTH words] or [.

With some standard conventions for names³¹, and standard abbreviations for different types of numbers, the stack comment may be all the documentation needed, especially for a short word.

§§4-3 Drop line (\)

The word \ (back-slash followed by space) has gained favor as a method for including longer comments. It simply means "drop everything in the input stream until the next carriage return". Instructions to the user, clarifications or usage examples are most naturally expressed in an included block of text with each line set off³² by \ .

§§4-4 Self-documenting code

By eliminating ungrammatical phrases like CALL or GOSUB, FORTH presents the opportunity —via telegraphic names³³ for words— to make code almost as self-documenting and transparent as a simple English or German sentence. Thus, for example, a robot control program could contain a phrase like

2 TIMES LEFT EYE WINK

which is clear (although it sounds like a stage direction for Brunnhilde to vamp Siegfried). It would even be possible without much

31. See, e.g., L. Brodie, TF, Ch. 5, Appendix E.

32. For those familiar with assembly language, \ is exactly analogous to ; in assembler. But since ; is already used to close colon definitions in FORTH, the symbol \ has been used in its place.

33. The matter of naming brings to mind Mark Twain's remark that the difference between the *almost-right* word and the right one is the difference between the lightning-bug and the lightning.

difficulty to define the words in the program so that the sequence could be made English-like:

WINK LEFT EYE 2 TIMES .

§§5 Safety

Some high level languages perform automatic bounds checking on arrays, or automatic type checking, thereby lending them a spurious air of reliability. FORTH has almost no error checking of any sort, especially at run time. Nevertheless FORTH is a remarkably safe language since it fosters fine-grained decomposition into small, simple subroutines. And each subroutine can be checked as soon as it is defined. This combination of simplicity and immediacy can actually produce safer, more predictable code than languages like Ada, that are ostensibly *designed* for safety.

Nonetheless, error checking – especially array bounds-checking – can be a good idea during debugging. FORTH lets us include checks in an unobtrusive manner, by placing all the safety mechanisms in a word or words that can be “vectored” in or out as desired³⁴.

34. See FTR for a more thorough discussion of vectoring. Brodie, TF, suggests a nice construct called DOER ... MAKE that can be used for graceful vectoring.

Floating Point Arithmetic

Contents

§1 Organization of floating point arithmetic	46
§§1 Fstack manipulation	47
§§2 Special constants	48
§§3 Arithmetic operators	49
§§4 Example: evaluating a polynomial	50
§§5 Optimizing: FORTH vs. FORTRAN	51
§2 Testing floating point numbers	53
§3 Mathematical functions – the essential function library	53
§4 Library extensions	54
§§1 The FSGN function	54
§§2 Cosh, Sinh and their inverses	54
§5 Pseudo-random number generators (PRNG's)	56
§§1 Testing random number generators	58
§§2 Random data structures	61

This chapter discusses various aspects of floating point arithmetic in FORTH. Our approach assumes the central processor (CPU) has a dedicated floating point co-processor (FPU) available to it, such as the 80x87 for the Intel 80x86 family; the built-in FPU on the 80486; the 68881/2 for Motorola 680x0 machines; various add-ins and clones like the Weitek, Cyrix, IIT and AMD chips; or digital signal processing and array-processing co-processor boards.

If no floating point co-processor were available, one could employ co-processor emulation routines. This is the approach taken in commercial software written in FORTH, such as the (unfortunately now-deceased) VP-Planner spreadsheet. Since this text is not a *vade mecum* for writing commercial software, but rather a handbook for using FORTH to solve computational

problems in science and engineering, we consider a co-processor essential.

§1 Organization of floating point arithmetic

FORTH was originally invented as a language for controlling machinery. It is still used extensively for this purpose, with the machines in question being as varied as industrial robots, laboratory instruments, the Hubble Space Telescope, special effects motion picture cameras, and other computers. The floating point co-processor in a typical computer is a *machine*, and any numerical calculation with floating point or complex numbers, e.g. can be organized in terms of loading operands into the coprocessor, and transferring results from it to memory. That is, FORTH can *control* the FPU through the calculation, as indicated in Fig. 3-1 below:

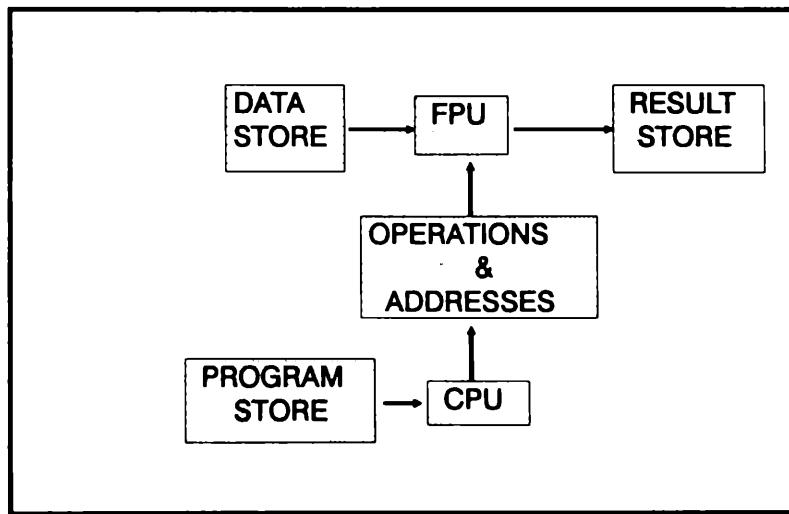


Fig. 3-1 Data flow diagram of a CPU controlling an FPU through a calculation.

We assume a stack for floating point numbers separate from the parameter or return stacks. We call this the **fstack**, and assume it has arbitrary depth. (We denote it by :: in stack comments.)

Whether the fstack should be distinct from the parameter stack is currently a subject of lively debate within the FORTH community. One faction wishes to combine the two. The other faction, including the author and most other FORTH number-crunchers, believes that to organize a floating point-intensive calculation as data flow through a dedicated coprocessor, the parameter stack must be reserved for addresses, loop indices and flags. The data fed to the coprocessor therefore has to stay elsewhere, i.e. in the data store and the fstack.

The words we shall need fall into the categories of fstack manipulation, special constants, arithmetic, tests, and mathematical functions. Their names are nearly standard¹.

§§1 Fstack manipulation

The fstack words are²

F@	(addr - - :: - - x)
F!	(addr - - :: x - -)
FDUP	(:: x - - x x)
FSWAP	(:: y x - - x y)
FDROP	(:: x - -)
FROT	(:: zyx - - yxz)
FOVER	(:: yx - - yxy)
S->F	(n - - :: - - n)
D->F	(d - - :: - - d)
F->S	(:: x - - :: - - int[x])

1. That is, the names in MMS FORTH, HS/FORTH, UniFORTH and PCFORTH are nearly the same and close to the proposed FORTH Vendors' Group (FVG) standard – see, e.g., Ray Duncan's and Martin Tracy's article in Dr. Dobb's Journal, September 1984, p. 110. The new ANSI standard does not address floating point, hence it is likely the FVG standard will become pre-eminent by default.
2. F@ and F! stand for a suite of words for fetching and storing 16-, 32-, 64-, and 80-bit numbers from/to the fstack. In HS/FORTH they have names like I16@ I16! I32@ I32! R32@R32! R64@ R64! R80@ R80! and equivalents with suffixes E or L, that transfer data from segments other than the data segment.

```
F->D      ( :: x - - : - - dint[x])
%         ( place a FP# from input stream on fstack)
```

To these we sometimes add³

```
: FUNDER  FSWAP FOVER ;
: FPLUCK  FSWAP FDROP ;
FnX      ( n = 2 - 6 | defined in code for speed)
FnR      ( n = 3 - 7 | defined in code for speed)
```

The Intel mathematics co-processors 80x87 (8087/80287/80387) and their clones incorporate a stack of limited depth (in fact 8 deep), the 87stack. It is far faster to get a number from the 87stack than from memory. Thus, as Palmer and Morse⁴ emphasize, optimizing for speed demands maximum use of the 87fstack to store intermediate results, frequently used constants, etc.

In Ch. 4 §7 we show how to extend the 87fstack into memory. The cost of unlimited fstack-depth is reduced speed when the 87stack spills over to memory.

§§2 Special constants

In defining various floating point operations it is convenient to be able to place certain constants on the fstack directly, by invoking their names. Here are some words that have proven useful:

F=0	(:: - - 0)
F=1	(:: - - 1)
F=PI	(:: - - pi = 3.14159...)
F=L2(10)	(:: - - log ₂ 10)
F=L2(E)	(:: - - log ₂ e)
F=L10(2)	(:: - - log ₁₀ 2)
F=LN(2)	(:: - - log _e 2)

3. FnX (exchange ST(n) and ST(0) on fstack) and FnR (roll ST(n) to ST(0) on fstack) are part of HS/FORTH's floating point lexicon.
4. J.F. Palmer and S.P. Morse, *The 8087 Primer* (John Wiley and Sons, Inc., New York, 1984). Hereinafter called 8087P. Basically the same principle holds for other coprocessors such as the Weitek 1167 or Transputer 800 that incorporate intrinsic fstacks. The Motorola 68881/2 have registers, hence we must synthesize an fstack in software for these chips.

§§3 Arithmetic operators

As noted in Chapters 1 and 2, FORTH arithmetic operators are words — *dumb* words. FORTH uses a distinct set of operators for each kind (16-bit integer, 32-bit integer, REAL, COMPLEX) of arithmetic, so the compiler has nothing to decide.

Languages like FORTRAN, BASIC and APL overload arithmetic operators — their meanings are context-dependent. This makes it possible to write —say— a FORTRAN expression using REAL*4, REAL*8, COMPLEX*8 or COMPLEX*16 literals and variables without worrying about how to fetch, store or convert them. Operator overloading increases the complexity of compilers and limits the speed and efficiency of compilation.

As we shall see in Chapter 5§1, FORTH enables an alternative solution in which “smart” data “know” their own types and “smart” operators “know” what kinds of data are being combined. The slightly reduced execution speed is offset by improved flexibility: *one* canned routine can work with all data types. Even better, adding this kind of “intelligence” makes no extra demands on the FORTH compiler.

The standard, dumb FORTH floating-point arithmetic operations and their actions are

F+	(:: y x -- y + x)
F-	(:: y x -- y-x)
FR-	(:: y x -- x-y)
F*	(:: y x -- y*x)
F/	(:: y x -- y/x)
FR/	(:: y x -- x/y)
FNEGATE	(:: x -- -x)
FABS	(:: x -- x)
1/F	(:: x -- 1/x)

To these it is sometimes useful to add words that do not consume all their arguments, such as F*NP (floating multiply, no pop)

F*NP (:: x y -- x y*x) ,

that are faster, more convenient, and less demanding of the stack than the phrase FOVER F*.

§§4 Example: evaluating a polynomial

Let us now write a little program to calculate something using the floating point lexicon, say a program to evaluate a general polynomial $P_N(x)$. The formula to evaluate is

$$\begin{aligned} P_N(x) &= a_0 x^0 + a_1 x^1 + \dots + a_N x^N \\ &= a_0 + x(a_1 + x(\dots + x a_N)) \end{aligned}$$

The algorithm can be represented by the (pseudo) FORTH flow diagram⁵, where █ indicates the end of the program.

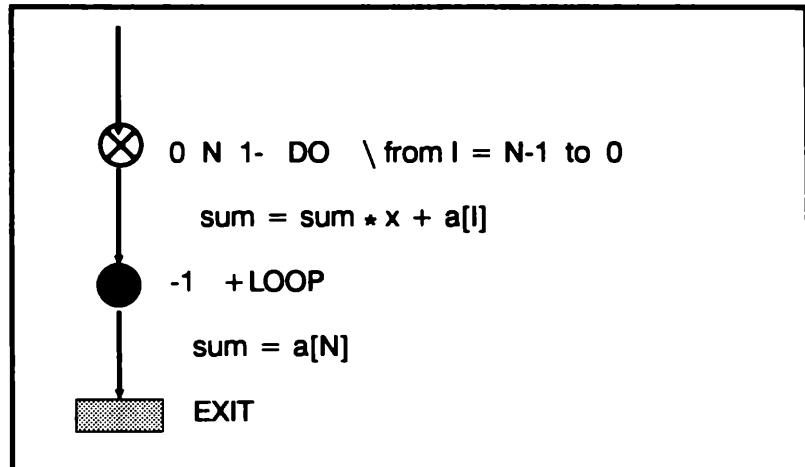


Fig. 3-2 Pseudo FORTH flow diagram of polynomial evaluation

Now we translate Fig. 3-2 above into FORTH^{6,7}:

-
- 5. See, e.g., SF. Lines trace program flow; a branch indicates a decision; at a loop ⊗ marks the beginning and • the branch back to the beginning, as in BEGIN...REPEAT or DO...LOOP.
 - 6. We assume arrays have been defined using the intrinsically typed data structures and generic operators of Ch. 5.
 - 7. The notation [x] means “the address of x”.

```

: }POLY ([a{}][x]--)          \ evaluate p(x,N)
  FINIT G@                   \ x on fstack
  DUP type@ G=0              ( ::-- x sum=0 )
  LEN@                         ( -- [a{} N)
  1 SWAP DO                   \ begin loop
    DUP I} G@                 ( ::-- x sum a[i] )
    G+ GOVER G*               ( ::-- x sum' )
  -1 +LOOP                     \ end loop
  GPLUCK                         ( ::-- sum )
  0 } G@ G+ ;                  ( ::-- p[x,n] )

```

\ Say: A{ X }POLY

Note that the function }POLY expects the addresses of its arguments on the stack, consumes them and leaves its result on the fstack. User-defined FORTH functions will in general have an interface of this sort. This will be especially true of the functions built into numerical co-processors.

Actually, such behavior is typical of subroutine linkage in most high level languages, as anyone knows who has written assembler subroutines that can be linked to compiled FORTRAN, C or BASIC. So FORTH really isn't different, only more explicit and efficient.

§§5 Optimizing: FORTH vs. FORTRAN

A simple-minded compiler will translate an expression such as

$$y = (\sin(x))^2$$

into a form requiring two function calls:

$$Y = \text{SIN}(X) * \text{SIN}(X).$$

Obviously this is silly. One of the claims often made for "optimizing" FORTRAN or C compilers is the ability to recognize an expression requiring unnecessary function calls, and to re-express it as, say,

$$\begin{aligned} \text{TEMP} &= \text{SIN}(X) \\ Y &= \text{TEMP} * \text{TEMP}. \end{aligned}$$

A globally optimizing compiler has a more extensive repertoire: usually it can recognize static expressions ("invariant code") within a loop and move them outside; and it can find and eliminate code that is never evaluated ("dead" code).

FORTH assumes a good programmer *never* overlooks trivial optimizations like this. Thus nothing in the FORTH incremental compiler or optimizer is inherently capable of recognizing silly code and eliminating it.

Optimization in FORTH takes one of several forms, that can be combined for best results. The simplest is the use of stacks and registers to avoid extra memory shuffling. Referring to the preceding bad example, we note that a simple floating-point function $f(x)$ finds its argument x on the top of the fstack, consumes it, and leaves the result in its place. A simple **F**2**, defined as

```
: F**2 FDUP F* ;
```

will then evaluate $[f(x)]^2$, with no fetch/store penalty from defining a temporary variable.

Some FORTHS can optimize by substituting inline code for jumps and returns to subroutines. In other words, by making the compiled code longer, some advantage in speed can be gained. HS/FORTH offers a recursive-descent optimizer of just this sort that –within its limitations– can optimize as well as good C or FORTRAN compilers. An optimizer-improved word consists of all the code bodies of the words in its definition, jammed end-to-end and with redundant pushes and pops deleted.

Virtually all FORTH implementations have a built-in assembler that permits defining a word in machine language. Judiciously machine-coding selected words can dramatically reduce execution time, since careful hand coding offers the ultimate performance the machine is capable of. Some versions of Pascal and C also have this ability; and of course most compiled and linked languages can link to functions and subroutines defined in machine code.

FORTH's advantage over other languages lies in making the process of designing, testing and linking hand-coded components nearly painless.

Another advantage of FORTH over other compiled languages is that one can specify which parts to optimize and which to leave as high-level definitions. This is both faster to compile and much more compact, than optimizing all of the program uniformly. The rationale of partial (sometimes called "peephole") optimization is that most programs spend 90% of their execution time in 10% of the code. This 10% is the only part of the program worth optimizing.⁸

§2 Testing floating point numbers

Analogous to the test words for integer arithmetic, we require the words **F0>** **F0=** **F0<** **F>** **F=** **F<**.

Test words leave a flag on the parameter stack depending on the relationship they discover. Moreover, these words consume one or two arguments on the fstack, following the standard FORTH practice. As a simple first example of test words, let us define **FMAX** and **FMIN** analogous to **MAX** and **MIN**:⁹

```
: XDUP  FOVER  FOVER ;
: FMAX  XDUP  F<  IF  FSWAP THEN FDROP ;
: FMIN  XDUP  F>  IF  FSWAP THEN FDROP ;
```

§3 Mathematical functions – the essential function library

Scientific programming in FORTH requires a suite of exponential, logarithmic and trigonometric functions (included with all FORTRAN systems, most BASICs, C's, APL, LISP, etc.) The minimal function library is

- 3. An example is discussed in Chapter 9, where the innermost loop of 3 nested loops is optimized (even hand-coded), and it is seen from the timings that little is to be gained by optimizing the next outer loop.
- 4. XDUP is called CPDUP in HS/PORTH's complex arithmetic lexicon.

FSQRT	(:: x -- √x)
FLN	(:: x -- ln[x])
FLOG	(:: x -- log ₁₀ [x])
F2**	(:: x -- 2 ^x)
F**	(:: x y -- y ^x)
FEXP	(:: x -- e ^x)
FSIN	(:: x -- sin[x])
FCOS	(:: x -- cos[x])
FTAN	(:: x -- tan[x])
DEG- > RAD	(:: x -- x*pi/180)
RAD- > DEG	(:: x -- x*180/pi)
FATAN	(:: x -- atan[x])
FASIN	(:: x -- asin[x])
FACOS	(:: x -- acos[x])

Machine code definitions of the above functions for the 80x87 chip will be given in Chapter 4.

§4 Library extensions

The minimal function library is easily extended. We illustrate below with the **FSGN** function and with hyperbolic and inverse hyperbolic functions. Complex extensions of the function library is deferred to Chapter 6.

§§1 The FSGN function

The most useful form of **FSGN** finds one argument *n* (from which to take an algebraic sign) on the parameter stack, and the floating point argument *x* on the fstack. We may define it using logic, as

```
: FSGN ( n -- :: x -- sgn[n]*abs[x])
      FABS 0< IF FNNEGATE THEN ;
```

§§2 Cosh, Sinh and their inverses

We now code the hyperbolic sine and cosine. The formulae are

$$\sinh(x) = \frac{1}{2}(e^x - e^{-x})$$

$$\cosh(x) = \frac{1}{2}(e^x + e^{-x})$$

and their definitions are

```
: F2/  F=1  ( :: x -- x/2 )
      FNNEGATE  FSWAP  FSSCALE  FPPLUCK ;
: HYPER  FEXP  FDUP  1/F ;    ( :: - e**x  e**-x )
: SINH  HYPER  F-  F2/ ;
: COSH  HYPER  F+  F2/ ;
```

The hyperbolic tangent is then

```
FIND XDUP 0 = ?(: XDUP  FOVER  FOVER  ; )
                  \ conditionally compile XDUP
: TANH  HYPER  ( :: - e^x  e^-x )
      XDUP  F-  F-ROT  F+  F/ ;
```

Finally, the inverse hyperbolic sine and cosine can be defined in terms of logarithms:

$$\text{arcsinh}(x) = \ln(x + (x^2 + 1)^{1/2}) , -\infty < x < \infty$$

$$\text{arccosh}(x) = \ln(x + (x^2 - 1)^{1/2}) , -\infty < x < \infty$$

The corresponding definitions are¹⁰

```
FIND F**2 0 = ?(: F**2  FDUP  F*  ; )
: ARCSINH  FDUP  F**2  F=1  F+
      FSQRT  F+  FLN ;
: ARCCOSH  FDUP  F**2  F=1  F-
      FDUP  F0<
      ABORT" x < 1 in ARCCOSH"
      FSQRT  F+  FLN ;
```

10. Note the test for $x^2 \geq 1$ in ARCCOSH, to prevent an error in FSQRT.

§5 Pseudo-random number generators (PRNG's)

The subject of computer-generated (pseudo) random numbers has been discussed extensively in the literature of computation^{11,12}. We shall confine ourselves here to translating two useful algorithms into FORTH, and discussing tests for pseudo-random number generators (PRNG's).

The first is a method called GGUBS¹³ based on the recursion

$$r_{n+1} = 16807 \times r_n \text{ MOD } (2^{31} - 1).$$

Since 32-bit modulo arithmetic is inefficient on a 16-bit processor, the program uses the 80x87 chip, and uses synthetic division to get N MOD ($2^{31} - 1$). A version that uses the 32-bit registers of the 80386/80486 would not be hard to program.

Two specialized words are needed¹⁴, that fetch/store 32-bit integers to/from the fstack from/to memory:

```
CODE I32@ <% 9B DB 07 5B 9B %> END-CODE
CODE I32! <% 9B DB 1F 5B 9B %> END-CODE
```

The program data are stored in variables rather than registers so they can be moved directly to the co-processor¹⁵.

DVARIABLE	BIGDIV
21474.83647	BIGDIV D! \ 2**31-1
DVARIABLE	DIVIS
1277.73	DIVIS DI
DVARIABLE	SEED
VARIABLE	M1 16807 M1 !
VARIABLE	M2 2836 M2 !

11. D.E. Knuth, *The Art of Computer Programming*, v. 2: *Seminumerical Algorithms*, 2nd ed. (Addison-Wesley Publishing Company, Reading, MA, 1981), Ch. 3.
12. R. Sedgewick, *Algorithms* (Addison-Wesley Publishing Company, Reading, MA, 1983).
13. P. Bratley, B.L. Fox and L.E. Schrage, *A Guide to Simulation* (Springer-Verlag, Berlin, 1983).
14. We anticipate using the FORTH assembler – see Ch. 4 §§§1.1.
15. Of course, it would have been feasible to define, via CREATE ... DOES >, data types that place the data on the 87stack, e.g. : FICONSTANT CREATE , DOES > I16@ ; But such words can't be optimized; to optimize time-critical word(s) in assembler requires VARIABLES.

The high-level FORTH program itself is^{16,17}

```
: RAND           ( :: -- seed)
    FINIT      SEED DUP I32@
    DIVIS      I32@
    XDUP       F/
    FTRUNC     FRNDINT
    FUNDER     F* FROT FR-
    M1         I16@ F*
    FSWAP      M2 I16@
    F* F- FDUP I32! ;
: RANDOM        ( :: -- random#)
    RAND BIGDIV I32@ ( :: -- seed 2**31-1)
    FSWAP FDUP F0< ( -- f :: - 2**31-1 seed)
    IF FOVER F+ THEN FR/ ;
BEHEAD" BIGDIV RAND
\ make BIGDIV, RAND local
```

To test the algorithm start with the seed 1, and generate 1000 prn's. The result should then be 522329230.

```
: GGUBS.TST 0.1 SEED DI
    1000 0 DO RANDOM LOOP
    SEED D@ D. ;
```

GGUBS.TST 522329230 ok

-
- 16. The new words are **FINIT** (initialize 80x87 chip), **FTRUNC** (set roundoff mode to truncate floating point numbers toward zero); and **FRNDINT** (round floating point number to integer). The 80x87 idiosyncratically possesses several roundoff policies: A policy is selected by setting bits 10 and 11 (numbering from 0) of the 16-bit control word using HEX **0C00 OR**. See **8087P** for details.
 - 17. **BEHEAD"** is one of several HS/FORTH words that remove dictionary entries and reclaim lost space. **BEHEAD'** beheads one word, **BEHEAD"** beheads all words, inclusive, in a range: in this case, **BIGDIV**, **DIVIS**, **SEED**, **M1**, **M2** and **RAND**.

§§1 Testing random number generators

When defining PRNGs it is always important to include a test for randomness. The simplest is called the χ^2 test: use the PRNG to generate N integers¹⁸ in the range $[0, n-1]$ and record the number of occurrences, f_s , of each integer $s = 0, 1, \dots, n-1$. If the PRNG is really random, then the probability that an integer should have any of the n values is $1/n$, hence the expected frequencies are $\langle f_s \rangle \equiv \lambda = \frac{N}{n}$. The χ^2 statistic for f_s is defined to be

$$\chi^2 = \sum_{s=0}^{\infty} \frac{1}{\lambda} (f_s - \lambda)^2, \quad \lambda = \frac{N}{n} \quad (2)$$

χ^2 should have a value roughly n ¹⁹. GGUBS passes the χ^2 test: A program to calculate this statistic (with $N = 1000$ and $n = 100$) is

```

CREATE FREQS 200 ALLOT OKLW
: IRAND RANDOM 100 S->F
  F* FROUND- F->S ;
: INIT-FREQS \ initialize freqs array
  FREQS 200 0 FILL ;

: GET-FREQS \ make frequency table
  1000 0 DO IRAND 2*
    DUP 199 > ABORT" IRAND TOO LARGE"
    FREQS + 1+!
  LOOP ;
: START.CHISQ INIT-FREQS GET-FREQS ;
: CHISQ 0 \ sum on stack
  200 0 DO I FREQS + @
    10 - DUP * +
  2 +LOOP ;

: .FREQS \display distribution
  200 0 DO I FREQS + @ I CR ..
  2 +LOOP ;

```

18. N is assumed $>>n$
19. For those interested in details, the probability a PRNG will produce the integer "s" f times is $p_f = \lambda f e^{-\lambda} / (f!)$. Thus the expected value of $(f-\lambda)^2/\lambda$ is 1, and therefore the expected value of χ^2 is n . The variance in the χ^2 statistic should then be $n(2+\lambda^{-1})$.

The results are displayed below in Table 3-1 on page 59. The mean of these χ^2 values is 99.1, and their variance σ^2 is 210. This agrees remarkably well with the theoretical formula

$$\sigma^2 = n \left(2 + \frac{1}{\lambda} \right),$$

when $n = 100$ and $\lambda = 10$.

Table 3-1 Values of χ^2 for GGUBS

START.CHISQ CHISQ . 1114 ok	START.CHISQ CHISQ . 1130 ok
START.CHISQ CHISQ . 840 ok	START.CHISQ CHISQ . 1064 ok
START.CHISQ CHISQ . 1294 ok	START.CHISQ CHISQ . 968 ok
START.CHISQ CHISQ . 650 ok	START.CHISQ CHISQ . 1052 ok
START.CHISQ CHISQ . 990 ok	START.CHISQ CHISQ . 780 ok
START.CHISQ CHISQ . 994 ok	START.CHISQ CHISQ . 842 ok
START.CHISQ CHISQ . 1072 ok	START.CHISQ CHISQ . 976 ok
START.CHISQ CHISQ . 1110 ok	START.CHISQ CHISQ . 1064 ok
START.CHISQ CHISQ . 1180 ok	START.CHISQ CHISQ . 950 ok
START.CHISQ CHISQ . 860 ok	START.CHISQ CHISQ . 956 ok
START.CHISQ CHISQ . 1080 ok	START.CHISQ CHISQ . 892 ok

In one application GGUBS was unsatisfactory because it contained correlations not revealed by the above tests. This led me to seek another PRNG with — perhaps — better properties, a longer cycle, etc. I offer it as an alternative, since — at the very least — it will enable the reader to test his applications with more than one PRNG²⁰. Here is the second PRNG:

```

\ PRNG -- B.A. WICHMAN & I.D. HILL, BYTE 3/87
VARIABLE X VARIABLE Y VARIABLE Z
: RAND-INIT 1 X ! 10000 Y ! 3000 Z ! ;
: GEN ( a b [n] -- n*a mod b ) \ hiLevel version
  DUP > R @ -ROT DUP > R
  ( -- n a b :R -- [n] b )
  */MOD DROP DUP 0<
  IF R > + ELSE RDROP THEN
  DUP R > ! ;
CODE GEN CX POP. AX POP.
\ [BX] WORD-PTR IMUL
\ CX IDIV. DX 0 I\W CMP. JGE. POSITIVE.
\ DX CX ADD.
\ >> > POSITIVE. [BX] DX MOV. END-CODE
(Ex.: 171 30269 X GEN )

```

: RANDOM FINIT FTRUNC
 171 30269 DUP S>F X GEN I16@ FRV
 172 30307 DUP S>F Y GEN I16@ FRV
 F+
 170 30323 DUP S>F Z GEN I16@ FRV
 F+ FRAC ;
 \FRAC takes the fractional part of a number
 \FTRUNC specifies rounding toward 0

The corresponding χ^2 results are given below in Table 3-2.

Table 3-2 χ^2 for Wichman-Hill PRNG

START.CHISQ CHISQ . 846	START-CHISQ CHISQ . 1172
START.CHISQ CHISQ . 1036	START-CHISQ CHISQ . 954
START.CHISQ CHISQ . 852	START-CHISQ CHISQ . 908
START-CHISQ CHISQ . 858	START-CHISQ CHISQ . 856
START-CHISQ CHISQ . 770	START-CHISQ CHISQ . 930
START-CHISQ CHISQ . 882	START-CHISQ CHISQ . 868
START-CHISQ CHISQ . 918	START-CHISQ CHISQ . 858
START-CHISQ CHISQ . 956	START-CHISQ CHISQ . 912
START-CHISQ CHISQ . 1202	START-CHISQ CHISQ . 952
START-CHISQ CHISQ . 1112	START-CHISQ CHISQ . 1016
START-CHISQ CHISQ . 778	

Interestingly, the mean of the χ^2 statistic for 21 tests is 93.5, perhaps a bit low, but not outrageously so; however, the variance in χ^2 is suspiciously small — only 135 vs. the expected 210. This may mean the distribution is excessively even!

One very useful test for randomness involves constructing a random walk — that is, a sequence of integers generated by the rule (r_n is the n 'th PRN)

$$x_{n+1} = x_n + \begin{cases} 1 & \text{if } r_{n+1} > 0.5 \\ -1 & \text{if } r_{n+1} \leq 0.5 \end{cases} \quad (3)$$

and taking the discrete Fourier transform (DFT) of the sequence²¹. Any serial correlations will show up as periodicities, with periods smaller than N, in the DFT of x_n .

§§2 Random data structures

The prng's we have discussed so far produce prn's uniformly distributed on the interval [0,1]. What if we want prn's that are distributed according to the normal distribution on $(-\infty, \infty)$, or according to some other standard distribution function of mathematical statistics?

There are algorithms for generating prn's whose distribution function is one of a few standard ones; however, in general one must resort to brute force. We now engage in a brief mathematical digression, before showing how prn's with distribution function²² $d\rho(\xi) = \theta(1 - \xi)d\xi$ can be converted to prn's with an arbitrary distribution function $d\rho(x) = f(x)dx$.

We suppose there is a function $X(\xi)$ that converts uniform prn's to prn's distributed according to $f(x)$. But if any of this is to make sense, the inverse function $\xi(x)$ must also exist, since there is nothing special about one distribution relative to another²³. The condition that both functions exist is $\frac{d\xi}{dx} \neq 0$.

Then²⁴

$$\begin{aligned} f(x) &= \int_0^1 d\xi \delta(x - X(\xi)) \\ &= \int_0^1 d\xi \delta(x - X(\xi(x) + \xi - \xi(x))) \end{aligned} \tag{4}$$

which, using standard manipulations, we can evaluate as

- 21. Using, e.g., the Fast Fourier Transform program from Chapter 8§2.
- 22. That is, random numbers uniformly distributed from 0 to 1.
- 23. Except in case of computation, of course.
- 24. Here $\delta(x)$ is the so-called Dirac δ -function. See any standard text on distributions.

$$f(x) = \int_0^1 d\xi \delta \left[(\xi - \xi(x)) \frac{dX}{d\xi} \right] = \frac{d\xi}{dx} \quad (5)$$

The ordinary differential equation 5 has the formal solution

$$\xi = \int_{x(0)}^{X(\xi)} dx f(x) \quad (6)$$

that defines the new prn's distributed according to $f(x)dx$, if ξ is a prn uniformly distributed on [0,1]. In other words, we have to solve a (usually) transcendental equation to calculate $X(\xi)$.

Since most simulation problems demand a *lot* of prn's, it is no use solving Eq. 6 in real time. The better solution is to define a large enough table of the X's, and look them up according to ξ . In this case we actually want an integer PRNG uniformly distributed on [0,N-1], where the table has N entries.

```
\c HARVARD SOFTWORKS 1986, ALL RIGHTS RESERVED.
HEX
1 VAR A 2 VAR B 03FF VAR MX

: RANDOM A B + MX OVER U<
  IF MX 1+ - THEN
  2* MX OVER U<
  IF MX - THEN
    B IS A DUP IS B ;
: RANDOMIZE (seed1 seed2 #bits--)
  2 MAX OF MIN      \#bits truncated to range 2-16
  1 SWAP SAL 1- IS MX
  MX MOD IS A
  MX MOD IS B
  5 0 DO RANDOM DROP LOOP ;
```

Fig. 3-3 PRNG supplied with HS/FORTH

In Fig. 3-3 above we exhibit a PRNG that produces 16-bit integers uniformly distributed on $[0, 2^N - 1]$. I have no idea of its *modus operandi* or its origin, but it passes the χ^2 tests described above.

We also need a way to invoke user-defined functions: the method we have found is shown in Fig. 3-4 below.

```
VARIABLE <F>
: USE( [COMPILE] ' CFA <F> ! ;
: F(X) <F> EXECUTE@ ;
BEHEAD' <F> \ make <F> local
```

Fig. 3-4 Protocol for function usage in FORTH

We also require a word analogous to , ("comma") that stores 32-bit floating point numbers in the parameter field of a word:

```
: F32, HERE-L 4 ALLOT R32! ;
\ store a 32-bit # from the fstack in the first
\ available place in the dictionary
```

With these auxiliary definitions we are in a position to define a word that creates tables of random variates and traverses them in a random order:

```
\ Defining word for tables of pm's distributed according to a given distribution:
\ P(x < X(xi)) = xi defines X(xi).
\ Note: the first entry is automatically 0.
\ fn.name converts xi to X(xi)
\ dist.name is the name of the random N-long table created by )DISTRIBUTION

: SHAKE.UP (#bits --) TIME@ XOR -ROT XOR ROT RANDOMIZE;
\ randomize seeds
: )DISTRIBUTION: DUP LG2 (- - N #bits = lg[N])
SHAKE.UP \ Initialize pmg
CREATE F=0 F32, \ make first entry
(N--) 1 DO \ N entry table
  I S->F I' S->F F/ \ xi on fstack
  F(X) F32, \ evaluate X(xi) and !
LOOP
DOES> RANDOM 4* + R32@ ;
BEHEAD" A RANDOM \ make these words local
\ Usage: USE( fn.name N )DISTRIBUTION: dist.name
```

In Fig. 3-5 below we exhibit a frequency plot of 10,000 random variates drawn from a (rather coarse) table of 64 entries, according to the distribution $p(x)dx = xe^{-x} dx$, together with $p(x)$.

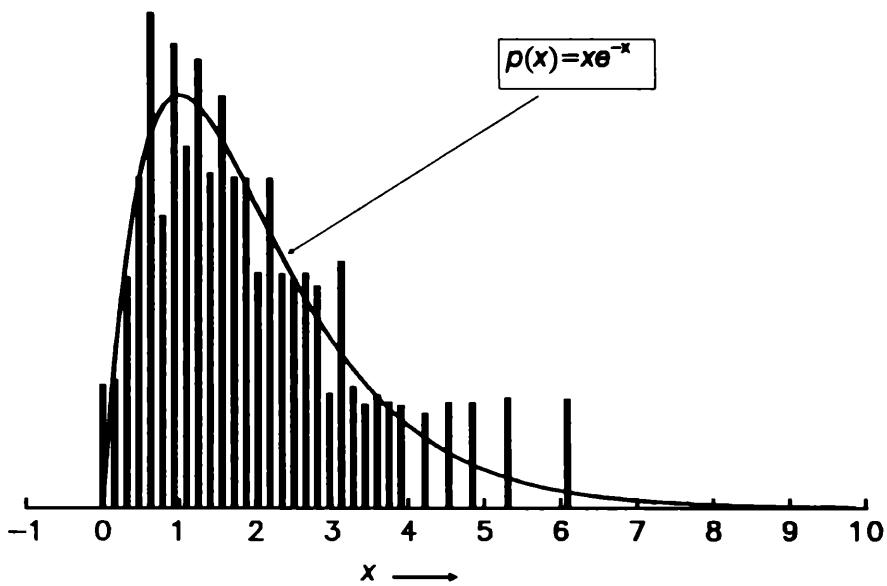


Fig. 3-5 Random variates from table of 64 from distribution $p(x) = xe^{-x}$

With slightly more elaboration we can arrange for each table to have a unique prng, thus minimizing correlations between tables. Because the resulting data structure is nearly an “object”, it is worthwhile to see how this may be done.

To make the prng unique, all that is necessary is to make the seed **VARs** unique to a table, and to redefine **RANDOMIZE** and **RANDOM** to know how to get a given table’s seeds. We see that **RANDOM** is invoked only in the runtime code for the structure. This means it has access to the base address, hence the **VARs** can be replaced with cells in the table, and the seeds planted there.

The 80x87 Family

Contents

§1 Internal 80x87 stack manipulation	67
§§1 The FORTH assembler	67
§§2 Using MS-DOS DEBUG	68
§2 Memory usage (storage and retrieval)	70
§3 Arithmetic words	72
§4 Special constants	73
§5 Test words	74
§6 Mathematical functions	76
§7 Extending the intrinsic 80x87 stack	80
§8 Clone wars	84

When we speak of the 80x87 mathematical co-processor family, we include the original Intel chips, the Cyrix D387 and IIT C287 and C387 chips, and the AMD 80287 and 80387 clones, as well as the on-chip floating point unit found on the Intel 80486 chips.

We now describe some features of the 80x87 floating point co-processors (FPUs) that affect scientific programming on IBM-PC compatible machines¹. The 8087 chip complements the Intel 8088/8086 CPU family, the 80287 works with the 80286 series, the 80387 with the 80386, and the 80486 includes an on-chip FPU. The 8087 chip is connected pin-for-pin to the 8088/8086. (The details are given in *The 8087 Primer*².) The interface and instruc-

-
1. Although we confine ourselves to the 80x87 chip, the Motorola 68881/2 coprocessors can be programmed in the same general manner to achieve floating point capabilities rivalling VAX® minicomputers.
 2. J. Palmer and S. Morse, *The 8087 Primer* (John Wiley & Sons, NY (1984), hereafter referred to as 8087P).

tion set automatically take care of bus arbitration (that is, which chip has access to the memory) and interrupts, in order to be sure that the CPU and 80x87 do not perform conflicting operations.

Instructions for the 80x87 are always appended to a code called “escape” (ESC, D8h) that alerts the coprocessor and diverts control to it. The MS-DOS assembler MASM and debugger DEBUG will automatically assemble this code with 80x87 instructions, so the user does not need to worry about including ESC except to be aware that it is happening. (Of course, a FORTH system that lacks 80x87 assembler extensions will need to include ESC explicitly to generate 80x87 codes.)

We shall see in Ch. 4 §1.1 how to use the FORTH assembler, and in Ch. 4 §1.2 how to use DEBUG³, to extend a FORTH system for 80x87 operations if they are not already included as a floating point lexicon.

The 80x87 machine code instruction set includes instructions for moving numbers to the registers from memory and *vice-versa*, as well as from one 80x87 register to another. The internal moves are of course much faster than those to or from external memory.

Advanced programming methods — such as recursive algorithms — require an fstack of unlimited depth. The designers of the 8087 anticipated the need for fstack extension and included instructions for this purpose. Unfortunately, the instructions were not well thought out⁴ so a moderately complex software fstack extension manager is needed to augment them. We design such a manager in Ch. 4 §7.

3. A treasure included with MS-DOS

4. see 8087P, p. 93ff.

§1 Internal 80x87 stack manipulation

The 80x87 is organized around a stack of 8 80-bit registers (the 87stack). The 8-deep stack can be subdivided into smaller stacks for special purposes, but this is only useful when coding in assembler for speed⁵.

We begin with words for performing 80x87 stack manipulation analogous to those defined for parameter stack. These are **FDUP** **FSWAP** **FDROP** **FROT** **FOVER** .

How are we to define them in terms of machine code primitives?

§§1 The FORTH assembler

Every FORTH worthy of the name includes an assembler, usually set up as an alternate vocabulary⁶ called **ASSEMBLER**. The assembler allows direct definition of a new word in terms of machine codes, which are referred to using standard mnemonics. A typical FORTH assembly language definition (we now specialize to HS/FORTH) for @ would have the form⁷

```
CODE @ BX [BX] MOV. END-CODE
```

A **CODE** definition is a machine-coded subroutine somewhere in memory. To use it, the compiler has to know where it is and insert appropriate unconditional JUMP (JMP) and RETURN (RET) instructions in the machine code representation of the calling program.

Here is what happened in the **CODE** version of @ above:

- The defining word **CODE** set up a dictionary entry with the name @, with an appropriate pointer and JUMP instructions to make the newly defined word run the code sequence comprising the definition.

5. see 8087P, p. 87ff.

6. Vocabularies are a method for subdividing the dictionary.

7. BX is a CPU register, and [BX] means "the memory location whose address is in BX".

HS/PORTH uses a naming convention in which assembler mnemonics end with a period, e.g. **MOV.** . Also, HS/PORTH makes the TOS the BX register, to reduce the number of pushes and pops needed to execute simple words.

- The word **END-CODE** cleans up the loose ends by adding the obligatory RET instruction sequence and turning off the compiler. (That is, **END-CODE** installs “**NEXT**”.)
- **END-CODE** is thus the analog of ; as **CODE** is of : .

Consider now the word **FROT** whose Intel assembly language definition would be

```

FROT:           ; entry point
FWAIT          ; hold 8086 operations
FXCH ST(1)     ; swap TOS and NOS
FWAIT          ; hold 8086 operations
FXCH ST(2)     ; swap TOS and ST(2)
RET            ; return

```

In HS/FORTH assembler, the definition becomes

```

CODE FROT FWAIT. 1 FXCH. FWAIT. 2 FXCH.
END-CODE

```

Note: the definition includes FWAIT (the same as WAIT), an instruction that makes the 8086 CPU wait for the FPU to complete its work before attempting to access the memory. If WAIT were omitted and the CPU accessed the memory, it could store incomplete results.⁸

§§2 Using MS-DOS DEBUG

The FORTH assembler, with its 80x87 extension, lets us develop machine-coded words while retaining Intel mnemonics for documentation, at the price of loading and compiling the entire **ASSEMBLER** lexicon. But if we know the actual machine code bytes we can bypass the assembler by entering the (hexadecimal) codes directly into a **CODE** definition. Most FORTH

8. The design of the 80286, 80386 and 80486 eliminates this problem, consequently FWAIT is not required when assembling 80287/80387/80487 machine code. See, e.g., John H. Crawford and Patrick P. Gelinger, *Programming the 80386* (Sybex, San Francisco, 1987). HS/FORTH allows the user to choose which class of machine to assemble for, when loading the 80x87 assembler extension. The 80287+ option simply defines FWAIT. as a null word.

systems, in addition to an assembler, provide a way to insert machine codes – hex numbers – directly into the code field of a word. HS/FORTH, e.g., uses the words <%> and %> to enclose the hex codes being inserted.

The problem is, how do we find out what these hex codes are?

The simplest way is to use DEBUG⁹ to generate (HEX) code sequences. Rather than try to explain, we shall illustrate by recording and annotating the DEBUG session for the word **FROT**:

C> DEBUG		starts DEBUG
-A100		Assemble from 100h
3B01:0100 FWAIT		enter assembler
3B01:0101 FXCH ST(1)		mnemonics
3B01:0103 FWAIT		
3B01:0104 FEXCH ST(2)	^ Error	DEBUG notes a typo
3B01:0104 FXCH ST(2)		
3B01:0106		no more, <cr> stops assembly
-U 100 105		Unassemble to check
3B01:0100 9B	WAIT	Hold CPU
3B01:0101 D9C9	FXCH	ST(1) (87: a b c -- a c b)
3B01:0103 9B	WAIT	Hold up CPU
3B01:0104 D9CA	FXCH	ST(2) (87: a c b -- b c a)
-D 100 105		Dump to get hex codes
3B01:0100 9B D9 C9 9B D9 CA		
-Q		Quit session

From the Dump (or Unassembly) we find code bytes 9B D9 F7 D9 C9 F6 D9 C9 which can be inserted directly into the definition of **FROT**:

9. sec, e.g., R. Lafore, *Assembly Language Primer for the IBM PC & XT* (Plume/Waite -New American Library, New York, 1984). Complete operating systems often include a code debugger that permits assembly; disassembly; modifying the contents of selected memory locations; setting breakpoints; and running programs under debugger control.

```
CODE FROT <% 9B D9 F7 D9 C9 F6 D9 C9 %>
END-CODE  ( :: a b c - - b c a )
```

The rest of the 80x87 stack words,

FDUP FSWAP FDROP FOVER F-ROT

whose assembler definitions are:

```
CODE FDUP      FWAIT. 0 FLD. END-CODE
CODE FSWAP     FWAIT. 1 FXCH. END-CODE
CODE FDROP     FWAIT. 0 FSTP. END-CODE
CODE F-ROT     FWAIT. 2 FXCH.
               FWAIT. 1 FXCH.           END-CODE
```

can be defined similarly in 8086/8087 machine code using the DEBUG program or a reference manual for the chip (e.g. 8087P) to determine the hex codes.

§2 Memory usage (storage and retrieval)

The 80x87 instruction set includes codes for loading ST(0) from memory and storing ST(0) to memory. The former involves a “push” and the latter may or may not involve a “pop”, from the 87stack.

For the moment we need words to retrieve and store 16-bit integers, short reals (32 bit) and temporary reals (80 bit). These have mnemonics FILD (“load integer to ST(0)”), FISTP (“store integer and pop ST(1) into ST(0)”), FLD, FSTP respectively. Typical (HS)/FORTH assembler definitions are

```
CODE I16@
      FWAIT. [BX] WORD-PTR FILD.
      [BX] POP.
      FWAIT.
END-CODE
CODE I16!
      FWAIT. [BX] WORD-PTR FISTP.
      [BX] POP.
      FWAIT.
END-CODE
```

Similarly, **I32@** and **I32!** can be defined by replacing **WORD-PTR** with **DWORD-PTR**. To define **I64@** and **I64!** –assuming these are needed– replace **DWORD-PTR** with **QWORD-PTR**. The 32-, 64- and 80-bit floating point analogues **R32@**, **R32!**, **R64@**, **R64!**, **R80@** and **R80!** are defined by replacing **FILD** by **FLD** and **FISTP** by **FSTP**, and using **DWORD-PTR**, **QWORD-PTR** or **TBYTE-PTR**¹⁰ as appropriate.

We also need words to load the 87stack from the stack and *vice versa*. In HS/FORTH, the top of the parameter stack is actually the **BX** register on the CPU. There is no machine instruction for loading the 87stack directly from a CPU register. Thus, we must first transfer the contents of **BX** to memory and thence to the 87stack. The inverse operation also must proceed through a memory location. The data-transfer words are named in an obvious way **S->F** and **F->S**. HS/FORTH defines them directly in machine code, manipulating the CPU register **BP** that points to the top of the CPU stack. That is, HS/FORTH uses two bytes immediately below NOS as the intermediate memory cell.

Here we define **S->F** and **F->S** directly in high-level FORTH by wasting a little memory for a (hidden) temporary variable:

```
VARIABLE TEMP
: S->F ( n -- ::-- float[n])
    TEMP !           \ TOS -> TEMP
    TEMP I16@ ;     \ TEMP -> ST(0)
: F->S ( ::x -- -- int[x])
    TEMP I16!        \ ST(0) -> TEMP
    TEMP @ ;         \ TEMP -> TOS
    BEHEAD' TEMP     \ hide address of TEMP
```

Faster machine code versions of **S->F** and **F->S** are¹¹

```
CODE S->F TEMP +[] BX MOV. BX POP.
FWAIT. TEMP +[] I16 FILD. END-CODE
```

```
CODE F->S BX PUSH. TEMP +[] FISTP.
BX TEMP +[] MOV. END-CODE
```

10. "Ten-byte pointer". Note HS/FORTH appends a period ":" to most Intel mnemonics.

11. **TEMP +[]** is HS/FORTH's phrase to assemble a named memory address.

These definitions will satisfy our present needs for storing and retrieving from the 87stack.

Note: a substantial gain in speed can be achieved with the 80386/80387 and 80486 families, by using instructions that effect 32-bit wide transfers¹².

§3 Arithmetic words

FPU (80x87) arithmetic is generally performed with the maximum precision allowed by the (80-bit) size of the registers¹³.

As noted in §4.2, the 80x87 allows 3 floating point representations for storage and retrieval in external memory: 32 bit (single precision), 64 bit (double precision) and 80 bit (“temporary real”).

To conserve memory we generally use 32 bit floating point numbers (REAL*4 in the old FORTRAN parlance) unless the nature of the calculation demands retention of more significant figures to prevent roundoff errors.

The FORTH arithmetic words we shall need are

F+ F- FR- F* F/ FR/ FNNEGATE FABS FSGN

whose definitions are (CODE and END-CODE are assumed)

F+	FWAIT.	FADDP.	(87: a b -- a+b)
F-	FWAIT.	FSUBP.	(87: a b -- a-b)
FR-	FWAIT.	FSUBRP.	(87: a b -- b-a)
F*	FWAIT.	FMULP.	(87: a b -- a*b)
F/	FWAIT.	FDIVP.	(87: a b -- a/b)
FR/	FWAIT.	FDIVRP.	(87: a b -- b/a)

12. See, e.g., John H. Crawford and Patrick P. Gelsinger, *Programming the 80386* (Sybex, San Francisco, 1987).
13. Although it is possible to force artificially 80x87 precision to 24 mantissa bits to simulate arithmetic performed on other machines (that is, to compare results while debugging), I see no virtue in a mode that slows up calculations while diminishing precision and refer the reader to refs. 2 or 8.

```

FNEGATE FWAIT.    FCHS.      ( 87: a .. -a)
FABS   FWAIT.    FABS.      ( 87: a .. |a|)

: FSIGN      ( n .. 87: x .. |x| *sgn[n] )
  FABS  0< IF FNEGATE THEN ;

```

§4 Special constants

For convenience the designers of the 8087 chip have arranged fast loading of certain constants into ST(0) of the fstack (TOS). The words that place these constants on the fstack, and the corresponding assembler mnemonics and (hex) codes are shown in Table 4-1 below.

Table 4-1 Special Constants

word	const.	mnemonic	codes
F=0	0	FLDZ	9B D9 E5
F=1	1	FLD1	9B D9 E8
F=PI	$\pi = 3.14\dots$	FLDPI	9B D9 E5
F=L2(10)	$\log_2 10$	FLDL2T	9B D9 E9
F=L2(E)	$\log_2 e$	FLDL2E	9B D9 EA
F=L10(2)	$\log_{10} 2$	FLDLG2	9B D9 EC
F=LN(2)	$\log_e 2$	FLDLN2	9B D9 ED

§5 Test words

We need to be able to determine the algebraic sign of a floating point number, as well as whether one is larger than another. The 80x87 chip has 4 instructions for this purpose, whose mnemonics are FTST, FCOM, FCOMP and FCOMPP; they are shown below in Table 4-2.

Table 4-2 Machine language floating point tests

mnemonic	comparison	pop?
FTST	ST(0) to 0	no
FCOM	ST(1) to ST(0)	no
FCOMP	ST(1) to ST(0)	pop once
FCOMPP	ST(1) to ST(0)	pop twice

The results of these comparisons are encoded as bits C3 (14) and C0 (8) of the 16-bit 80x87 STATUS register. In order to get these bits by bit-masking techniques, the STATUS register must be moved to the parameter stack¹⁴. This is done with the the 80x87 instruction FSTSW via the assembler sequence

```
VARIABLE F.STATUS
CODE FSTSW BX PUSH.
F.STATUS +[] FSTSW.
BX F.STATUS +[] MOV.
END-CODE
BEHEAD' F.STATUS
```

14. Note: the 80387 includes a new instruction whereby the status control word can be moved directly into the AX register of the 80386 CPU. The hex codes for FSTSW AX are DF E0.

Now we have to consider how to bit-mask the status integer left on the stack by **FSTSW**. We use the logical **AND** with 4000h or 0100h (**Exercise**: Why?) to pick out C3 and C0. Now from 8087P¹⁵ we have the truth table 4-3 below:

Table 4-3 Truth table of test status bits (x is the operand – 0 or ST(1))

Condition	C3	C0
ST(0) > x	0	0
ST(0) = x	1	0
ST(0) < x	0	1

Now we are in a position to define the test words **F0>**, **F0=**, **F0<**, as well as **F>**, **F=**, **F<**. We have¹⁶

CODE FTST FWAIT. FTST. 0 FSTP. END-CODE
 CODE FCOMPP FWAIT. FCOMPP. END-CODE

HEX

```

: FTSTP    FTST    FSTSW ;  

: F0>    FTSTP    4100 AND NOT 0> ;  

: F0=    FTSTP    4000 AND 0> ;  

: F0<    FTSTP    0100 AND 0> ;  

: F<    FCOMPP    FSTSW    4100 AND NOT 0> ;  

: F=    FCOMPP    FSTSW    4000 AND 0> ;  

: F>    FCOMPP    FSTSW    0100 AND 0> ;
    
```

DECIMAL

15. sec, e.g., Table 4.1
16. **Note:** the test words **F<** and **F>** are defined opposite to **F0>** and **F0<**. This reversal of directions is *not* a typographical error: it is *demanded* by the operation of **FCOMPP** – see 8087P.

§6 Mathematical functions

We now proceed to develop a suite of special functions for the 80x87 FPU. These will include the usual trigonometric functions, logarithms, and exponentials. We retain initial **F**s in the names to remind us the FPU is being used. The functions are given in Table 4-4 below:

Table 4-4 *Mathematical function primitives*

name	action	code(s)	mnemonic
FSQRT	(87: x - - \sqrt{x})	9B D9 FA	FWAIT FSQRT
FY*LG2X	(87: y x - - $y \cdot \log_2[x]$)	9B D9 F1	FWAIT FYL2X
FY*LG2XP1	(87: y x - - $y \cdot \log_2[x + 1]$)	9B D9 F9	FWAIT FYL2XP1
F2XM1	(87: x - - $2^x - 1$)	9B D9 F0	FWAIT F2XM1

These primitive functions allow us to define the logarithms and exponentials. To get $\log_2(x)$, for example, we need to decide whether x lies between 0 and 2: this can best be accomplished with the sequence (we assume x is already on the 87stack)

```
: F=2 F=1 FDUP FSSCALE FPLUCK ; (87:--2)17
: LOG.TST FDUP F0> NOT
    ABORT" Can't take log(-|x|) !!" ;
: FLG          (87: y x-- y*lg[x] )
    LOG.TST FDUP F=2 F<
    IF      F=1 F-  FY*LG2XP1
    ELSE    FY*LG2X THEN ;
```

17. See below for a discussion of **FSSCALE**.

```
: FLN    F = LN(2)      FSWAP  FLG ;
: FLOG   F = L10(2)     FSWAP  FLG ;
```

Now we can use the fundamental definition of exponentiation to define both the operation of raising an arbitrary positive number to a real power, as well as the standard mathematical function e^x :

```
: F2** (87: x -- 2**x)  F2XM1 F=1 F+ ;
: F**  (87: x y -- y**x) FLG F2** ;
: FEXP (87: x -- exp[x]) F=L2(E) F* F2** ;
```

We now have only to implement the trigonometric and inverse-trigonometric functions. We need (TREAL) 10-byte constants:

```
: F, HERE 10 ALLOT R80I ;
: FCONSTANT CREATE F, DOES> R80@ ;
```

Also, for simplicity, we define FORTH functions for degree/radian conversions and conversely:

```
FINIT F=PI 180 S->F F/      (87: -- p/180)
FCONSTANT PI/180          \ make constant
: DEG->RAD PI/180 F* ;
: RAD->DEG PI/180 F/ ;
```

The 80x87 chip has a fast way to multiply or divide by powers of 2, called **FSCALE**. The **CODE** definition is

```
CODE FSCALE <% 9B D9 FC %> END-CODE
```

FSCALE adds ST(1) to the (powers-of-2) exponent of ST(0). Thus, e.g., we can write fast divide- and multiply-by-2 instructions:

```
: F2* F=1 FSWAP FSCALE FPLUCK ;
: F2/ F=1 FNNEGATE FSWAP FSCALE FPLUCK ;
```

Now, according to **8087P**, we may evaluate trigonometric and inverse-trigonometric functions using the instructions

```
CODE FPTAN FWAIT. <% D9 F2 %> END-CODE
CODE FPATAN FWAIT. <% D9 F3 %> END-CODE
```

The 8087 and 80287 implement an *unnormalized* tangent function, whose effect is (87: z -- y x), with $\tan(z) = y/x$. Let us define

$$\frac{y}{x} = \tan(z/2) \quad (1)$$

That is, we obtain the tangent of half the angle. The other trigonometric functions can be computed in software using the identities

$$\sin(z) = \frac{2(y/x)}{1 + (y/x)^2} \quad (2)$$

$$\cos(z) = \frac{1 - (y/x)^2}{1 + (y/x)^2} \quad (3)$$

$$\tan(z) = \frac{\sin(z)}{\cos(z)} \quad (4)$$

A further problem created by the 8087/80287 instruction set is that the argument z must lie in the range $0 < z < \pi/4$. Thus we must shift the argument to this range, using a special instruction **FPREM** (“exact” partial remainder¹⁸) that can be used to extract multiples of π :

```
CODE FPREM FWAIT. <% D9 F8 %> END-CODE
: XDUP FOVER FOVER ;
: ENUF? FSTSW 1024 AND 0= ; \ bit C2 =0?
: FNORM (87: x k -- x mod k) FSWAP
  BEGIN FPREM ENUF? UNTIL FPLUCK ;
  \ extract multiples of k
```

Here is how we code the tangent in high-level FORTH:

Pseudocode version:

$x=0$ is an exception – set $\tan=0$ and exit.
 $x < 0$? Save sign as a flag on stack.
 reduce by multiples of π .
 \ cont'd ...

18. See 8087P, p. 100ff

x in 1st quadrant ($\pi/2 < x < \pi$)? Flag, reduce by $\pi/2$
 x in 1st octant ($\pi/4 < x < \pi/2$)? Flag, reduce by $\pi/4$

```
\ HIGH LEVEL FORTH VERSION
: REDUCE    (:87: x k -- x mod k -- f)
  XDUP F> DUP IF F- ELSE FDROP THEN ;
: FTAN      (87: x -- tan[x]) FDUP F0=
  IF EXIT THEN          \ tan=0
  FDUP F0< FABS        ( -- fsgn 87: -- |x| )
  F=PI FNORM           \ 0 < x < pi
  F=PI F2/  REDUCE     ( -- fsgn f1q )
  F=PI F2/ F2/ REDUCE  ( -- fsgn f1q f1o )
  FPTAN F/              (87: |x|-- tan[x] )
  IF F=1 XDUP F+ F-ROT F- F/ THEN
    \ adjust for octant
  IF 1/F FNEGATE THEN   \ adjust for quadrant
  IF FNEGATE THEN ;     \ adjust sign
```

The remaining trigonometric functions (sine, cosine, secant, cosecant) can easily be defined in terms of $\tan(z/2)$. For example, here are **FSIN** and **FCOS**:

```
: FSIN  F2/ FTAN FDUP FDUP F* F=1 F+
  FR/ F2* ;
: FCOS  F2/ FTAN FDUP F* F=1 FOVER F-
  FSWAP F=1 F+ F/ ;
```

Note: The 80387 improves on the 8087/80287 by eliminating the need to adjust the argument in software. Further, the tangent produced by the 80387 is normalized (that is, $x = 1.0$ in Eq. 4.1 above). Finally, the 80387 has instructions FSIN, FCOS and FSINCOS built in, so all the software emulation is unnecessary.

We define inverse-trigonometric functions using **FPATAN** defined previously, whose action is ($87: y x -- \arctan[y/x]$). The 80387 has no additional instructions for inverse trig functions, relative to the 8087/80287, so the same code fits all.

To calculate the inverse functions, we make use of standard identities (the forms chosen minimize roundoff error). Thus,

$$\text{Arcsin}(z) = \text{Arctan}\left(\frac{z}{\sqrt{(1-z)(1+z)}}\right) \quad (5)$$

$$\text{Arccos}(z) = 2 \arctan \left\{ \left(\frac{1-z}{1+z} \right)^{\frac{1}{2}} \right\} \quad (6)$$

```

: FATAN F=1 FPATAN ;
: FASIN FDUP FABS F=1 F>
    ABORT' argument of arcsin > 1"
    F=1 FOVER FDUP F* F- FSQRT
    F/ FPATAN ;

: FACOS FDUP FABS F=1 F>
    ABORT' argument of arccos > 1"
    FDUP F=1 FR- FSWAP F=1 F+ F/
    FSQRT FPATAN ;

```

Note: the argument of $\arcsin(x)$ or $\arccos(x)$ must be smaller than 1 in absolute value – hence we include a bounds check to avoid taking the square root of a negative number.

§7 Extending the intrinsic 80x87 stack

As promised in the beginning of the chapter, we now design a fstack manager in software that allows more than 8 cells. First we examine the structure of the 80x87 stack (87stack).

From 8087P we see that the 8 registers in the 87stack are organized as a circular stack. A 3-bit pointer, ST, records which physical register is actually TOS. The instruction set allows ST to be incremented (FINCSTP) or decremented (FDECSTP) modulo 8. The 87stack is shown below in Fig. 4-1 on page 81.

A FLD instruction decrements ST (mod 8) before storing whereas a FSTP instruction increments ST. To build our software fstack we need to do the following things:

- When the 87stack gets full, put ST(7) on the memory extension and vice-versa.
- Keep track of how many numbers are on the 87stack.
- Keep track of where we have stored the last number removed from the 87stack.

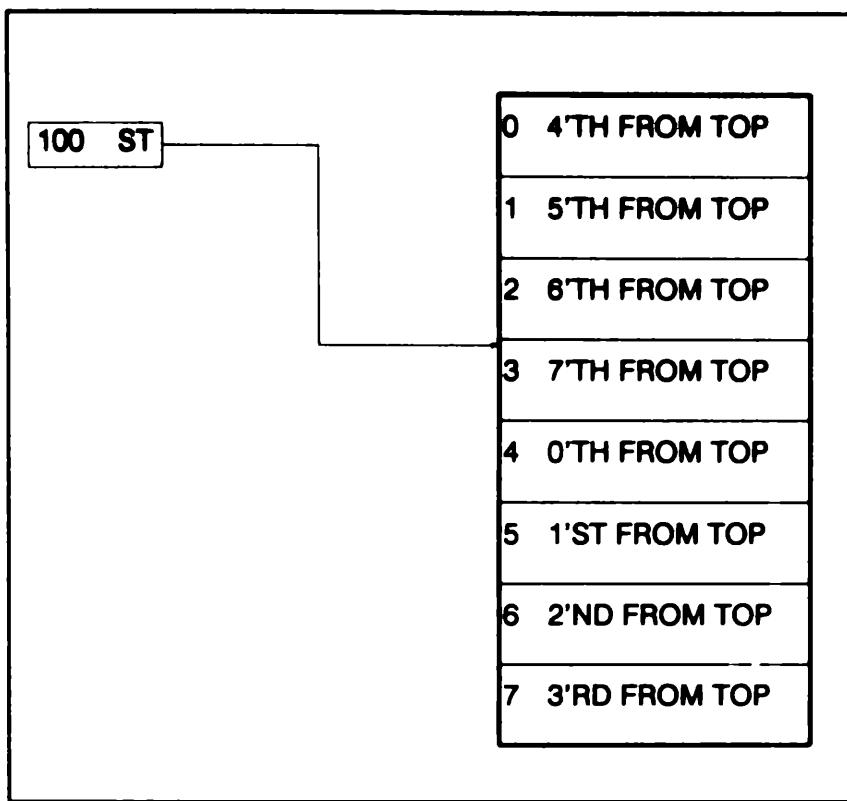


Fig. 4-1 The 80x87 stack, from 8087P

The algorithm has the following expression in pseudo-FORTH

Redefine operations that put #'s on 87stack:
 increment stack_pointer
 if 87stack full put st(7) on fstack
 push onto st(0)

Redefine operations that take #'s off 87stack:
 pop st(0)
 decrement stack_pointer
 if fstack not empty, put tofs onto st(7)

We begin by defining the data structure (the fstack proper) where we will stash and retrieve the numbers coming off the 87stack.

- We need to decide how deep the fstack will be, in 80-bit (TREAL) wide cells, and then **ALLOT** 10⁶ that number of bytes of storage.
- We need a word that will initialize the 80x87 and fstack.
- We need a fast way to increment (or decrement) the address of the next available space in the fstack by 10.
- We need to test whether the 87stack is full (or empty).
- Finally, what do we do if the memory we set aside gets full? The solution chosen below makes the extension circular using modulo arithmetic to compute the addresses within it.

We now exhibit the fstack manager program in Fig. 4-2 on page 83 below. By now the reader should be familiar enough with FORTH style to understand the program logic with only moderate commenting and explanation. Remember — the program reads from the bottom up!

The key words are **FPUSH** and **FPOP** — they do the work. Every word that changes the 87stack has to be redefined to include either **FPUSH** or **FPOP** as appropriate.

As the test results in Table 4-5 below make clear, the high-level stack extension is too slow. Some optimization is necessary.

Table 4-5 *fstack manager timings*^{†‡}

[†]8086 machine @ 4.77 MHz

[‡]40,000 FPUSH's and FPOP's

High-level	Optimized	Hand-coded
29 sec	6 sec	4 sec

HS/FORTH comes with a very helpful utility: a recursive-descent optimizer. The optimizer replaces the subroutine calls of ordinary high-level threaded FORTH code by in-line machine code.

```

\ FLOATING POINT STACK MANAGER
DECIMAL
VARIABLE FS-SIZE           ( size of fstack in TREAL's)
40 FS-SIZE                 ( the fstack is 40 deep)
VARIABLE FLGTH              ( length of fstack in bytes)
VARIABLE FSP                  ( current offset into fstack)
VARIABLE FDEPTH             ( current size of stack)
                           ( -8 fdepth f-length)

CREATE FSTACK FS-SIZE @ 10 * ALLOT OKLW
:FSINIT FSP 0! -8 FDEPTH ! FS-SIZE @ 10 *
    FLGTH ! FINIT ;          ( initialize 8087 and stacks)
CODE 10+      < % B3 C3 0A % >      END-CODE
CODE 10-      < % B3 EB 0A % >      END-CODE
                           ( fast way to add or subtract 10)
FSINIT
:WRAP( fsp -- fsp mod flgth ) \make fstack circular
    [FLGTH @] LITERAL UNDER + SWAP MOD ;

:AWAY! DUP ROT ! ; ( adr n -- n) \useful factored word
:INC-FSP ( --fsp' ) FSP DUP@ 10+ WRAP AWAY! ;
:DEC-FSP ( --fsp' ) FSP DUP@ 10- WRAP SWAP ! ;

:INC-FDEPTH ( --fdepth )
    FDEPTH DUP@ 1+ FS-LENGTH @ MIN
    AWAY! ;

:DEC-FDEPTH           ( --fdepth )
    FDEPTH DUP@ 1- DUP -8 <
    ABORT" FSTACK UNDERFLOW" AWAY! ;

:F PUSH INC-FDEPTH 0 < NOT
    IF INC-FSP FSTACK +
        FDECSTP R80! ( st[7] -- fsp )
    THEN ;

:F POP DEC-FDEPTH -1 < NOT
    IF FSP @ FSTACK +
        R80@ FINCSTP ( fsp -- st[7] )
        DEC-FSP THEN ;

```

Fig. 4-2 Hi-level fstack manager for 80x87

stripping out redundant pushes and pops of the parameter stack. It is fairly straightforward to construct a similar optimizer for any FORTH dialect. Alternatively, one can machine code the time-critical words.

The results of the tests, presented in Table 4-5 on page 82 above, are interesting:

- the optimizer does nearly as well as hand-tuned machine code.
- An FPUSH or an FPOP takes about 50 μ sec on the average, when coded by hand, @ 4.77 MHz, or about 240 clock cycles.
- The irreducible minimum on an 8086/80286 — since TREAL storage from, or retrieval to the 87stack is demanded — cannot possibly be less than 170 clock cycles: 1 fetch and 1 store¹⁹. (The main place to save some time would be in moves to or from memory; the depth of the fstack and the pointer must be kept as variables, but FS-SIZE and FLGTH do not change and could be compiled as literals, thereby saving 30 cycles in FPUSH and 10 in FPOP.)
- Substantially greater efficiency (at best another 1.5x improvement over the code version discussed above) would require a different algorithm — based, perhaps, on the 87stack overflow or underflow interrupt. But because other error conditions can initiate this interrupt, the testing and decision-making needed to use this method seemed to me likely to produce equivalent overhead to the method employed here.

§8 Clone wars

Several companies have produced non-infringing clones of the Intel 80x87 family of chips. American Micro Devices is a second source for 80287 and 80387 chips. Cyrix Corporation has produced 80287 and 80387 equivalents with significantly faster transcendental functions and moderately faster arithmetic than the Intel originals.

And finally, Integrated Information Technology, Inc. (founded by the designers of the Weitek chips) has produced the most inter-

19. Somewhat less, \approx 100 clocks, if a 32-bit bus is utilized with the '386/'387 pair. See, e.g., John H. Crawford and Patrick P. Gelsinger, *Programming the 80386* (Sybex, San Francisco, 1987).

esting of the clones: the 80c287/80c387 not only perform arithmetic significantly faster than the Intel originals, they possess 24 additional 80-bit on-chip registers. Unfortunately these cannot be combined directly with the eight original registers to make a 32-deep stack, since this would have required increasing the 80x87 stack-pointer (see §7 above) from 3 bits to 5. Since there was no place to find the 2 extra bits, one cannot really fault IIT's designers.

But if they cannot be used to extend the 87stack, what are the 24 extra registers good for? Eight (bank 3) are not even accessible, being used to speed up on-chip arithmetic. However, banks 0–2 – 24 registers – can be accessed (e.g. for on-chip cache memory). Moreover, IIT has provided an on-chip linear transformation: a 4×4 matrix multiplies a 4-dimensional column vector in place. (The original vector is overwritten, but the matrix is unchanged.) It works like this²⁰:

First we define the instructions

```
CODE FSBP0  <% DB E8 %>  END-CODE
CODE FSBP1  <% DB EB %>  END-CODE
CODE FSBP2  <% DB EA %>  END-CODE
CODE F4x4   <% DB F1 %>  END-CODE
```

Then we load the 4×4 matrix into banks 1 and 2, the vector into bank 0, and multiply:

```
0 VAR a{{ 0 VAR v{
: VEC->c87 (adr --)      \ load vector into 80c387
    IS v{                  \ ! vector address to v{
        FINIT FSBP0          \ reset ST, select bank 0
        3 0 DO v{ 1 } G@    LOOP ;
: MAT->c87 (adr --)      \ load matrix into 80c387
    IS a{{                  \ ! matrix address to a{{ 
        FINIT FSBP2          \ reset ST, select bank 2
        1 0 DO
            3 0 DO a{{ 1 J }} G@  LOOP
        LOOP                  \ cont'd ...
```

20. Note: FINIT operates differently on IIT's coprocessors than on Intel's. It does not place NAN's in all 8 87stack cells.

```

FINIT FSBP1           \ reset ST, select bank 1
3 2 DO
  3 0 DO a{{ I J }} G@ LOOP
  LOOP ;

: c87->VEC ( adr -- )      \ ! from 80c387 to vector
IS v{                         \ ! vector address to v{
FSBP0                         \ select bank 0
  3 0 DO v{ I } GI LOOP ;
\ example: ANS = A*V
A{{ MAT->87 V{ VEC->87 F4x4 ANS{ c87->VEC

```

The on-chip 4×4 matrix multiply was intended to accelerate 3-dimensional graphics (rotation and translation can be expressed as a single 4-dimensional transformation). However, most scientific programmers spend little time on 3-D graphics. The matrix instructions are more interesting for their potential to accelerate general matrix operations²¹. For example, suppose we need to transform a vector by multiplying with an arbitrary matrix. Normally we would write

$$y_i = \sum_{j=1}^n A_{ij} x_j \quad (7)$$

But consider, *e.g.*, an 8×8 matrix operating on an 8-dimensional column vector: we **partition** the matrix and vector into 4-dimensional sub-matrices a_{11} , *etc.* and sub-vectors x_1 , *etc.*:

$$\begin{bmatrix} A \end{bmatrix} [x] \equiv \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 \\ a_{21}x_1 + a_{22}x_2 \end{bmatrix} \quad (8)$$

From timings on assembly-coded demonstration programs that multiply vectors by constant 4×4 matrices, we estimate an overall speedup of 6–7-fold on an 80c387 system. The timings for this process are as shown in Table 4-6 below (assume 32-bit REAL*4 matrices). The execution time for a 4-dimensional linear transformation using conventional operations is approximately 1744 clock cycles. The time using the vector operation is some 250–300 clocks, based on measured performance. Since 256 clocks are

21. See Ch. 9 of this book for a fuller discussion of standard matrix algorithms.

needed to load and store a 4-dimensional vector, we therefore estimate the vector operation **F4x4** takes only about 50 clocks, i.e. about 1 floating point multiplication time. We can now estimate the time to multiply two 4×4 matrices as about 1200 clocks vs. about 7000 for the scalar process, i.e. the same speedup factor

Table 4-6 Timings for 4×4 matrix · vector

<u>Operation:</u>	\times	$+$	$@$!
<u>No. x Clocks:</u>	16*52	12*28	20*20	4*44
<u>Total clocks:</u>	832	336	400	176

as for *matrix · vector*. If one must load the 4×4 matrix each time, the speedup factor is less: about 3.5-fold because of the 16 additional fetches.

The conventionally programmed 8×8 matrix-vector multiply should also be some 3-5 times faster than the scalar operations, i.e. there is no obvious speed gain – except being able to employ the built-in vector instruction **F4x4** – from partitioning the 8×8 system into 4×4 sub-units. However, Strassen²² has pointed out that if one can evaluate matrix products recursively, partitioning can substantially speed the most time-consuming matrix operations, multiplication and inversion. For example, it appears as though the product of two partitioned matrices,

$$\begin{bmatrix} A \\ A \end{bmatrix} \begin{bmatrix} B \\ B \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \equiv [C] \quad (9)$$

$$[C] = \begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \end{bmatrix}$$

22. V. Strassen, *Numer. Math.* 13 (1969) 184. See also V. Pan, *SIAM Review* 26 (1984) 393.

requires 8 matrix multiplications and 4 matrix additions to evaluate. Strassen has shown that in fact the evaluation can be performed with 7 matrix multiplications:

$$\begin{aligned}
 p_1 &= (a_{11} + a_{22}) (b_{11} + b_{22}) \\
 p_2 &= (a_{21} + a_{22}) b_{11} \\
 p_3 &= a_{11} (b_{12} - b_{22}) \\
 p_4 &= (-a_{11} + a_{21}) (b_{11} + b_{12}) \\
 p_5 &= (a_{11} + a_{12}) b_{22} \\
 p_6 &= a_{22} (-b_{11} + b_{21}) \\
 p_7 &= (a_{12} - a_{22}) (b_{21} + b_{22})
 \end{aligned} \tag{10}$$

and 18 matrix additions:

$$\begin{aligned}
 c_{11} &= p_1 - p_5 + p_6 + p_7 \\
 c_{12} &= p_3 + p_5 \\
 c_{21} &= p_2 + p_6 \\
 c_{22} &= p_1 - p_2 + p_3 + p_4
 \end{aligned} \tag{11}$$

Equations 10 and 11 look at first blush half as efficient as 8 multiplications and 4 additions. But let us examine the time to multiply two partitioned matrices, first by the straightforward method and then by Strassen's: clearly,

$$M_{2n} = 8M_n + 4A_n \tag{12}$$

where M_n is the multiplication time and A_n the addition time, for square matrices of order n .

Setting $n = 2^k$ that (note $A_n \approx O(n^2)$) we see the recursion, Eq. 12, is satisfied by an expression of form

$$M_n = m \lambda^k + ca 4^k \tag{13}$$

where m and a are the elementary multiplication and addition times. Substituting 13 in 12 we find $\lambda = 8$ and $c = -1$, i.e.,

$$M_n = mn^3 - an^2 \quad (14)$$

Applying the same idea to Strassen's method we obtain

$$\hat{M}_{2n} = 7\hat{M}_n + 18an^2 \quad (15)$$

or

$$\hat{M}_n = mn \lg 7 - 6an^2, \quad (16)$$

where

$$\lg 7 \equiv \log_2 7 = 2.807\dots$$

That is, partitioning allows a potentially large reduction in the time to multiply dense matrices.

By writing a partitioned matrix in the form

$$[A] = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ a_{21}a_{11}^{-1} & 1 \end{bmatrix} \begin{bmatrix} a_{11} & a_{12} \\ 0 & z \end{bmatrix} \quad (17)$$

where

$$z = a_{22} - a_{21}a_{11}^{-1}a_{12} \quad (18)$$

we may express the inverse of A as

$$[A]^{-1} \equiv \begin{bmatrix} a_{11}^{-1} & a_{11}^{-1}a_{12}z^{-1} \\ 0 & z^{-1} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -a_{21}a_{11}^{-1} & 1 \end{bmatrix} \quad (19)$$

which leads to the recursion for I_n , the time to invert an $n \times n$ matrix:

$$I_{2n} \approx 2I_n + 5M_n \quad (20)$$

whose solution is

$$I_n = mn \lg 7 + O(n) \quad (21)$$

i.e., the time needed to invert is comparable with that needed to multiply.

Suppose we merely wish to solve a linear system *without* inverting the matrix: can we gain some speed that way? From 17 we see that the problem

$$Ax = y$$

reduces to three sub-problems:

$$a_{11}u_1 = y_1 \quad (22a)$$

$$z x_2 = y_2 - a_{21}u_1 \quad (22b)$$

$$a_{12}x_2 = y_1 - a_{11}u_1 ; \quad (22c)$$

that is, we have the recursion

$$S_{2n} = 3S_n + mn^{\lg 7} + 2mn^2 \quad (23)$$

whose solution is dominated by

$$S_n = m \left(\frac{1}{4} n^{\lg 7} + 2n^2 \right) + O(n^{\lg 3}) \quad (24)$$

Linear equation solution *via* recursion thus has the same asymptotic running time as matrix multiplication, except that $4 \times$ fewer operations are required than for inversion. That is, it should be about $5 \times$ faster to solve a dense system of 1000 linear equations by recursive partitioning than by ordinary Gaussian elimination even on a scalar processor. The vector instruction on the IIT 80c387 chip, together with Strassen's algorithm, offers the possibility of solving very large systems in practical times, on desktop computers.

Scientific Data Structures

Contents

\$1 Typed data structures	92
 \$S1 Type descriptors	94
 \$S2 Typed scalars	94
 \$S3 Defining several scalars at once	95
 \$S4 Generic access	97
 \$S5 The intelligent floating point stack (ifstack)	99
 \$S6 Unary and binary generic operators	100
\$2 Arrays of typed data	104
 \$S1 Improved (FORTRAN-like) array notation	105
 \$S2 Large matrices	106
 \$S3 Using high memory	108
 \$S4 A general typed-array definition	110
 \$S5 2ARRAY and {}	113
\$3 Tuning for speed	114

Data structures are the soul of any computer program in any language. Some languages, most notably FORTRAN and BASIC, predefine some data structures but require extensive contortions to define others. This straitjacket approach has virtues as well as defects:

- The pre-defined structures are what most users need to solve standard problems, so meet 80-90% of the cases in practice. That is, they are not terribly restrictive.
- Because the most-needed structures are predefined and have a standard format, they do not have to be invented each time a program is written. Standardization facilitates the exchange and portability of programs.
- Standardized data structures aid program development in discrete modules, permitting sections written by different persons or teams to interface properly with minimal tuning.

FORTH pre-defines a minimal set of data structures but allows unlimited definition of new structures. How is this different from

Pascal, Ada or even C? FORTH not only permits extension of the set of data structures, it permits definition of new operators on them. Thus, e.g., FORTH permits simple implementation of complex arithmetic whereas the aforementioned do not.

This chapter¹ suggests protocols for arrays and typed data² that will increase the portability of code and encourage the exchange of scientific programs. The keys to this are generic operations that recognize the data type of a scalar or array variable at run-time and act appropriately.

§1 Typed data structures

One of the virtues of FORTRAN or BASIC is that the programmer does not have to keep track of what type of data he is fetching and storing from memory. In fact, the user does not even program such operations explicitly — the compiler takes care of everything including the bookkeeping. Mixed-arithmetic expressions like

$$Z = -37.2E-17 * CEXP(CMPLX(R^{**}2,W)/32) / DSIN(W)$$

place great demands on a compiler. The compiler first tabulates the types of the variables and literals in the expression, and then decide which run-time routines to insert. With two types of integers and four types of floating-point numbers (REAL*4, REAL*8, COMPLEX*8 and COMPLEX*16) a typical binary operator such as exponentiation (**) offers 36 possibilities. No wonder FORTRAN compilers are slow.

FORTH sacrifices automation, opting for a small, fast, flexible compiler. The traditional FORTH style gives each type of data its own operators. However, if a program demands all the standard REAL*4, REAL*8, COMPLEX*8 and COMPLEX*16 data types (not to mention INTEGER*2 and *4), having to remember them all and use them appropriately is a chore. This problem has

1. Much of the material in this chapter has appeared previously in J.V. Noble, *J. FORTH Ap. and Res.* 6 (1990) 47.
2. Most languages classify data by type: in FORTRAN, e.g., we have INTEGER, INTEGER*4, REAL, REAL*8, COMPLEX and COMPLEX*16, requiring 2, 4, 4, 8, 8 and 16 bytes of memory, respectively.

led me to experiment with generic access operators, **G@** and **G!**. These let FORTH keep track of which words to use in fetching and storing the "scientific" data types to the fstack (which may partly reside on a co-processor like the 80x87 or MC68881 chips). Corresponding generic unary and binary floating point operators **GDUP**, **G***, etc. allow programs themselves to be generic.

I have lately further modified the scheme to permit more complete automation. The kernel of the method is an "intelligent" fstack, or ifstack, that records the type of each number on it. The generic arithmetic operators and library functions decide from the information on the ifstack how to treat their operands.

An ifstack-based protocol for floating point and complex arithmetic has drawbacks and advantages. A major drawback is the run-time overhead in maintaining the ifstack, and in choosing the appropriate operator for a given situation. In other words we trade convenience for a non-negligible execution speed penalty. To some extent this can be mitigated by *computing* decisions and by vectoring rather than branching (*i.e.* no Eaker CASE statements or **IF ... ELSE ... THEN**s). Moreover, although the definitions are coded in high-level FORTH for portability, the key words should be hand-assembled for the target machine. Finally, my high-level ifstack manager has plenty of error checking that could be dispensed with when speed is an issue.

The chief advantages of the ifstack are:

- Unlike FORTRAN, this scheme permits generic routines that will accept several types of input. Hence, *e.g.*, a matrix inversion routine will happily invert **REAL*4**, **REAL*8**, **COMPLEX** and **DCOMPLEX** matrices.
- A **FORTRAN → FORTH** translator³ becomes simple with generic operators.
- The ifstack permits recursive programming *a la LISP*.

3. See J.V. Noble, *J. FORTH Ap. and Res.* 6 (1990) 131. See also Chapter 11, where we describe a simple FORTRAN TRANSLATOR.

§§1 Type descriptors

To decide at run-time which @ or ! to use for a particular datum, FORTH needs to know what type of datum it is. The scheme described here wastes a little memory by attaching to each variable a label that tells G@ and G! how to get hold of it.

Here is how we label types:

```
\ Data type identifiers
0 CONSTANT REAL*4          \ 4 bytes long
1 CONSTANT REAL*8          \ 8 bytes long
2 CONSTANT COMPLEX          \ 8 bytes long
3 CONSTANT DCOMPLEX         \ 16 bytes long

\ a simple version of #BYTES
CREATE #bytes 4 C, 8 C, 8 C, 16 C,
:#BYTES ( type - #bytes ) #bytes + C@ ;
```

§§2 Typed scalars

We want the machine to remember for us the data-specific fetches and stores to the co-processor. To accomplish this, the typed variable has to place its address and type on the stack. Thus we need a data structure that we might visualize diagrammatically in Fig. 5-1 below (a cell |■■| represents 2 bytes):

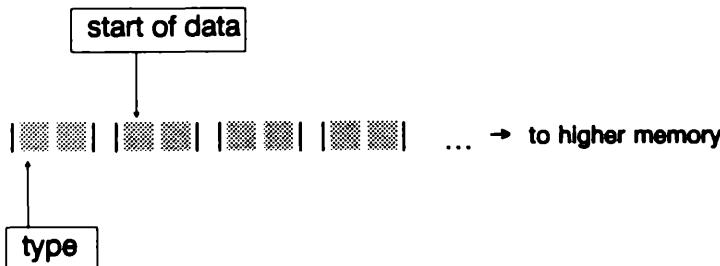


Fig. 5-1 Memory structure of a typed scalar

We implement a scalar through the defining word

```
: SCALAR ( type -- )
CREATE DUP , #BYTES ALLOT
DOES> DUP@ SWAP 2+ SWAP ; ( -- adr !)
```

The word **SCALAR** is used as

```
REAL*4 SCALAR X
REAL*8 SCALAR XX
COMPLEX SCALAR Z
DCOMPLEX SCALAR ZZ
... etc. ...
```

Defining several scalars at once

One aspect of the FORTH method of handling variables, that seems strange to programmers familiar with Pascal, BASIC or FORTRAN, is that **VARIABLE**, **CONSTANT** or a new defining word like **SCALAR** need to be repeated for each one defined, as above. That is, such defining words generally do not accept name-lists.

This idiosyncracy can be traced to FORTH's abhorrence of variables:

- Easily read (and maintained) FORTH code consists of short definitions with few (generally ≤ 4) numbers on the stack. Such programs have small use for variables, especially since the top of the return stack can serve as a local variable.
- In FORTH as in BASIC, variables tend to be global and hence corruptible. The variables in a large program can have unmnemonic names or names that do not express their meaning simply because we run out of names.
- Experienced FORTH programmers tend to reserve named variables for such special purposes as vectoring execution.
- The standard FORTH kernel therefore discourages named variables by making them as tedious as possible.

Most objections to variables can be resolved by making them local. Local variables are relatively easy to define in FORTH: a

straightforward but cumbersome method for making "header-less" words is given in Kelly's and Spies's book⁴.

HS/FORTH⁵ provides beheading in a particularly simple form: **BEHEAD' NAME**, or **BEHEAD" NAME1 NAME2**.

Used after **NAME** has been invoked in the words that need to reference it, **BEHEAD'** removes **NAME**'s dictionary entry leaving pointers and code fields intact and recovering the unused dictionary space. The more powerful word **BEHEAD"** does the same for the range of dictionary entries **NAME1 ... NAME2**, inclusive.

Beheading variable or constant names makes them local to the definitions that use them; they cannot be further accessed — or corrupted — by later definitions. (Pountain⁶ has given yet another method for making variables local, using a syntax derived from "object-oriented" languages such as SMALLTALK.)

Variables are essential for scientific programming. Since we must often have more than two variables, it is silly to repeat **SCALAR**. A simple way to allow **SCALAR** to use a list is

```
: SCALARS ( n -- )
    SWAP 0 DO DUP SCALAR LOOP DROP ;
\ Examples:
\ 2 REAL*4 SCALARS A B
\ 5 COMPLEX SCALARS XA XB XC XD XE
```

I find the use of **SCALARS** with modifiers and lists more convenient and readable than many repetitions of **SCALAR**. Its resemblance to FORTRAN (thereby helping me live with my FORTRAN-inspired habits) is pure coincidence. Although possible to use a terminator (", e.g.) rather than a count (to define the variable list) I feel it is desirable for the programmer to know

- 4. M.G. Kelly and N. Spies, *Forth, a Text and Reference* (Prentice-Hall, New Jersey, 1986), p. 324 ff.
- 5. CHarvard Softworks, P.O. Box 69, Springboro, Ohio 45066 Tel: (513) 748-0390.
- 6. Dick Pountain, "Object-oriented FORTH", Byte Magazine, 8/86; *Object-oriented FORTH* (Academic Press, Inc., Orlando, 1987).

how many variable names he has supplied, hence the counted version.

§§4 Generic access

A major theme of FORTH is to replace decisions by calculation whenever possible⁷. This philosophy usually pays dividends in execution speed and brevity of code.

But there is an even more important reason to avoid IF ... THEN decisions, especially when working with modern microprocessors. CPUs like 80x86 and MC680x0 achieve their speed in part by pre-fetching instructions and storing them in a queue in high speed on-chip cache memory. A conditional-branch machine instruction (the crux of IF ... THEN) empties the queue whenever the branch is taken. Branches should be avoided because they slow execution far more than one might expect based on their clock-counts alone.

To replace decisions, we use the standard FORTH technique of the execution array (analogous to the familiar assembly language jump table). This lets us compute from the type descriptor which fetch or store to use.

We now define G@ and GI as execution arrays using⁸ an execution-array-defining word G:

```
: G: CREATE ]
    DOES> OVER ++ @ EXECUTE ; (t--)
G: G@ R32@ R64@ X@ DX@ ;
G: GI R32! R64! XI DXI ;
```

assembled from components of the FORTH compiler. That is, the ordinary colon : might have the high-level definition (shorn of error detection)

7. Leo Brodie, *Thinking FORTH* (Prentice-Hall, Inc., Englewood Cliffs, N.J., 1984), p. 118ff. See also J.V. Noble, "Avoid Decisions", *Computers in Physics* 5,4 (1991) 386.
8. HS/FORTH uses a word pair CASE: ... ;CASE that performs the same task as G: ... ; below. G: was inspired by Michael Ham (*Dr. Dobb's Journal*, October 1986).

```
: : CREATE ] DOES> @ EXECUTE ;
```

CREATE makes the new dictionary entry, and **]** switches to compile mode. **DOES>** specifies the run-time action (recall any word created by **CREATE** leaves its parameter field address **-pfa-** on the stack at run-time, *prior* to the actions following **DOES>**). In the case of **:** the run-time action is to fetch the pfa of the new word and execute it. At run-time, words defined using **G:** add twice the type descriptor to the pfa (to get the offset into the array) then fetch the desired address and **EXECUTE** it.

Microprocessors like the MC680x0 and 80386 that can address large, level memories require no further elaboration for **G@** and **G!**. However, if large arrays are to be addressed within the segmented memory addressing protocol of the 8086/80286 chips, we would have to define **G@** and **G!** to use the “far” forms of addressing words⁹. For example, in HS/FORTH such words as **R32@L** expect a segment paragraph number and offset (32 bits total) as the complete address of the variable being fetched to the 87stack. In that case we modify the definition of **SCALAR** to include the segment paragraph number (seg) in the definition (**LISTS** is nonstandard – it is HS/FORTH’s name for the portion of the dictionary containing the word headers)

```
: SCALAR ( type -- )
  CREATE DUP ,           \ make header , type
  #BYTES ALLOT          \ reserve space
  DOES> >R
  [ LISTS @ ] LITERAL   ( -- seg)
  R@ 2+
  R> @ ;
\ Ex: REAL*4 SCALAR X
```

9. Consult, e.g., L.J. Scanlon, *op. cit.*; or R. Lafore, *op. cit.* HS/FORTH defines “far” access operators, **@L** and **IL** of all types, that expect a “long” address on the stack. For example, **CODE R32@L DS POP. FWAIT. DS: [BX] DWORD-PTR. FLD. END-CODE**

§§5 The Intelligent floating point stack (ifstack)

The ifstack is a more complex data structure than either a simple fstack or the parameter/return stacks. When a typed datum is placed on the ifstack its type must be placed there also.

But the typed data have varying lengths, from 4 to 16 bytes. We can deal with this two different ways: either ALLOT enough memory to hold a stack of the longest type, making each position on the ifstack 18 bytes wide (to hold datum plus type); or manage the ifstack as a modified heap, with the address of a given datum being computable from the ifstack-pointer and the data type.

The 18-byte wide ifstack wastes memory, but is easy to program. (In retrospect, this is exactly the method I used to program adaptive numerical quadrature¹⁰.) After several false attempts I settled on the fixed-width ifstack. High level FORTH code for this variant is given below.

```
\ TYPED DATA STACK MANAGER
TASK FSTACKS
FIND CP@ 0=
?( FLOAT COMPLEX.FTH )

\ define data-type tokens
0 CONSTANT REAL*4
1 CONSTANT REAL*8
2 CONSTANT COMPLEX
3 CONSTANT DCOMPLEX

CREATE #bytes 4 C, 8 C, 8 C, 16 C,
:#BYTES #bytes + C@ :
(type -- length in bytes)

\ define scalar and scalars
: SCALAR      ( type-- )
  CREATE DUP , #BYTES ALLOT
  DOES> DUP@ SWAP 2+ SWAP ;

          ( -- seg off type)
\ say: REAL*4 SCALAR X
: SCALARS           ( n type -- )
  SWAP 0 DO DUP SCALAR LOOP
  DROP ;
\ say: 4 DCOMPLEX SCALARS XA XB XC XD

\ definitions for the parallel stack of types and data
\ Brodie, TF (Brady, NY, 1984) p. 207.

CREATE FSTACK 20 18 * 2+ ALLOT
\ 2 tos-pointer, 20 18-byte cells
: FS.INIT FSTACK 0! ;
: >EMPTY      ( -- seg off )
  [ LISTS @ ] LITERAL
  FSTACK DUP@ 18 * 2+ + ;
: >FS   ( seg off type-- ) \ say: X >FS
  >R >EMPTY      ( -- seg off seg' off )
  R@ OVER !       \ store type on ifstack
```

10. J.V. Noble, "Scientific Computation in FORTH", *Computers in Physics* 3 (1989) 31; and Ch. 8§1§§5 of this book.

```

FS>  ( seg off type -- ) \ say: X  FS>
FSTACK 1-!  \ dec lstack ptr
>R  >EMPTY ( -- seg off seg' off )
DUP@ R@ = \ srce.type = dest.type ?
IF 2+ DSWAP ( -- seg' off' seg off )
R > #BYTES ( -- seg' off' seg off n)
CMOVEL \ move data from lstack
ELSE RDROP CR
." ATTEMPT TO STORE TO
WRONG DATA TYPE" ABORT
THEN ;

\ execution-array defining word
\ HS/FORTH has the faster
\ CASE: ... ;CASE pair for the same job

: G: CREATE ] DOES> OVER + +

```

```

@ EXECUTE ; ( t-- )

G: G@ R32@L R64@L X@L DX@L ;
G: G! R32!L R64!L X!L DX!L ;
\ move data from lstack to/from FPU
: FS>F ( -- t 87: --x )
FSTACK 1-! \ dec lstack ptr
>EMPTY ( -- seg off )
DUP@ >R 2+
R@ ( -- seg off type)
G@ R> ;
\ move data from lstack to 87stack, leave type

: F>FS ( t-- 87: x-- )
>R >EMPTY ( -- seg off )
R@ OVER !
2+ R> ( -- seg off type)

```

The stack comments and comments should make the preceding code self-explanatory.

§§6 Unary and binary generic operators

We want to define generic unary and binary operators whose run-time action selects the desired operation using information contained in the ifstack. A unary operator such as **FNEGATE** or **FEXP** expects one argument and leaves one result. With a floating-point coprocessor (FPU) the only distinction is between real or complex. This distinction is contained in the second bit of the type descriptor, which we exhibit in Table 5-1 on page 101, in binary notation.

Real and complex can then be distinguished via the code fragment

2 AND (type -- 0 = real | 2 = complex)

Table 5-1 Bit-patterns of data type descriptors

Type	BINARY representation
REAL*4	00000000 00000000
REAL*8	00000000 00000001
COMPLEX	00000000 00000010
DCOMPLEX	00000000 00000011

Since most unary operators produce results of the same type as their argument, we write a defining word for *generic* unary operators:

```
: GU: CREATE ] DOES>      ( -- pfa)
FS>F ( -- pfa t)          \ get data
UNDER 2 AND +             ( -- t adr )
@ EXECUTE                 \ do it
F>FS ;                   \ return ans.
```

When we use **GU:** in the form

```
GU: GNEGATE FNEGATE XNEGATE ;
```

CREATE produces a dictionary entry for **GNEGATE**; **]** turns on the compiler so the previously defined words **FNEGATE** and **XNEGATE** have their addresses compiled into **GNEGATE**'s parameter field; and **DOES>** attaches the run-time code. The run-time code converts the real/complex bit into an offset, 0 or 2 which is added to the address of the daughter word to get the address where the pointer to the actual code is stored. This pointer is fetched and **EXECUTEd**.

A few unary operators like **XABS** (complex absolute value) return real values from complex arguments. If we want to use **GU:** to define, say, **GABS**, we must remember to redefine **XABS** so it zeros the second bit of the type descriptor left on

the stack, before returning its result to the ifstack. This is just a **1 AND** so is fast.

A binary operator (one that takes two arguments) expects its arguments *and* their types on the ifstack. There is no distinction between single- and double-precision arithmetic on most numeric coprocessors. However, the result must leave the proper type-label on the stack. Here is what we want to happen, illustrated in Fig. 5-2 as a matrix **TYPE(arga, argb)**

TYPE_{ab}		<i>a</i>	<i>b</i>	R	D	X	DX
R	R			R	R	X	X
D	D			R	D	X	DX
X	X			X	X	X	X
DX	DX			DX	X	X	DX

Fig. 5-2 Types resulting from 2-argument operators

Note: this protocol avoids misleading precision for the results of computations. It seems more scientific than FORTRAN's "convert intermediate results to the precision of the highest-precision operand" protocol.

If we think of the indices and entries in Fig. 5-2 as numbers 0, 1, 2, 3 (so we can use them as indices into a table) rather than as letters, a simple algorithm emerges: the first bit of the result is the logical-AND of the first bits of the two operands, and the second bit of the result is the logical-OR of their second bits. Although we would program this in assembler for speed, the high-level definition is

```

: NEW.TYPE { a b -- a2 + b2 + a1b1)
  DDUP { -- a b a b)
  AND { -- a b ab)
  1 AND { -- a b [ab]1)
  -ROT OR { -- [ab]1 a+b)
  2 AND { -- [ab]1 [a+b]2)
  + ; { -- a2 + b2 + a1b1)

```

Since only logical operations are used, **NEW.TYPE** is faster than table lookup or branching. Note that in programming this key word we have obeyed the central FORTH precept: "Keep it simple!" by choosing a data structure (the numeric type tokens 0-3) that is easily manipulated.

We will also need a way to select the appropriate operator from a jump table of addresses. Given that the precision (internal) is irrelevant, again all that matters is whether the number is real or complex, i.e. the second bits of the numbers. The first operation must then be to divide by 2 (right-shift by one bit). We then have the matrix of Fig. 5-3 below

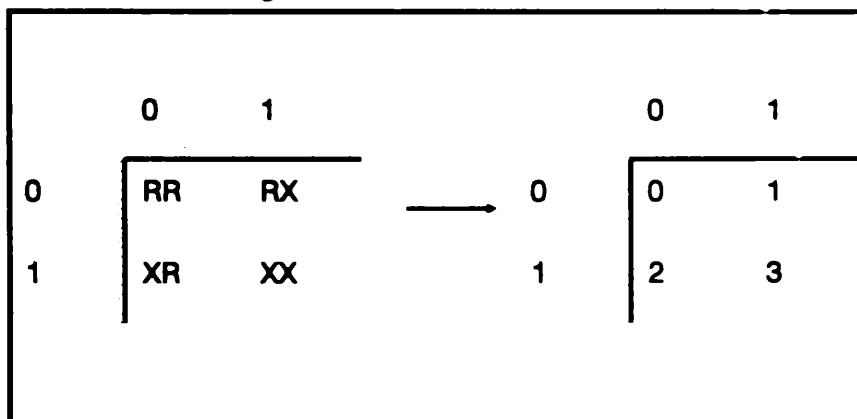


Fig. 5-3 Operator selection matrix

where RR stands for real-real, etc. The numerical elements are generated as $2^J \cdot I$. This leads to the word

```

: WHICH.OP { a b -- c)
  2/ SWAP 2 AND +

```

Thus we come to the binary generic-operator defining word

```
: GB: CREATE ] DOES>      ( -- pfa)
    FS>F FS>F      ( -- pfa t0 t1)
    NEW.TYPE UNDER   ( -- t' pfa t')
    WHICH.OP 2* +     \ make result-type
    @ EXECUTE        \ select binop
    F>FS ;           \ save result
\say: GB: G*      F*  F*X  X*F  X* ;
```

The generic multiply **G***, e.g., picks out, at run-time, which of four routines to use. By using only logical or shift operations we have made even the high-level definitions fairly quick in comparison with the times of floating point operations.

The only instance where one might forego the overhead penalty paid for the convenience of generic coding would be in nested inner loops, such as occur in matrix operations. Here it might pay to code four inner loops, one for each type, and then access them generically, e.g.

```
: RLOOP      ... real words ...
: DRLOOP     ... dreal words ...
: XLOOP      ... complex words ...
: DXLOOP     ... dcomplex words ...
G: GLOOP RLOOP DRLOOP XLOOP DXLOOP ;
```

§2 Arrays of typed data

Numerical arrays represent a frequently encountered characteristic feature of scientific programming. Arrays *per se* are hardly foreign to FORTH. Arrays of typed data are novel, however, and therefore worth elaborating. Following Brodie's advice(TF, p. 48ff) we first specify the "user interface" (matrix notation) and then proceed to implementation.

§§1 Improved (FORTRAN-like) array notation

Something like $V(15)$ — the 15'th element of V — is the commonest notation for array elements in high-level languages because lineprinters and terminals do not easily recognize subscripts. In FORTH, the most natural notation would be postfix (RPN), $15\ V$ — but this is both hard to read and unintuitive¹¹. That is, $15\ V$ does not say, immediately and unambiguously, “I am the 15th element of the array V !”

FORTH's idiosyncrasies forbid saying $V(15)$ because the parser recognizes $V(15)$ as a single word¹². Since we want the 15 to be parsed, we would have to modify the FORTRAN-ish notation to $V\cdot(•15•)$ or $V(•15•)$, where \cdot stands for a blank space (ASCII 32). Unfortunately, “(” is a reserved word. While we might place the matrix definitions in a separate vocabulary — which would let us redefine anything we want — “(” is too useful as a comment delineator to dispense with.

This leaves the second possibility, where “(” becomes part of the array name, $V(.$. To make $V(•15•)$ work, “)” must become an operator — unless we want to leave postfix notation entirely, with all the complication *that* would entail¹³. Since “)” is not a reserved word, nothing in principle prevents defining it as an operator. However, such usage would conflict with comments.

The square braces, [], are commonly used in matrix notation; however both are reserved FORTH words, *i.e.* forbidden. This leaves the curly braces { }, which are unused by FORTH.

Of the two possible forms, $V\cdot\{•15•\}$ or $V\{\cdot15•\}$, the latter has the advantage that the opening brace, {, is only part of the name, but reminds us that the name $V\{$ is an array, exactly as names ending with \$ are strings, *etc.* The notation suggests a further

1. Other authors have noted this and proposed more readable matrix notations. See, *e.g.*, Joe Barnhart, “FORTH and the Fast Fourier Transform” *Dr. Dobb’s Journal*, September, 1984, p. 34. Also Dick Pountain’s book *Object-oriented FORTH* (*op cit.*) uses an array naming convention with brackets.
2. Recall that the standard FORTH delimiter is the ASCII blank (20h = 32d).
3. See, *e.g.*, L. Brodie, *Thinking FORTH* (*op. cit.*) p. 113ff.

mnemonic refinement, namely to place $\{\{$ and $\}\}$ at the ends of 2-dimensional arrays, as in $M\{\{ \cdot 3 \cdot 5 \cdot \}\}$.

How will this notation operate? Clearly, to place the (generalized) address of the n 'th element (of a 1-dimensional array) on the stack we would say

$V\{\cdot n \cdot\}$,

whereas

$M\{\{ \cdot m \cdot n \cdot \}\}$

should analogously place the address of the m,n 'th element of a 2-dimensional array on the stack.

§§2 Large matrices

The defining word **SCALAR** given in §2 above allots space in the dictionary —for most FORTHS, code + data must fit here— or in the LISTS segment of HS/FORTH (part of the dictionary). This is OK for variables, but not for arrays, since even a modest matrix would exhaust the (≤ 64 Kbyte) LISTS segment.

A **REAL*4** matrix uses 4 bytes per element. The largest such array that can be stored in a 65,536 (i.e., 2^{16}) -byte segment is 128×128 . This is the largest array that can be addressed with unsigned 16-bit numbers. On the other hand, a filled IBM PC/XT clone has 640 Kbytes of memory under MS-DOS. Even a generous FORTH kernel (plus DOS) takes up less than 150 K; hence 450 K available to hold large arrays. Up to 8 Mbytes can be added via EMS storage, assuming a suitable memory management scheme¹⁴. That is, in principle one could tackle matrix problems of order 350×350 . What about speed? The dominant term in solving linear equations by —say— Gaussian elimination with partial pivoting is

14. See, e.g., Ray Duncan, "FORTH support for Intel/Lotus expanded memory", *Dr. Dobb's Journal*, August 1986; also, John A. Leffor and Karen Lund, "Reaching into expanded memory", *PC Tech Journal*, May 1987.

$$T = \frac{1}{3}mn^3$$

where m is the time for 1 multiply and 1 add, and n is the order of the matrix. We should also include the fetch + store time, since the bus bandwidth is as much a limiting factor as the FPU arithmetic speed. For the 8086/8087 the time m is of order 400 clock cycles. Thus the asymptotic execution time on a 10 MHz machine should be of order 10 minutes for $n = 350$.

On the 80386/80387 combination running at 25 MHz, the execution time for the same problem should be only 2.3 minutes or so. Thus it would be practical (*i.e.*, execution time \approx 1 hour) on such machines, even without special equipment such as the IIT 80c387, or an array co-processor, or a faster procedure such as Strassen's algorithm (see Ch. 4 §8), to tackle $10^3 \times 10^3$ dense matrix problems.

The crucial question therefore, is memory. The Intel machines were designed around a segmented memory architecture. That is, to avoid having to use (expensive) 32-bit address registers, the 8086/80286 chips were designed to use 16-bit registers. However, these chips have more than 16 external address lines — 20 for the 8086, 24 for the 80286. Thus the absolute address is compounded of two numbers: a **segment descriptor** and an **Offset**, which must be present in appropriate registers. The segment descriptor is the **absolute address** of a 16-byte **paragraph**, divided by 16. The offset is any (unsigned) integer from 0 to $2^{16} - 1 = 65,535$ that can fit in a 16-bit register.

The chips contain 4 segment registers: SS (stack segment), CS (code segment), DS (data segment) and ES (extra segment). Five registers, BP, SP, SI, DI and BX, can be used for offsets, although they are not entirely interchangeable (some have specific functions in some of the more complex machine instructions, such as string operations). Manifestly, since the 8086 can address

$$2^{20} = 1,048,576 \text{ bytes ("1 megabyte")},$$

the largest segment number is $2^{14} - 1 = 16,383$.

A typical (segmented) address is expressed in Intel assembly code as

CS: [BX + SI + 0008]

which translates in words to "add the offset in BX to that in SI and then add 8 to get the total offset; take the segment descriptor in CS, multiply by 16 and add to produce the absolute address.

§§3 Using high memory

HS/FORTH permits accessing all the memory in a PC/AT (up to 1 megabyte) in the following manner:

- Define a named segment of length 1 byte: this marks the beginning of available memory.
- Then tell both FORTH and DOS how much memory you want.

As might be expected, HS/FORTH defines non-standard words (coded as DOS function calls) to use the various DOS service routines that allocate memory, etc.¹⁵

```

MEMORY 4+ @ S->D
DCONSTANT MEM.START \ beg. of free memory
40.960 DCONSTANT MAX.PARS
\ 40960 = 655360 /16
: TOTAL.PARS MAX.PARS MEM.START D- ;
\ # pars of memory available
1 SEGMENT SUPERSEG
\ define named segment 1 byte long
TOTAL.PARS DROP FREE-SIZE
\ tell DOS and HS/FORTH about it

```

Having allocated the memory, how can we address it efficiently? We would like the simplicity of double-length integer arithmetic for computing an (absolute) array address, as in

$$\text{abs.adr}(A_{ij}) = \text{abs.adr}(A_{00}) + (\text{row.length} * i + j) * \# \text{BYTES}$$

However, although the absolute address referenced by a segment and offset is unique, i.e. the absolute address in bytes is

15. See, e.g., D.N. Jump, *Programmer's guide to MS-DOS*, rev. ed. (Brady Books, New York, 1987).

$$\text{abs.adr} = 16 \cdot \text{segment} + \text{offset},$$

the reverse translation, of an absolute address (in bytes) to the segment + offset notation expected by 80x86 processors is *not* unique. This naturally poses a problem when the processor tries to prevent segments from overlapping (protected mode). In such cases, the only answer is a memory management scheme that computes segments and offsets (by brute force) in a non-overlapping fashion. For example, we might define large arrays such that each row has its own segment paragraph.

The 80386 CPU has a third mode that permits direct 32-bit addressing of 4 gigabytes (albeit few computer users have quite this much fast memory available). A scheme for addressing large amounts of RAM in 80386 machines (without leaving MS-DOS) has been discussed in *Dr. Dobb's Journal*¹⁶.

For 8086 PC's and/or real-mode programming on 80286+ machines, we can merely ignore whether segments overlap. Oddly, standard assembly programming books¹⁷ omit this way of addressing segmented memory.

The 8086 permits 32-bit addressing as long as we translate 32-bit addresses to the segment + offset notation expected by the 80x86 processors in real mode. A word that performs this conversion is **SEG.OFF**, defined as

```
: >SEG.OFF          ( d -- seg off)
  OVER 15 AND        ( -- d off)
    -ROT   D16/  DROP      ( -- off seg)
    SWAP ;
```

The 32-bit address is placed on the stack as a double-length integer, with the low-order (*i.e.* offset part) above the segment part. The phrase **OVER 15 AND** saves bits 0-3 (of the 32-bit

-
5. See, e.g., Al Williams, "DOS + 386 = 4 Gigabytes", *Dr. Dobb's Journal*, July 1990, p. 62.
 7. C. Morgan and M. Waite, *8086/8088 16-bit microprocessor primer* (Byte/McGraw-Hill, Peterborough, 1982); L. Scanlon, *IBM PC & XT assembly language: a guide for programmers* (Brady/Prentice-Hall, Bowie, Md., 1983); R. Lafore, *Assembly language primer for the IBM PC & XT* (Plume/Waite, New York, 1984).

address); **-ROT D/16 DROP** then shifts the (32-bit) address right 4 bits and drops the least-significant part, to produce the offset. This conversion method produces offsets in the range 00-0F (hex), that clearly have nothing to do with the original offset (that led to the 32-bit absolute address *via* $16 \cdot \text{seg} + \text{off}$).

§§4 A general typed-array definition

For the new syntax to work the word } must compute the address of the n'th element of V{ from the information on the stack, and } } must do the same for M{{. In order to encompass matrices of typed data we specify that the results of the phrases V{ n } and M{{ m n }} be to leave the generalized address on the parameter stack, i.e. to leave the stack picture (- seg off type) exactly as with SCALARs.

Before we can define }, however, we must specify the data structure it operates on, i.e. the array header.

Once again we begin with the user interface. We can opt for maximum generality or maximum simplicity. My first attempt fell into the first category, permitting the user to define a named segment of given length and to define an array in that segment. Lately I realized this generality accomplishes little, so have abandoned it. All arrays will be defined in the heap, named **SUPER SEG** as above. To define a length- 50 **1ARRAY** of 4-byte numbers we will say

```
50 LONG REAL*4 1ARRAY V{
```

Now, before we work out the mechanics of **1ARRAY**, we imagine that an array will be stored as in Fig. 5-4 below:

The proposed data structure consists of an 8-byte header (the **array descriptor**) in the dictionary (LISTS in HS/FORTH), with the **body** of the array stored elsewhere. The array descriptor points to the absolute address of the array data (body).

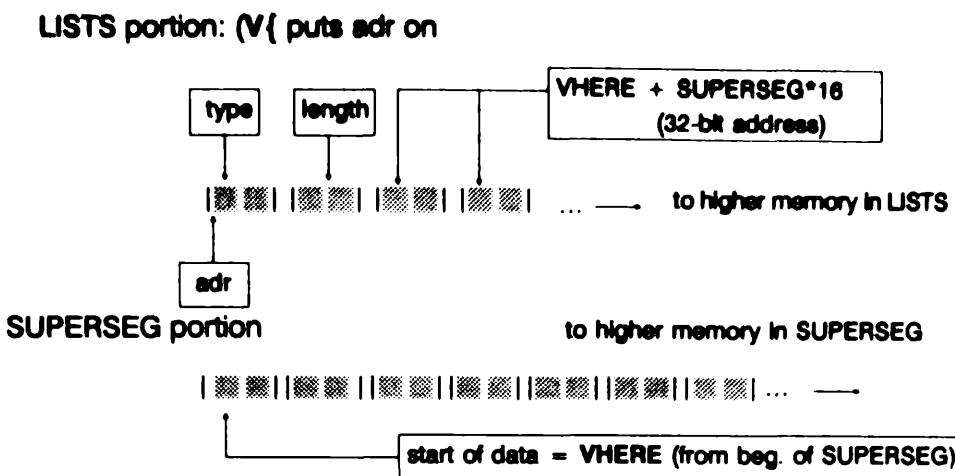


Fig. 5-4 Structure of a 1-dimensional array in SUPERSEG

The array-defining word **1ARRAY** must perform the following tasks:

- place the length and type of the data in the first two cells (4 bytes) of the array descriptor.
- place the 32-bit address (start of data) in the next two cells (4 bytes) of the array descriptor.
- allot the necessary storage in **SUPERSEG**.
- at run-time, place the generalized address, length and type on the parameter stack.

The start of data is handled by **VHERE**, a word that puts the next vacant (32-bit) address in **SUPERSEG** on TOS.

We define **VALLOT** to keep track of the storage used by arrays. **VALLOT** increments the pointer in **VHERE** (and aborts with a warning if the segment length is exceeded).

We first define some auxiliary words:

```

MEMORY 4+ @
S->D DCONSTANT MEM.START
\ beg. of free memory

```

```

40.960 DCONSTANT MAX.PARS \40960 = 655360/16

: TOTAL.PARS MAX.PARS MEM.START D- ;
\ # pars avail. mem.

1 SEGMENT SUPERSEG \ named seg. 1 byte long
TOTAL.PARS DROP FREE-SIZE \ tell DOS and HS/FORTH

DVARIABLE VHERE>
: INIT.VHERE> 0.0 VHERE> DI ;
INIT.VHERE>
: VHERE ( -- d.offset) VHERE> D@ ;
: D4/ D2/ D2/ ; : D16/ D4/ D4/ ;

: TOO-BIG? VHERE >SEG.OFF
0 AND TOTAL.PARS D>
ABORT" INSUFFICIENT ROOM IN SUPERSEG" ;
\ check whether new value of VHERE> passes end
\ of SUPERSEG

: VALLOT ( d.#bytes -- )
VHERE D+ DDUP TOO.BIG?
VHERE> D! ;

\ Array-defining words
\ Ex: 50 LONG REAL*4 1ARRAY V{
\ V{ ( -- adr)
\ V{ 17 } >FS ( :: -- V[17])}

: LONG DUP ;
FIND D, 0= ?( :D, SWAP , , ;
\ conditionally compile D,
: 1ARRAY ( -- )
CREATE UNDER D, \ t,l into 1st 4 bytes ( -- lt)
SUPERSEG @ 16 M* \ start of SUPERSEG
VHERE D+ D, \ abs. address → next 4 bytes
#BYTES M* ( lt -- #bytes to allot)
VALLOT ; \ allot space in the segment
\ run-time action: ( -- adr)

```

We also need some words to go with **1ARRAY**:

```
: }      ( adr n -- seg.off[n] t )
SWAP DUP@ R> \ type → rstack
4+ D@
ROT R@ #BYTES M*
D+ >SEG.OFF R> ; ( -- seg.off[n] t )
```

Finally, here is a useful diagnostic word

```
: ?TYPE ( t-- ) \ it's ok for this to be slow!
DUP 0 = IF DROP ." REAL*4" EXITTHEN
DUP 1 = IF DROP ." REAL*8" EXIT THEN
DUP 2 = IF DROP ." COMPLEX" EXIT THEN
DUP 3 = IF DROP ." DCOMPLEX" EXIT THEN
." NOT A DEFINED DATA TYPE" ABORT ;
```

§§5 2ARRAY and } }

We now want to define arrays of higher dimensionality. For example, to define a 2-dimensional array we might say

```
90 LONG BY 90 WIDE COMPLEX 2ARRAY XA{{
```

This leads to the definitions

```
: BY ; \ a do-nothing word for style
: WIDE * ; ( l w -- l*w )
: 2ARRAY ( l*w t - ) 1ARRAY ;
```

Now let us define } } to fetch the double-indexed address:

```
: } } ( adr m n -- a[m*l + n] t )
>R OVER 2+ @ ( -- adr m l*w )
* R> + } ;
```

By correct factoring (putting some of the work into **WIDE**) we achieved an easy definition of **2ARRAY**. Careful factoring also let us define } } in terms of } .

§3 Tuning for speed

Some of the words in our typed-data/matrix lexicons should be optimized or redefined in machine code. Accessing matrix elements imposes a non-trivial overhead on matrix operations. We can reduce the execution time with inline code, either in the traditional FORTH manner *via* selected assembler definitions, or with a recursive-descent optimizer such as HS/FORTH's¹⁸.

Experience teaches that optimization is most fruitful (most bang for the buck) applied to entire inner loops and other selected areas of code, rather than to access words *per se*. By hand-coding the innermost loop in matrix inversion and FFT routines, one achieves programs that run in (asymptotically) minimum time on the 8086/8087 chip set.

Significant speed increases in data access could perhaps be obtained with multiple code field (MCF) words, as described by Shaw¹⁹, and as implemented by HS/FORTH in the words **VAR**, **AT**, **IS**, and variants thereof. The disadvantage of MCF style is that compile-time binding, while faster in execution, loses the flexibility of run-time binding. That is, data types would — as with FORTRAN — be specified at compile-time, and lexicons would be recompiled to run with specific types. Run-time binding as described in this Chapter produces *generic* words that can handle all four standard scientific data types, a major advantage over MCF.

FORTH data structures, especially as defined in this Chapter, do little— or no bounds checking, hence do not prevent accidentally overwriting key parts of the operating system.

The new fetch and store words were defined in high-level FORTH for safety. Adding bounds-checking to arrays, at least during the debug cycle, is strongly recommended to avoid crashing, or even damaging, the system.

18. J.S. Callahan, *Proc. 1988 Rochester FORTH Conference* (Inst. for Applied FORTH Research, Inc., 1988), p. 39.
19. G. Shaw, "Forth Shifts Gears, I", *Computer Language* (May 1988) p. 67; "Forth Shifts Gears, II", *Computer Language* (June 1988) p. 61; *Proc. 9th Asilomar FORMAL Conference (JFAR 5)* (1988) 347.)

Programming Examples

Contents

Chapter 6 – Programming Examples	115
§1 Infinite series	116
§§1 Examples of infinite series	116
§§2 Numerical examples of convergent series	118
§§3 The infinite sum program	122
§2 Transcendental equations	127
§§1 Binary search	127
§§2 Regula falsi	128
§3 Ordinary differential equations	131
§§1 Runge-Kutta method	132
§§2 An implicit Runge-Kutta formula	136

This chapter illustrates how we may apply the floating point extensions of FORTH, developed in preceding chapters, to some standard problems in numerical analysis.

§1 Infinite series

Frequently we must evaluate a function defined by an infinite sum

$$f(x) = \sum_{n=0}^{\infty} c_n x^n \quad (1)$$

where x is a real number and c_n is an infinite sequence of coefficients. The extensive mathematical theory¹ of such functions can be summarized as follows: the series of terms only has meaning if – for fixed x – the sequence of partial sums

$$f_N(x) = \sum_{n=0}^{N-1} c_n x^n \quad (2)$$

has a definite limit.

§§1 Examples of infinite series

Perhaps the formal definitions will seem clearer after some concrete examples. We now examine some divergent and convergent series.

§§1-1 Divergent examples

What does it mean to say a series diverges? Consider the series whose terms are all 1's (that is, $c_n = 1$, $x = 1$):

$$1 + 1 + 1 + 1 + \dots \quad (3)$$

The partial sum f_N is just N , and therefore increases without limit as more terms are added. The series *diverges*.

1. The *Handbook of Mathematical Functions* ed. Milton Abramowitz and Irene Stegun (Dover Publications, Inc., New York, 1965) – henceforth abbreviated HMF – is a mine of useful information and references, on this as well as many other aspects of numerical analysis.

A harder case is the series whose terms are alternately 1's and -1's:

$$1 - 1 + 1 - 1 + 1 - \dots \quad (4)$$

Depending on how the terms are grouped together, the partial sums can have any value. Certainly the partial sums do not settle down to any definite value. The series *diverges*.

Yet a third case is the *harmonic series*

$$f_N = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{N}. \quad (5)$$

It can be shown that for large N, f_N is approximately $\log_e(N)$, so the harmonic series *diverges*.

§§1-2 A convergent example

Are there ever cases of infinite series that *do* mean something? Obviously, or there could hardly be a theory of them! An example² is $c_n = 1$, $x = \frac{1}{2}$. Here the partial sums

$$\begin{aligned} f_0 &= 1 \\ f_1 &= 1 + \frac{1}{2} \\ f_2 &= 1 + \frac{1}{2} + \frac{1}{4} \\ &\dots \\ f_N &= 1 + \frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{2^N} \end{aligned} \quad (6)$$

$$\lim_{N \rightarrow \infty} f_N \equiv \lim_{N \rightarrow \infty} \frac{1 - (\frac{1}{2})^N}{1 - \frac{1}{2}} = 2$$

do have a definite limit, so the series *converges*.

This is the example of Zeno's paradox where you are at one end of a sofa and an attractive person of the opposite gender is at the other end. You move half the distance, then half the remainder. Clearly an infinite number of moves is necessary to achieve the desired proximity. Thus, according to Zeno, you never get there. According to Eq. 6, however, you *do* get there and in a finite time, to boot!

§§2 Numerical examples of convergent series

How do we tell whether a series has converged? As a practical matter, we keep adding terms to the partial sum until it no longer changes, within the desired precision. Sometimes this can involve a great many terms, even when the series converges. There exists an extensive mathematical literature on testing for the convergence of an infinite series. Mathematicians have also developed many tricks for accelerating the convergence of slowly converging series³. Consulting the literature in difficult cases is strongly recommended — it can save time galore!

As an heuristic exercise, let us write some simple programs to evaluate partial sums and see how convergence works.

First we sum the terms 2^{-n} from Eq. 6. The flow diagram is shown in Fig. 6-1 below.

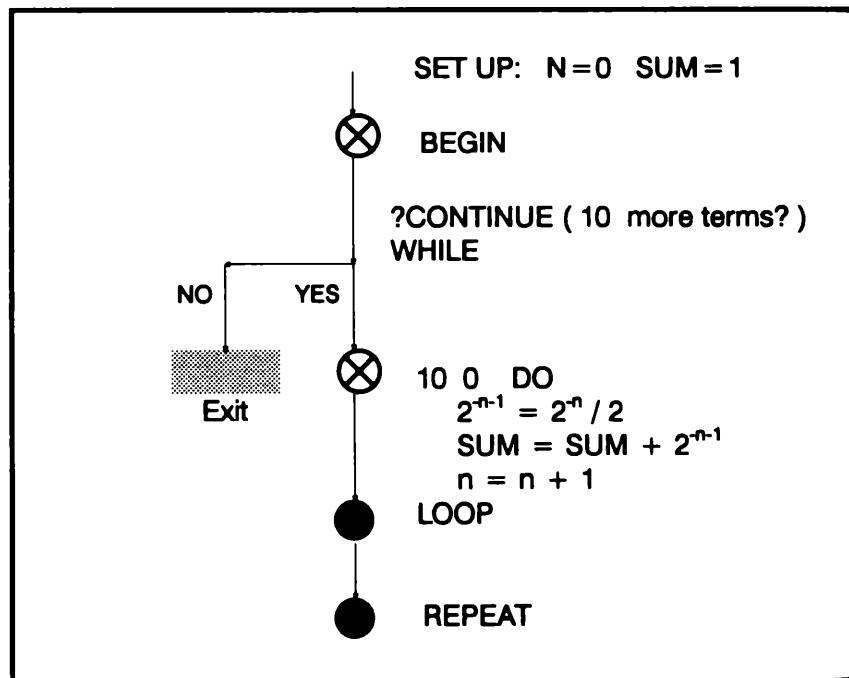


Fig. 6-1 Computing 2 the hard way

3. HMF, Ch. 3 §6 ff.

The corresponding program is

```

15 #PLACES !           \ set F. to 15 digits
:NEXT.TERM
  ( n -- n+1 :: sum 2^-n -- sum' 2^-n-1 )
  F2/ FUNDER F+ FSWAP 1+ ;
:SET.UP FINIT F=1 F=1 0 ;
:EXHIBIT CR DUP ."n = "
  2 SPACES
  FOVER ."sum = " F. ;
:?CONTINUE CR ."Another 10 terms?" ?YN ;
\ ?YN expects a "y" or "n" from the keyboard and
\ leaves -1 ("true") if "y" is pressed, 0 if "n"
:SUM SET.UP
  BEGIN EXHIBIT ?CONTINUE
  WHILE 10 0 DO NEXT.TERM LOOP
  REPEAT ;

```

This is what a run looks like:

```

FLOAD TEST.SUM Loading TEST.SUM ok
SUM
n = 0 sum = 1.000000000000000
Another 10 terms? Y
n = 10 sum = 1.99902343750000
Another 10 terms? Y
n = 20 sum = 1.99999904632568
Another 10 terms? Y
n = 30 sum = 1.9999999906867
Another 10 terms? Y
n = 40 sum = 1.9999999999909
Another 10 terms? N ok

```

The partial sums converge rapidly to 2 (which, as we saw in §1.1.2 above, is the exact sum).

For a second example, let us sum a standard infinite series representation for $\pi/4$: We note that the function $\tan^{-1}(x)$ (that is, $\arctan(x)$) has the infinite series representation⁴

$$\tan^{-1}(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} \dots \quad (7)$$

Since a 45° right triangle has equal height and base, the tangent of 45° is 1 (side opposite over side adjacent). That is,⁵,

$$\frac{\pi}{4} \equiv \tan^{-1}(1) = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} \dots \quad (8)$$

Actually the series Eq. 8 is slowly converging and therefore a poor way to compute π ⁶. But anyway, let us proceed. The flow diagram is now that of Fig. 6-2 below.

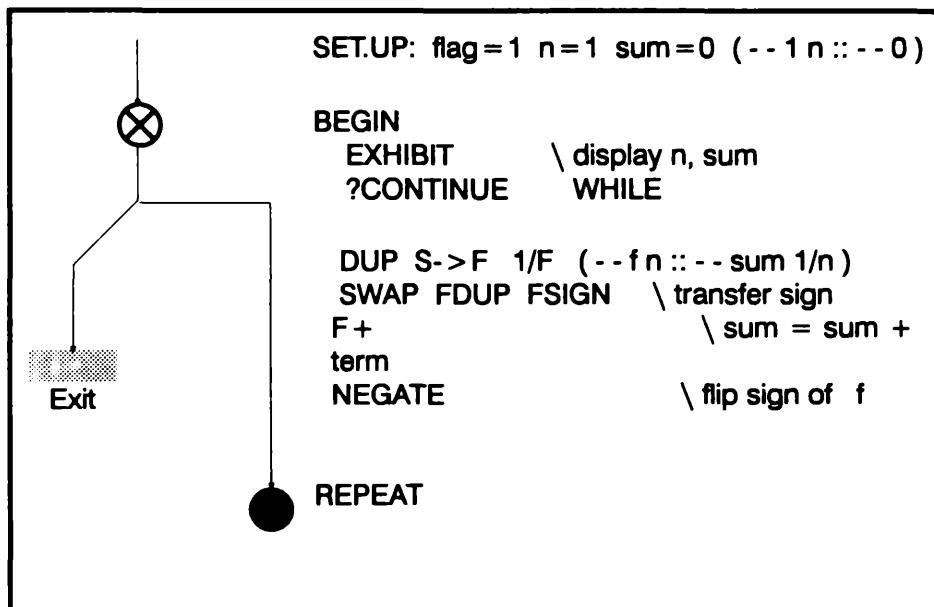


Fig. 6-2 Computing $\pi/4$ by the infinite series for $\tan^{-1}(1)$

-
- 5. Since $\pi/4$ radians is 45° .
 - 6. A better method would be to evaluate the series for $x = 1/\sqrt{3}$ which is the tangent of $\pi/6$, or $x = \sqrt{2}-1$, the tangent of $\pi/8$;

The corresponding program is

```

15 #PLACES !           \set F. to 15 digits
:NEXT.TERM (f 2n+1 - f' 2n+3 :: sum -- sum')
    DUP S->F 1/F SWAP DUP FSIGN F+
    NEGATE SWAP 2+ ;
:SET.UP FINIT F=0 1 1 ;
:EXHIBIT CR DUP
    ."n = " . 2 SPACES
    FDUP F2* F2* ."4*sum = " F. ;
:?CONTINUE ."Another term?" ?YN ;

:PI/4 SET.UP
BEGIN EXHIBIT
?CONTINUE
WHILE NEXT.TERM REPEAT ;

```

Now we run the program⁷:

FLOAD PIBY4 Loading PIBY4 ok
PI/4

n = 1 4*sum = 0.00000000000000 Another term? Y	n = 25 4*sum = 3.05840278592733 Another term? Y
n = 3 4*sum = 4.00000000000000 Another term? Y	n = 27 4*sum = 3.21840278592733 Another term? Y
n = 5 4*sum = 2.68888888888888 Another term? Y	n = 29 4*sum = 3.07025461777918 Another term? Y
n = 7 4*sum = 3.48888888888888 Another term? Y	n = 31 4*sum = 3.20818585228194 Another term? Y
n = 9 4*sum = 2.89523809522809 Another term? Y	n = 33 4*sum = 3.07915330419742 Another term? Y
n = 11 4*sum = 3.33988253980253 Another term? Y	n = 35 4*sum = 3.20036551540854 Another term? Y
n = 13 4*sum = 2.97804617604617 Another term? Y	n = 37 4*sum = 3.08807980112383 Another term? Y
n = 15 4*sum = 3.28373848373848 Another term? Y	n = 39 4*sum = 3.19418790923194 Another term? Y
n = 17 4*sum = 3.01707181707181 Another term? Y	n = 41 4*sum = 3.09162380686783 Another term? Y
n = 19 4*sum = 3.25236503471887 Another term? Y	n = 43 4*sum = 3.18918478227759 Another term? Y
n = 21 4*sum = 3.04183981892940 Another term? Y	n = 45 4*sum = 3.09616152646364 Another term? Y
n = 23 4*sum = 3.23231580940559 Another term? Y	n = 47 4*sum = 3.18505041536253 Another term? N ok

The first thing we notice is that the numbers seem to be converging to *something*; however, unlike the previous series, the differences between successive partial sums are fairly large.

An infinite series of terms that alternate in sign and decrease in magnitude is guaranteed to converge⁸. The error (that is, the difference between a partial sum and the limit) is of the order of

Note we have set up to display π rather than $\pi/4$.

This theorem, due to Weierstrass, is found in all standard intermediate-level calculus texts.

the first neglected term and has the same sign. We see that for this case, the error in computing π is of order $2/n$, where n is the number of terms in the sum. To get π to 3 significant figures, therefore, we need about 1000 terms! This is why the series representation for $4\tan^{-1}(1)$ is not a very good way to calculate π . A better way uses e.g.,

$$\pi = 16\tan^{-1}\left(\frac{1}{5}\right) - 4\tan^{-1}\left(\frac{1}{239}\right),$$

which converges much faster⁹.

§§3 The infinite sum program

Evaluating a function from its infinite series representation Eq. 1 provides an illustration both of indefinite loops and of tests of floating point numbers. We anticipate a program structure something like this:

```
BEGIN
    Calculate next term
    Not converged?
WHILE
    Update:
        Add next term to sum
        increment n
REPEAT
```

To actually write the program, we begin at the end, by specifying how we want to invoke the function.

Functions in the standard FORTH library (Ch. 3 §3) typically expect a single real number on the fstack replacing it with the function: $x \rightarrow f(x)$. Since the sum is infinite, we supply the coefficients c_n as a function of n rather than as an array. For any given $c_n = f(n)$, it is easy enough to write a FORTH word that evaluates it and leaves the result on the appropriate stack (87stack, fstack).

9. See, e.g., J. Mathews and R.L. Walker, *Mathematical Methods of Physics*, 2nd ed. (W.A. Benjamin, Inc., New Jersey, 1970).

ifstack). For example, to evaluate the exponential via the power series

$$e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \quad (9)$$

we would define the word **NEXT.TERM** as

```
: NEXT.TERM  (87: term x n -- term*x/[n + 1] x  n + 1 )
F=1 F+  \n -> n + 1
FROT FOVER F/  (87: -- x n + 1 term/[n + 1])
FROT FUNDER F* F-ROT ;
```

§§3-1 Function calls in FORTRAN

However, FORTRAN (as well as languages that emulate it) achieves readability by passing arguments to functions in a list. In fact, function names can also be passed in the argument list. Thus, e.g., a general FORTRAN program to evaluate a function by summing an infinite series could be written

```
REAL FUNCTION XINFSUM(X, E, C)
C
C      EVALUATE INFINITE POWER SERIES
C
      EXTERNAL C
      REAL X, E, C
      SUM = 0
      TERM = 1
      N = 1
1      SUM = SUM + TERM
      TERM = TERM * X*C(N)
      N = N + 1
      IF (TERM .LE. E) RETURN
      GOTO 1
END
```

where the program to evaluate the exponential would be

```
REAL FUNCTION EXP(X)
EXTERNAL XINFSUM, COEFEXP
REAL C0, XINFSUM
COMMON /CBLK/C0
C0 = 1
EXP = XINFSUM(X, 1.E-7, COEFEXP)
RETURN
END
```

given the coefficient function

```
REAL FUNCTION COEFEXP(N)
COMMON /CBLK/C0
C0 = C0/N
COEFEXP = C0
RETURN
END
```

§§3–2 A function protocol for FORTH

We would like to extend to FORTH FORTRAN's ability to write a generic series summation function, passing the variable and the name of the coefficient function as arguments. In other words, we now face the task of devising the function protocol we plan to use throughout the rest of the book and in our future programming — a heavy responsibility since we do not know what form these future programs will take. **For once we must engage in top-down programming!**

We want our protocol to have several features:

- it must be **telegraphic**, i.e. it must immediately suggest what it is doing — a matter of choosing good names.
- it must be **simple** to implement and easy to remember — the advantages of using it must not be outweighed by complexity.
- it must be **fast** — a major drawback to FORTRAN's way of doing things is the overhead in function calls.
- It must be **portable** — it cannot depend on specific details of the FORTH implementation or the machine it is running on.

It is simpler to thread this particular maze in reverse: begin with where we want to end up, and determine what steps got us there. We suppose we have defined a generic power series summation function, using the **lstack** defined in Ch. 5 §2§§5:

```

: SUM.POWERS          ( 87: err -- :: x -- sum)
  E GI                \ store error
  FS>F DUP F>FS      \ get type of x
  DUP G=1              \ x**0
  G=0 0 DUP            ( -- n=0 :: - x 10 )
  adr.c EXECUTE G+    ( -- 0 :: - x 1 c[0] )
  BEGIN 1+              ( -- n :: - x x**n-1 sum )
  FS>F GOVER G*        ( -- n 87: -- sum :: - x x**n)
  DUP adr.c EXECUTE   ( -- n :: - x x**n c[n] )
  3 GPICK G* F>FS      ( -- n :: - x x**n term sum)
  ENUF? NOT WHILE
  G+
  REPEAT G+ CLEAN.UP ;

```

The words **E**, **ENUF?** and **CLEAN.UP** have straightforward definitions with obvious meanings; **adr.c** has not been defined because we have yet to figure out what it will do.

Manifestly, **adr.c** must place the execution address (“code-field address” — **cfa**) on the stack for **EXECUTE** to find. There are several ways to accomplish this. Clearly, we want to keep the variable that holds the **cfa** local, so it will not get confused with another function’s **cfa**. While it is straightforward to define a variable and then make it headerless —hence local— (via **BEHEAD**”, e.g.) we would prefer to avoid defining a variable at all. Here is a perfect opportunity to use the **return stack** (**rstack**).

We imagine that **SUM.POWERS** expects the **cfa** of the function **c.** on the stack. Then the first thing **SUM.POWERS** must do is stash the **cfa** somewhere convenient but local. One such place is the stack itself. Even easier, since **SUM.POWERS** does not use the **rstack** explicitly (e.g. in a **DO ... LOOP**), is to let the first step in **SUM.POWERS** be **> R**. Then the code for **adr.c** would be merely **R@**. The final word, after invoking **CLEAN.UP** (that drops unwanted items from the various stacks), then must be **RDROP** (for systems without it, : **RDROP R > DROP ;**).

The revised version of **SUM.POWERS** is then

```
: SUM.POWERS      ( adr.c -- 87: err -- :: x -- sum)
    >R           \adr.c -> rstack
    E GI          \store error
    FS>F DUP F>FS \get x's type
    DUP G=1       \x**0
    G=0 0 DUP     ( -- n=0 :: - - x 1 0 )
    R@ EXECUTE G+   ( -- 0 :: - - x 1 c[0] )
    BEGIN 1+       ( -- n :: - - x x**n-1 sum )
    FS>F GOVER G*   ( -- n 87: -- sum :: - - x x**r
    DUP R@ EXECUTE ( -- n :: - - x x**n c[n] )
    3 GPICK G* F>FS( -- n :: - - x x**n term sum)
    GOVER
    ENUF? NOT WHILE G+ REPEAT
    G+ CLEAN.UP RDROP ;
```

The full program to evaluate the exponential by summing power series would then have the form shown below:

<pre>\ evaluate exponential by summing series REAL*4 SCALAR E : ENUF? (:: term --) (-- f) GABS FS>F EG@ F> ; : CLEAN.UP (n -- :: x x**n sum -- sum) DROP FS>F GDROP GDROP F>FS ; \ definition of SUM.POWERS from above BEHEAD' E CLEAN.UP \ hide these def'n REAL*8 SCALAR c F=1 c G! : COEF.EXP (n --) (:: -- c[n]) DUP 1 <= IF F=1 c G! F=1 DROP ELSE c G@ S>F F* FDUP c G! 1/F</pre>	<pre>THEN REAL*8 F>FS ; BEHEAD' c \ hide this variable : USE([COMPILE] ' CFA LITERAL ; IMMEDIATE \ this means "use address of" \ -- crucial def'n of function lexicon : E**X (:: x -- e**x) % 1.E-8 \ error on 87stack USE(COEFF.EXP \ adr.c on stack SUM.POWERS ;</pre>
--	--

§2 Transcendental equations

A transcendental equation has the form

$$f(x) = 0, \quad (10)$$

where $f(x)$ is a transcendental function rather than, say, a polynomial or ratio of polynomials¹⁰ (rational function).

There are several standard methods for finding a (or possibly, *the*) value of x that satisfies this equation, i.e. a root of Eq. 10 (which might have many or no roots). To guarantee that we find a root we must know an interval of the x -axis that it certainly will be found in. Two methods that can be applied under these circumstances are binomial search and regula falsi .

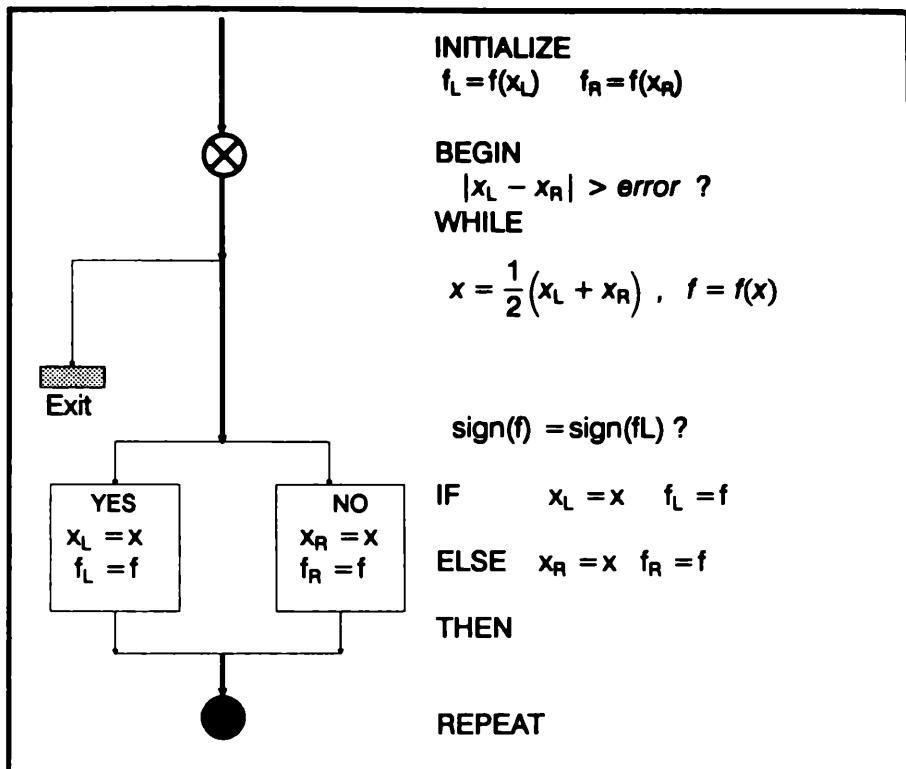
§§1 Binary search

Let us look first at binomial search, since its algorithm is easy to understand. We know some interval, $x_L \leq x \leq x_R$, contains a root because $f(x)$ changes sign when x goes from $x_L \rightarrow x_R$. In pseudocode (and FORTH flow chart) the binomial search algorithm is shown in Fig. 6-3 on page 128.

The method begins with upper and lower bounds on x that capture the root. Next we look at $f(x_{AV})$ halfway between x_L and x_R . If $f = f(x_{AV})$ has the same sign as $f_L = f(x_L)$, the new left end of the interval becomes x_{AV} . If the signs are opposite, x_{AV} becomes the new right end of the interval. The algorithm is done when left and right ends agree within some predetermined accuracy.

Binary search has the following virtues: the time it takes to achieve a given accuracy is predictable, and it is guaranteed to find a captured root. Creating a FORTH program from the pseudocode skeleton of Fig. 6-3 is left as an exercise.

10. We specialize to transcendental equations because our root-finding methods will work also for polynomials, whereas the methods developed for polynomials will not work in the more general case.

Fig. 6-3 Binary search algorithm for roots of $f(x)$

§§2 Regula falsi

Now we look at *regula falsi*, Latin for “rule of false approach”. Here the basic premise is:

- Assume the root lies in the interval (x_L, x_R) , and plot a straight line between the points (x_L, f_L) and (x_R, f_R) .
- This line must intersect the x-axis somewhere in the interval and we take that point, call it x' , as our next guess.
- If x' is to the left of the root, adjust the interval accordingly, and the same if x' is to the right of the root.

As Fig. 6-4 on page 129 shows, the straight line is supposed to approximate the curve $f(x)$. The new guess may be much closer to the root than the previous one.

the root than is the midpoint of the interval (which was the next guess in binomial search).

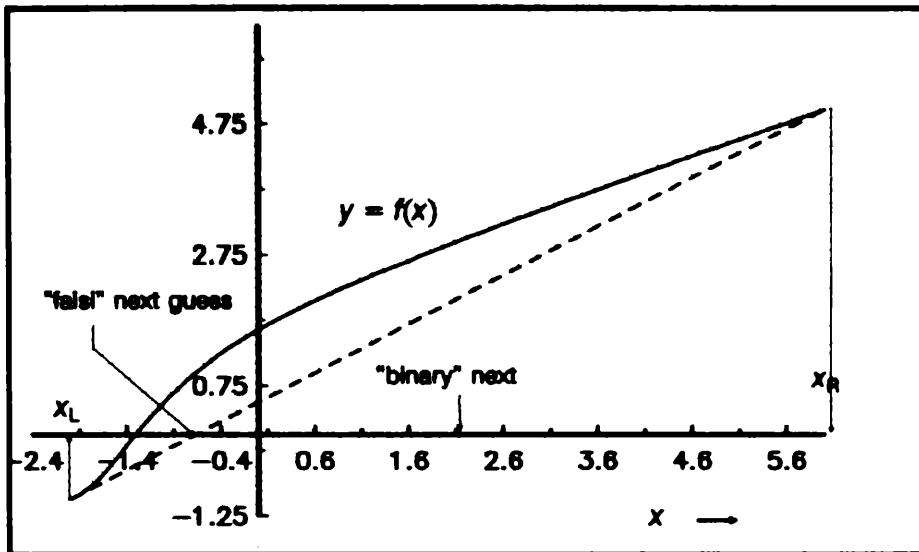


Fig. 6-4 Graphical illustration of regula falsi

A straight line in the x - y plane has the analytic form

$$y = ax + b \quad (11)$$

where a and b are constants. The intercept of the straight line with the x -axis is gotten by setting $y = 0$ and solving for x :

$$x' = \frac{-b}{a} \quad (12)$$

To determine a and b we use the two equations

$$\begin{aligned} f_L &= ax_L + b \\ f_R &= ax_R + b \end{aligned} \quad (13)$$

giving

$$a = \frac{1}{2} \left[f_R - f_L - \frac{f_R - f_L}{x_R - x_L} (x_L + x_R) \right] \quad (14)$$

and thus

$$x' = \frac{f_R x_L - f_L x_R}{f_R - f_L}. \quad (15)$$

A FORTH flow diagram for this algorithm appears in Fig. 6-5.

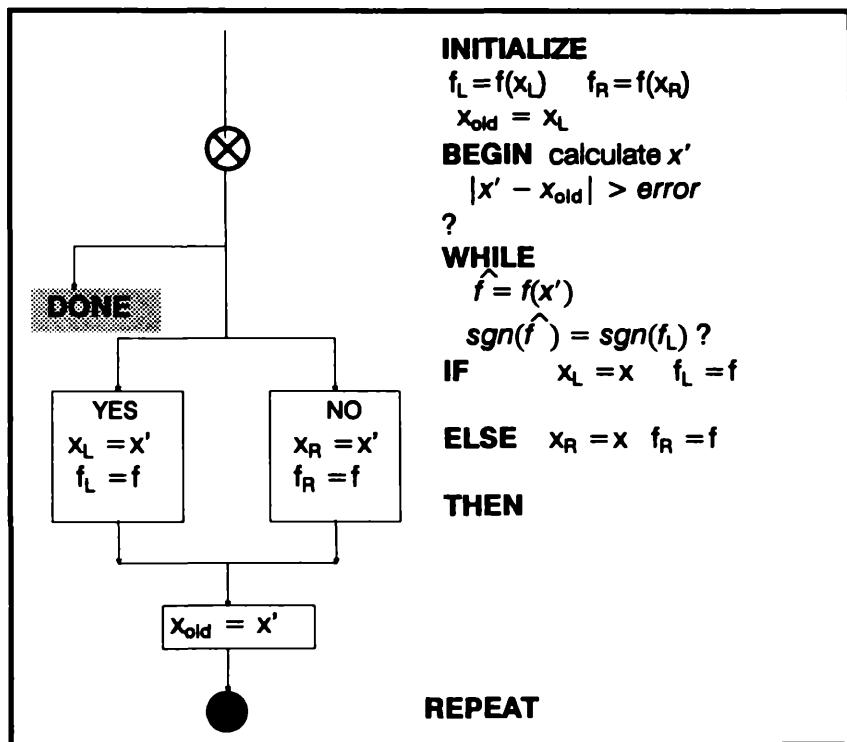


Fig. 6-5 *Regula falsi* algorithm for roots of $f(x)$

The corresponding program is shown on page 132.

Here is an example of the program in action:

```

INIT 6 #PLACES ! ok      \ set display to 6 digits
\ Example: f(x) = exp(-x) - x
:FNA FDUP FNEGATE FEXP FR- ;
\ Clearly, the root lies between 0 and 3. (Why?)

```

```

USE( FNA % 0. % 3. % 1.E-6 )FALSI
\ st(4)  st(3)  st(2)    x'  x' - xold
?????? ?????? ?????? .759452 -.291530
?????? ?????? ?????? .588025 -.0326025
?????? ?????? ?????? .569459 -.00362847
?????? ?????? ?????? .567400 -.000403560
?????? ?????? ?????? .567171 -.0000448740
?????? ?????? ?????? .567146 -.0000050002
?????? ?????? ?????? .567143 -.0000005544 ok

```

The display was generated by the word **.FS** – placed in the definition of **APART?** for debugging – we show the top 5 of the eight 80x87 registers. The **??????** means the contents of that register are not a properly defined fp number, either because of a mistake or because nothing was stored in them after **FINIT**. Here, since the program is obviously working, the latter explanation is the correct one.

§3 Ordinary differential equations

We wish to solve the first-order general differential equation

$$\dot{x} \equiv \frac{dx}{dt} = f(x,t) \quad (16)$$

In general we can only solve Eq. 16 approximately, starting from the value of x – call it x_0 – at some initial time t_0 , then advancing the time by small increments $dt = h$, using the differential equation itself to give us $x(t+h)$ given $x(t)$.

For example, we could expand in Taylor's series¹¹

$$x(t+h) = x(t) + h\dot{x}(t) + \frac{h^2}{2}\ddot{x}(t) + \dots \quad (17)$$

11. See, e.g., HMF §3.6.1.

```

\ USE( Fname % a % b % err )FALSI
( 87: -- root)
\ Fname is the name of a FORTH function

\ function notation
: USE( [COMPILE] ' CFA LITERAL ;
    IMMEDIATE

6 REAL*4 SCALARS ERR XL XR YL YR OLDX
0 VAR f1      \ a place to store cfa
: SAME.SIGN? ( 87: x y -- -- f )
    F* F0> ;

: INITIALIZE ( cfa -- 87: a b e -- )
    IS f1 \ store cfa
    XDUP \ interval has root?
    SAME.SIGN?
    ABORT" Even number of roots!!!"
    XR G! XL G!
    XL G@ f1 EXECUTE YL G!
    XR G@ f1 EXECUTE YR G!
    F=0 OLDX G! ;
    )FALSI ( cfa -- 87: a b e -- root) INITIALIZE
    : X'   XL G@ FR G@ ( 87: -- x')
        FUNDER F* ( 87: -- yR xL*yR)
        XR G@ YL G@
        FUNDER F* ( 87: -- yR xL*yR yL xR*yL)
        FROT F- ( 87: -- yR yL xR*yL -xL*yR)
        F-ROT F- F/ ;
    : APART? ( 87: x' -- x' -- f )
        FDUP OLDX G@ F-
        .FS FABS ERR G@ F> ;
    : REVISE ( 87: x' -- )
        FDUP f1 EXECUTE ( 87: -- x' y')
        FDUP YL G@
        SAME.SIGN? FOVER
        ( -- f 87: -- x' y' x')
        IF     XL G! YL G!
        ELSE XR G! YR G! THEN
        OLDX G! ; ( -- 87: -- )
    : )FALSI ( cfa -- 87: a b e -- root) INITIALIZE
    :

```

and keep only the lowest order terms:

$$x(t+h) \approx x(t) + hf(x(t), t). \quad (18)$$

§§1 Runge-Kutta method

One standard class of methods that had fallen into disfavor but now are popular again, are the Runge-Kutta algorithms¹². These algorithms can be classified according to order n (that is, if h is the step size, the error at each step will be $O(h^n)$). The second order Runge-Kutta algorithm is ($x' \equiv x(t+h)$, $x \equiv x(t)$)

$$k = hf(x, t)$$

(19)

$$x' = x + \frac{1}{2}(k + hf(x+k, t+h)) + O(h^3).$$

How does this work? Clearly,

$$\begin{aligned} k + hf(x+k, t+h) &= hf(x, t) + hf(x, t) + h^2 \frac{\partial f}{\partial t} + hk \frac{\partial f}{\partial x} \\ &\equiv 2h\dot{x}(t) + h^2 \ddot{x}(t) + O(h^3) \end{aligned} \quad (20)$$

Substituting 19 in 20 we now find

$$x' = x(t+h) = x(t) + h\dot{x}(t) + \frac{h^2}{2} \ddot{x}(t) ;$$

that is, the Runge-Kutta x' agrees with the Taylor's series expansion 17, to $O(h^3)$.

The flow chart of second-order Runge-Kutta is shown in Fig. 6-6 below. We express the algorithm in FORTH as shown in Fig. 6-7 on page 134 below.

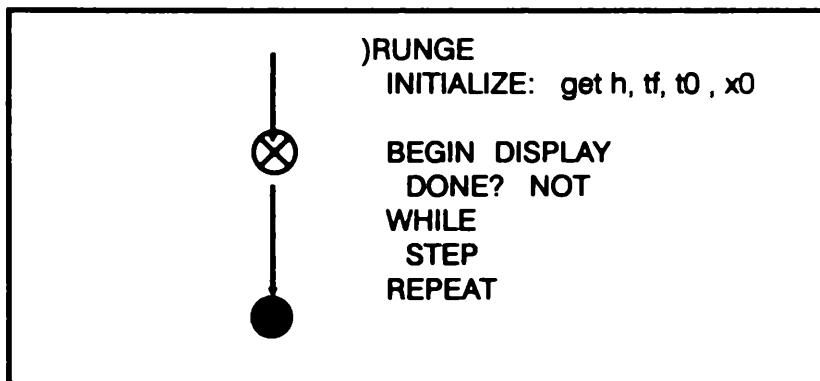


Fig. 6-6 2nd-order Runge-Kutta for $dx/dt = f(x, t)$

<pre>\ STRAIGHT 2ND ORDER RUNGE-KUTTA \ SOLUTION OF FIRST-ORDER DIFEQ \ dx/dt = f(x,t) \ See Abramowitz & Stegun, HMF §25.5.6 \ Usage: \ USE(FNB % x0 % t0 % tf % h)RUNGE \ \ FNB (: [t] -- 87: x -- f[x,t]) evaluates f(x,t) \ x0 = starting value of dep. variable \ t0 = initial time \ tf = end time \ h = step.size \ \ k = hf(x,t), x' = x + (k + hf(x+k, t+h))/2 6 REAL*4 SCALARS T T' H X TMAX K 0 VAR f1 \ to hold cfa : USE([COMPILE]' CFA LITERAL; IMMEDIATE : INITIALIZE (:cfa -- 87: x0 t0 tf h --) IS f1 H G! TMAX G! T G! X G! ;</pre>	<pre>\ These words increment x & t. : Inc.T T G@ HG@ F+ T G! ; : Inc.X X G@ T f1 EXECUTE (87: -- f[x,t]) H G@ F* (87: -- k = hf[x,t]) FDUP K G! \ save k X G@ F+ (87: -- x + k) T' G@ T G! T f1 EXECUTE (87: -- f[x+k, t+h]) H G@ F* K G@ F+ F2/ (87: -- [k + f[x+k, t+h]] /2) X G@ F+ X G! ; : DONE? T G@ TMAX G@ F> ; (--- f) 0 VAR exact \ cfa : DISPLAY exact EXECUTE X G@ T G@ CR F. F. F. ; \ emit "t x exact" :)RUNGE (:cfa -- 87: x0 t0 tf h --) BEGIN DISPLAY DONE? NOT</pre>
---	--

Fig. 6-7 Explicit 2nd-order Runge-Kutta solver

As an example of second-order Runge-Kutta in action, let us solve numerically the equation

$$\dot{x} = t^2 e^{-x} \quad (21)$$

with the initial condition $x(t=0) = 0$, whose exact solution is

$$x(t) = \log_e \left(1 + \frac{1}{3} t^3 \right). \quad (22)$$

t	x	x_{ex}	t	x	x_{ex}
.0000	.0000	.0000	2.5999	1.9254	1.9256
.10000	.00060	.00033	2.6999	2.0228	2.0230
.20000	.00300	.00286	2.7999	2.1181	2.1183
.30000	.00946	.00898	2.8999	2.2113	2.2115
.40000	.02177	.02111	2.9999	2.3023	2.3025
.50000	.04164	.04082	3.0999	2.3912	2.3915
.60000	.07049	.06953	3.1999	2.4781	2.4784
.70000	.10934	.10825	3.2999	2.5630	2.5633
.80000	.15875	.15757	3.3999	2.6459	2.6462
.90000	.21877	.21752	3.4999	2.7270	2.7273
1.0000	.28898	.28768	3.5999	2.8081	2.8085
1.1000	.36846	.36718	3.6999	2.8836	2.8839
1.2000	.45612	.45489	3.7999	2.9592	2.9598
1.3000	.55063	.54946	3.8999	3.0333	3.0336
1.4000	.65061	.64954	3.9999	3.1057	3.1060
1.5000	.75473	.75377	4.0999	3.1768	3.1769
1.6000	.86175	.86091	4.1999	3.2460	3.2463
1.7000	.97061	.96989	4.2999	3.3139	3.3142
1.8000	1.0803	1.0797	4.3999	3.3804	3.3808
1.9000	1.1902	1.1897	4.4999	3.4456	3.4460
2.0000	1.2996	1.2992	4.5999	3.5095	3.5099
2.1000	1.4080	1.4078	4.6999	3.5722	3.5725
2.2000	1.5151	1.5149	4.7999	3.6336	3.6339
2.2999	1.6206	1.6205	4.8999	3.6939	3.6942
2.3999	1.7242	1.7241	4.9999	3.7531	3.7534
2.4999	1.8258	1.8258	5.0999	3.8111	3.8114

Fig. 6-8 Second-order Runge-Kutta - results

Thus define

```
: FNB (:[T] -- 87: x -- f[x,t])
  FNNEGATE FEXP G@ F**2 F* ;
: EXACT T G@ FDUP FDUP F* F* (87: -- t^3)
  3 S->F F/ F=1 F+ FLN ;
```

and say

```
USE( EXACT IS exact ok
USE( FNB % 0. % 0. % 5. % 0.1 )RUNGE
```

The resulting output is shown in Fig. 6-8 above.

§§2 An implicit Runge-Kutta formula

A variation on straight Runge-Kutta is a so-called *implicit algorithm*¹³. For example, in the second-order formulae given above, suppose $x + k$ were replaced by x' :

$$k = hf(x, t) \quad (23)$$

$$x' = x + \frac{1}{2} (k + hf(x', t+h)) + O(h^3)$$

and the resulting (transcendental) equation solved for x' by –say– *regula falsi*. Since we have already written a *regula falsi* program, we can apply it here to get the algorithm shown diagrammatically in Fig. 6-9 below. We program it¹⁴ as shown in Fig. 6-10 on page 137 below.

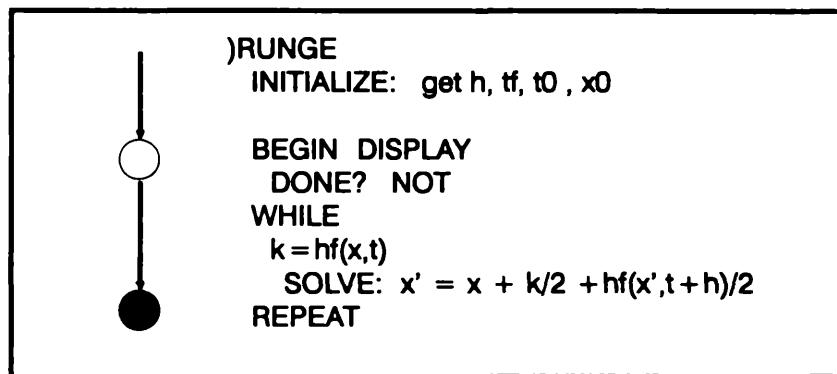


Fig. 6-9 2nd-order *implicit Runge-Kutta* for $dx/dt = f(x, t)$

-
- 13. See, e.g., A. Ralston, *A First Course in Numerical Analysis* (McGraw-Hill Book Company, New York, 1965) Ch. 5. Implicit methods increase the stability of numerical solution, compared with explicit methods. The formula below is exact for second-order polynomials. The error is of the same order as the explicit formula, but the coefficient may be smaller.
 - 14. Note: “?(” means “conditionally execute to next right parenthesis”.

Now we consider the same example as previously:

```
: FNB      (: [T] -- 87: x -- f[x,t] )
    FNNEGATE FEXP G@ F**2 F* ;
: EXACT   T G@ FDUP FDUP F* F* (87: -- t^3)
    F=3 F/ F=1 F+ FLN ;
```

and say

```
USE( EXACT IS exact ok
USE( FNB % 0. % 0. % 5. % 0.1 )RUNGE ok
```

<pre> FIND)FALSI 0= ?(FLOAD FALSI.FTH) 7 REAL*4 SCALARS T T' H X X' TMAX K 0 VAR f1 \ to hold cfa of f(x,t) : INITIALIZE IS f1 H G! TMAX G! T G! X G! ; \ These words increment x & t : Inc.T T G@ HG@ F+ T G! ; : k X G@ T f1 EXECUTE (87: -- f[x,t]) H G@ F* K G! ; (87: -- k = hf[x,t]) : X" (87: x' -- g[x']) T f1 EXECUTE (87: -- f[x',t+h]) H G@ F* K G@ F+ F2/ (87: -- [k + hf[x+k,t+h]] /2)</pre>	<pre> X G@ F+ ; % 3. FCONSTANT F=3 : INTERVAL X G@ FDUP K G@ F=3 F* F+ ; : X' USE(X' INTERVAL % 1.E-6)FALSI ; : Inc.X k X' X G! T G@ T G! ; : DONE? T G@ TMAX G@ F> ; (:--f) 0 VAR exact \ cfa : DISPLAY exact EXECUTE X G@ T G@ CR F. F. F. ; \ emit "t x exact" :)RUNGE (: cfa -- 87: x0 10 tf h --) BEGIN DISPLAY</pre>
--	--

Fig. 6-10 Implicit 2nd-order Runge-Kutta program

The resulting output is shown in Fig. 6-11 on page 138.

Clearly the implicit form is more accurate; whether the putative gain in stability justifies solving a transcendental equation is unclear, however.

t	x	x _{ex}	t	x	x _{ex}
00000	.00000	.00000	2.5999	1.9253	1.9255
.10000	.00050	.00033	2.6999	2.0228	2.0230
.20000	.00299	.00266	2.7999	2.1181	2.1183
.30000	.00945	.00896	2.8999	2.2113	2.2115
.40000	.02173	.02111	2.9999	2.3024	2.3025
.50000	.04155	.04082	3.0999	2.3914	2.3915
.60000	.07032	.06953	3.1999	2.4783	2.4784
.70000	.10906	.10825	3.2999	2.5632	2.5633
.80000	.15834	.15757	3.3999	2.6461	2.6462
.90000	.21822	.21752	3.4999	2.7272	2.7273
1.0000	.28825	.28768	3.5999	2.8064	2.8065
1.1000	.36762	.36718	3.6999	2.8838	2.8839
1.2000	.45518	.45489	3.7999	2.9595	2.9596
1.3000	.54962	.54946	3.8999	3.0336	3.0336
1.4000	.64958	.64954	3.9999	3.1060	3.1060
1.5000	.75370	.75377	4.0999	3.1769	3.1769
1.6000	.86077	.86091	4.1999	3.2463	3.2463
1.7000	.96969	.96989	4.2999	3.3142	3.3142
1.8000	1.0795	1.0797	4.3999	3.3808	3.3808
1.9000	1.1895	1.1897	4.4999	3.4460	3.4460
2.0000	1.2990	1.2992	4.5999	3.5099	3.5099
2.1000	1.4075	1.4078	4.6999	3.5725	3.5725
2.2000	1.5147	1.5149	4.7999	3.6340	3.6339
2.2999	1.6202	1.6205	4.8999	3.6942	3.6942
2.3999	1.7239	1.7241	4.9999	3.7534	3.7534
2.4999	1.8256	1.8258	5.0999	3.8114	3.8114

Fig. 6-11 Second order *implicitRunge-Kutta* – results

Let us now compare the two algorithms for functions $f(x,t)$ that lead to singular solutions. This time we consider the equation

$$\dot{x} = t^2 e^x \quad (24)$$

whose exact solution is

$$x(t) = -\log_e \left(1 - \frac{1}{3}t^3 \right). \quad (25)$$

Manifestly, 25 blows up at $t = 3^{1/3} = 1.442\dots$. We expect this behavior to become apparent in the numerical solution. So we s

```
: FNB  FEXP  G@  F**2  F* ;
( [t] -- 87: x -- f[x,t] )

: EXACT  F=1  T G@          ( 87:--t^3)
FDUP FDUP F* F* F=3 F/ F+ FLN ;
```

and say again

```
USE( EXACT IS exact ok
USE( FNB % 0. % 0. % 5. % 0.1 )RUNGE ok
```

The results of doing this with straight-, and then implicit Runge-Kutta are displayed in Fig. 6-12 (p. 140) and 6-13 (p. 141), respectively. We only show the second half of the interval (near the singular point) in either case.

The straight Runge-Kutta algorithm, without the fancy implicit solution for $x(t+h)$, appears more accurate near the singularity, although both methods are acceptably accurate. Does this mean implicit Runge-Kutta is no good? No!

The implicit scheme lost accuracy through roundoff: the arithmetic was insufficiently precise. To take advantage of the method's power, we must increase the precision beyond one part in 10^6 . This requires changing all scalars to 64-bit precision (**REAL*8**) rather than 32-bit as we have done here. The generic fetch/store techniques developed in Chapter 5 and used here, permit this change with a minimum of fuss. We leave this as an exercise.

t	x	x _{ex}	t	x	x _{ex}
.71999	.13287	.13286	1.0899	.56508	.56508
.72999	.13889	.13888	1.0999	.58840	.58838
.73999	.14512	.14511	1.1099	.60860	.60857
.74999	.15156	.15154	1.1199	.63171	.63169
.75999	.15821	.15820	1.1299	.65580	.65578
.76999	.16509	.16508	1.1399	.68093	.68091
.77999	.17220	.17219	1.1499	.70718	.70715
.78999	.17955	.17954	1.1599	.73461	.73458
.79999	.18714	.18713	1.1699	.76331	.76329
.80999	.19499	.19497	1.1799	.79337	.79335
.81999	.20309	.20308	1.1899	.82491	.82489
.82999	.21147	.21145	1.1999	.85803	.85801
.83999	.22012	.22010	1.2099	.89287	.89286
.84999	.22906	.22904	1.2199	.92959	.92958
.85999	.23829	.23828	1.2299	.96834	.96834
.86999	.24783	.24782	1.2399	1.0093	1.0093
.87999	.25769	.25767	1.2499	1.0527	1.0527
.88999	.26788	.26786	1.2599	1.0989	1.0989
.89999	.27841	.27839	1.2699	1.1481	1.1482
.90999	.28928	.28926	1.2799	1.2007	1.2008
.91999	.30053	.30051	1.2899	1.2571	1.2572
.92999	.31215	.31213	1.2999	1.3179	1.3180
.93999	.32417	.32415	1.3099	1.3836	1.3837
.94999	.33660	.33657	1.3199	1.4550	1.4552
.95999	.34945	.34943	1.3299	1.5332	1.5334
.96999	.36274	.36272	1.3399	1.6193	1.6196
.97999	.37650	.37648	1.3499	1.7151	1.7154
.98999	.39074	.39072	1.3599	1.8226	1.8231
.99999	.40548	.40546	1.3699	1.9449	1.9457
1.0099	.42075	.42073	1.3799	2.0865	2.0876
1.0199	.43656	.43654	1.3899	2.2539	2.2557
1.0299	.45295	.45293	1.3999	2.4581	2.4611
1.0399	.46995	.46993	1.4099	2.7187	2.7242
1.0499	.48757	.48755	1.4199	3.0760	3.0884
1.0599	.50586	.50584	1.4299	3.6376	3.6782
1.0699	.52485	.52483	1.4399	4.8831	5.3657
1.0799	.54458	.54456			

Fig. 6-12 Straight Runge-Kutta for singular case

Chapter 6 - Programming Examples

t	x	x_{ex}	t	x	x_{ex}
.71999	.13288	.13288	1.0799	.54464	.54466
.72999	.13890	.13890	1.0899	.56515	.56508
.73999	.14513	.14511	1.0999	.58848	.58838
.74999	.15156	.15154	1.1099	.60888	.60857
.75999	.15822	.15820	1.1199	.63180	.63169
.76999	.16510	.16508	1.1299	.65590	.65578
.77999	.17221	.17219	1.1399	.68104	.68091
.78999	.17956	.17954	1.1499	.70729	.70715
.79999	.18715	.18713	1.1599	.73473	.73458
.80999	.19500	.19497	1.1699	.76345	.76329
.81999	.20310	.20308	1.1799	.79353	.79336
.82999	.21148	.21145	1.1899	.82508	.82489
.83999	.22013	.22010	1.1999	.85822	.85801
.84999	.22907	.22904	1.2099	.89309	.89286
.85999	.23831	.23828	1.2199	.92963	.92958
.86999	.24785	.24782	1.2299	.96861	.96834
.87999	.25771	.25767	1.2399	1.0096	1.0093
.88999	.26789	.26786	1.2499	1.0531	1.0527
.89999	.27842	.27839	1.2599	1.0993	1.0989
.90999	.28830	.28926	1.2699	1.1486	1.1482
.91999	.30055	.30051	1.2799	1.2013	1.2008
.92999	.31217	.31213	1.2899	1.2578	1.2572
.93999	.32419	.32415	1.2999	1.3186	1.3180
.94999	.33662	.33657	1.3099	1.3845	1.3837
.95999	.34947	.34943	1.3199	1.4561	1.4552
.96999	.36277	.36272	1.3299	1.5345	1.5334
.97999	.37653	.37648	1.3399	1.6209	1.6196
.98999	.39077	.39072	1.3499	1.7171	1.7154
.99999	.40551	.40546	1.3599	1.8252	1.8231
1.0099	.42078	.42073	1.3699	1.9485	1.9457
1.0199	.43660	.43654	1.3799	2.0914	2.0876
1.0299	.45300	.45293	1.3899	2.2612	2.2557
1.0399	.46999	.46993	1.3999	2.4696	2.4611
1.0499	.48762	.48755	1.4099	2.7395	2.7242
1.0599	.50592	.50584	1.4199	3.1222	3.0884
1.0699	.52491	.52483	1.4299	3.8149	3.6782

Fig. 6-13 Implicit Runge-Kutta for singular case

Complex arithmetic in FORTH

Contents

§1 The complex number system	144
§§1 Cartesian representation of complex numbers	144
§§2 Polar representation of complex numbers	147
§2 Load, store, manipulate fstack	148
§3 Arithmetic operations	149
§4 Roots of complex numbers	151
§§1 Complex square roots	152
§§2 The complex square root program	154
§5 Complex exponentials and trigonometric functions	154
§6 Logarithms	155

One of the most crucial features of FORTRAN for scientific computation is the ease with which it embeds complex arithmetic into formulae. No other compiled language has this feature¹. From time to time someone gets a bright idea, and adds complex arithmetic to Pascal *via* subroutines (since Pascal functions can return only a single number, and complex functions must return two numbers)². The same problem would afflict C and the new structured BASICs.

Recently, a mechanism for adding complex arithmetic to Pascal by defining a stack and using postfix notation has been proposed³. Does this sound at all familiar?

This chapter deals with complex arithmetic and its implementation in FORTH. We do not consider complex numbers whose real

1. APL gracefully admits complex numbers and functions, but is an *interpreted* language.

2. D.D. Clark, "Simple Calculations with Complex Numbers", *Dr. Dobb's Journal*, October 1984, p. 30.

3. D. Gedeon, "Complex Math in Pascal", *Byte Magazine*, July 1987, p.121.

and imaginary parts are integers, since these are virtually useless in scientific computing. Therefore we perform complex arithmetic solely in single- or double-precision floating point format. We begin with a brief review of the complex number system.

The complex number system

The algebraic equation

$$x^2 - 1 = 0 \quad (1)$$

has 2 roots, ± 1 , in the standard number system (real numbers). But the (otherwise quite similar) equation

$$x^2 + 1 = 0 \quad (2)$$

has no roots. That is, there is no ordinary number which, when squared, gives a negative result. Eighteenth century mathematicians disliked a state of affairs wherein some polynomial equations of n'th degree had n roots, whereas others had n-2, n-4, etc. This seemed disorderly and unpredictable. How much simpler life would be if every polynomial of n'th order could always be factored into n primitives of the form (x_n are roots)

$$\begin{aligned} a_0 + a_1x + a_2x^2 + \dots + a_nx^n \\ \equiv a_n(x - x_1)(x - x_2) \dots (x - x_n) \end{aligned} \quad (3)$$

§§1 Cartesian representation of complex numbers

In order that every polynomial have n roots it was necessary to extend the idea of number to include objects of the form

$$z = x 1 + y i \quad (4)$$

where i is – by definition – a “number” whose square is -1 ; 1 is a “number” whose square is $+1$, and x and y are ordinary numbers. Conventionally, x is called the **real part** of z , and y the **imaginary part**:

$$x = \text{Re}(z), y = \text{Im}(z). \quad (5)$$

Thus the solutions of the polynomial equation

$$z^2 + 1 = 0 \quad (6)$$

are

$$z = 01 \pm 1i \quad (7)$$

(that is, $x = 0, y = \pm 1$) .

To reassure readers who find uncomfortable the notion of a "number" $i = \sqrt{-1}$ whose square is negative, it is possible to find a 2×2 matrix representation for 1 (unit matrix) and i :

$$1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad i = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \quad (8)$$

It is easy to verify that – in the sense of matrix multiplication –

$$i \times i = -1, \quad 1 \times i = i \times 1 = i, \quad \text{and } 1 \times 1 = 1. \quad (9)$$

These are just the usual multiplication rules for complex numbers. Note that $x 1$ and $y i$ then mean multiplication of a matrix by a scalar:

$$x 1 \equiv \begin{pmatrix} x & 0 \\ 0 & x \end{pmatrix}, \quad y i \equiv \begin{pmatrix} 0 & y \\ -y & 0 \end{pmatrix}$$

$$z = x + iy \equiv \begin{pmatrix} x & y \\ -y & x \end{pmatrix}$$

The complex numbers obey all the algebraic rules of ordinary arithmetic – commutative and associative laws of multiplication and addition. They are complete in the sense that when we multiply or add two complex numbers we get a complex number, not some other kind of number:

$$(a + ib)(x + iy) = (ax - by) + i(ay + bx) \quad (10)$$

$$(a + ib)(x + iy) = (ax - by) + i(ay + bx) \quad (10)$$

For every complex number, $z = x + iy$, there is a corresponding complex conjugate complex number,

$$z^* = x - iy. \quad (11)$$

Each non-zero complex number z has a multiplicative inverse $1/z$. To see this, multiply numerator and denominator of $1/z$ by the complex conjugate Eq. 11, and note that

$$zz^* \equiv z^*z = x^2 + y^2$$

is a non-zero real number. We can therefore calculate its inverse by ordinary division, and (scalar-) multiply z^* by the result:

$$\frac{1}{z_2} \equiv \frac{z_2^*}{z_2 z_2^*} \equiv \left(\frac{x_2}{x_2^2 + y_2^2} - i \frac{y_2}{x_2^2 + y_2^2} \right). \quad (12)$$

To divide one complex number z_1 by another, z_2 , invert z_2 and multiply: $z_1/z_2 \equiv z_1 \times (1/z_2)$, i.e.

$$\frac{z_1}{z_2} \equiv (x_1 + iy_1) \times \left(\frac{x_2}{x_2^2 + y_2^2} - i \frac{y_2}{x_2^2 + y_2^2} \right) \quad (13)$$

(Numbers that satisfy the addition, multiplication and closure properties of real or complex numbers are said to form a field.)

It is easy to prove that even when the coefficients of an n'th degree polynomial are complex, it has n roots that can be represented as complex numbers.

Complex numbers can be used to represent points in the x-y plane with y plotted vertically and x horizontally⁴. (Sometimes this graphical representation is called an Argand plot or Argand diagram. The x-y plane is then called the Argand plane.)

4. The 2-dimensional graphical representation of complex numbers makes complex arithmetic attractive for computer graphics.

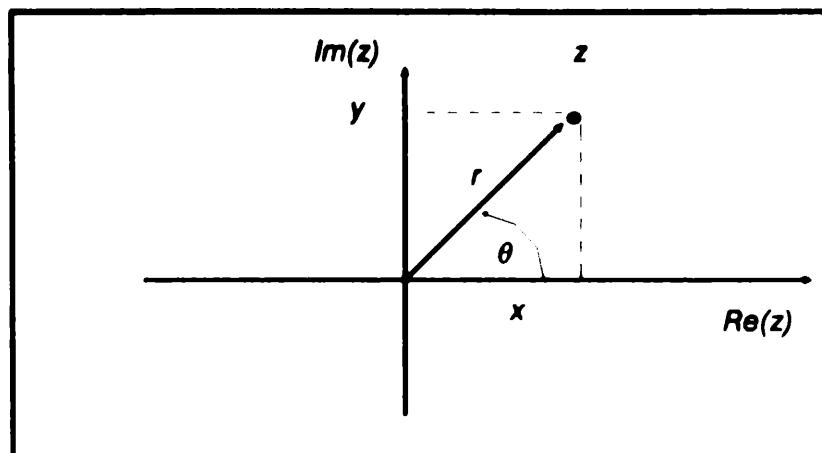


Fig. 7-1 Representing a complex number as a point in the Argand plane, in Cartesian and polar coordinates

§§2 Polar representation of complex numbers

The complex number $z = x + iy$ is said (remember the Argand plane) to be represented in **Cartesian form** (after René Descartes, the inventor of analytic geometry, i.e. the idea of graphing equations). However, it is equally valid to use the relationship (**Euler's theorem**)

$$e^{i\theta} \equiv \cos \theta + i \sin \theta \quad (14)$$

to write the **polar representation** of z ,

$$z = re^{i\theta}, \quad r \geq 0. \quad (15)$$

Clearly, since $z^* \equiv r e^{-i\theta}$, we have

$$r = \sqrt{zz^*} = (x^2 + y^2)^{\frac{1}{2}} \quad (16)$$

and⁵

$$\theta = \tan^{-1}(y/x). \quad (17)$$

5. See Ch. 3 §§§1 for definitions of the inverse trigonometric functions.

Note that θ , sometimes called $\text{Arg}(z)$, is defined only up to multiples of 2π – that is, adding 2π to an angle changes neither *sine* nor *cosine* because of their periodicity. The relations between (x,y) and (r, θ) are illustrated in Fig. 7-1 above:

§2 Load, store, manipulate fstack

We now define FORTH words to perform complex arithmetic.
All complex operations will be prefixed by the letter X⁶. We assume all complex operations are performed on the FPU for speed, hence we shall give 87stack diagrams.

In 87stack (or fstack) diagrams z stands for complex number, x for real part, and y for imaginary part. Where useful, the 87stack diagrams show the operations decomposed into real and imaginary parts. By convention the imaginary part is higher on the fstack than the real part.

By now most FORTH code should be self-explanatory. Many words have been coded in high-level FORTH because the overhead of threading is negligible compared with the time spent executing.

With by-now obvious meanings we have

```
\ -- single & double-precision complex fetch and store.
: X@ DUP R32@ 4+ R32@ ; ( adr -- 87: -- z)
: X! DUP 4+ R32! R32! ; ( adr -- 87: z -- )
CODE 8+ BX 08 IW ADD. END-CODE
: DX@ DUP R64@ 8+ R64@ ; ( adr -- 87: -- z)
: DX! DUP 8+ R64! R64! ; ( adr -- 87: z -- )
\ -- end complex fetch and store.
```

6. The letter C is more mnemonic, but FORTH conventionally reserves C for prefixing byte ("character") operations – as in C@, CI, etc. The complex extension supplied with HS/FORTH uses the prefix C when it is unambiguous (as in C+, C-, C*, C/, etc.), and CP when C alone won't do, as in CP@, etc.

```
\fstack manipulation
: REAL FDROP ;           ( 87: z -- x )
: IMAG FPLUCK ;          ( 87: z -- y )
: CONJG FNEGATE ;        ( 87: x y -- x , -y )

: XDROP FDROP FDROP ;    ( 87: z -- )
: XSWAP F4R F4R ;        ( 87: z1 z2 -- z2 z1 )
: XDUP FOVER FOVER ;     ( 87: z -- z z )
```

§3 Arithmetic operations

The standard complex operations should be virtually self-explanatory. For efficiency we define multiplication and division of a complex number by a scalar, as well as complex×complex and complex/complex.

```
: CMPLX F=0 ;           ( 87: x -- x 0 )
: X+ FROT F+ F-ROT F+ FSWAP ;
: X- FROT FR- F-ROT F- FSWAP ;
: XOVER F3P F3P ;       ( 87: z1 z2 -- z1 z2 z1 )
: X*F FUNDER F*         ( 87: x y a -- ax ay )
: F-ROT F* FSWAP ;
: F*X FROT X*F ;        ( 87: x a b -- ax bx )
\ CODE X*F 2 FMUL. 1 FMULP. END-CODE
: X*I FNNEGATE FSWAP ;
: X/F 1/F X*F ;
```

The critical operation of complex×complex can be defined in high level FORTH for portability:

```
: X*
: XOVER X*I FROT      ( 87: z1 z2 -- z1*z2 )
: X*F             ( 87: -- ab x -b a y )
: F3X FSWAP        ( 87: -- a ay x b -by )
: F4X FSWAP        ( 87: -- -by ay x a b )
: F*X             ( 87: -- -by ay xa xb )
: X+ ;
```

Actually, complex multiplication is sufficiently involved to be worth defining in code. Here is a code definition for the 80x87 family of FPUs. The equivalent for the Motorola 68881/2 family is virtually identical, allowing for minor differences in Intel and

Motorola assembler mnemonics, as well as for the fact that the 68881/2 has registers but no stack.

<u>\ operation</u>	<u>87 stack contents</u>
CODE X*	\ x y a b
3 FLD.	\ x y a b x
2 FMUL.	\ x y a b x a
4 FXCH.	\ x a y a b x
1 FMUL.	\ x a y a b x b
1 FXCH.	\ x a y a x b b
3 FMUL.	\ x a y a b x y b
4 FSUBRP.	\ x a-y b y a b x
2 FXCH.	\ x a-y b b x a y
1 FMULP.	\ x a-y b b x a y
1 FADDP.	\ x a-y b b x + a y
END-CODE	(x y a b - - x a-y b x b + y a)

Once we have multiplication, division is easy:

```
: XMODSQ          ( 87: x y -- x**2+y**2 )
  F**2 FSWAP F**2 F+ ;
: 1/X CONJG XDUP XMODSQ
  FDUP F0= ABORT" Can't divide by 0" X/F ;
: X/ 1/X X* ;      ( 87: z1 z2 -- z1/z2 )
```

With the preceding discussion and referring to Fig. 7-1 on page 147 the FORTH words to accomplish Cartesian-polar transformation and *vice versa* should also be fairly transparent⁷:

```
: ARG FSWAP FPATAN ; ( 87: x y -- atan[y/x] )
SYNONYM FATAN2 ARG
\ FORTRAN defines FATAN2 so we do also.

: XABS           ( 87: z -- |z| )
  XMODSQ FSQRT ;
: >POLAR         ( 87: x y -- r θ[radians] )
  XDUP XABS F-ROT ARG ;
: POLAR>        ( 87: r θ [radians] -- x y )
  FSINCOS FROT X*F ;
```

7. **SYNONYM** is an HS/FORTH innovation to save some code by avoiding :FATAN2 ARG ;

§4 Roots of complex numbers

The n'th root of a complex number z is that complex number, $z^{1/n}$, whose n'th power is the original number. That is, as for real numbers,

$$(z^{1/n})^n \equiv z \quad (18)$$

It might seem almost trivial to define the n'th root of a complex number in polar representation:

$$z = r e^{i\theta} \quad (19)$$

hence

$$z^{1/n} = r^{1/n} e^{i\theta/n} \quad (20)$$

Certainly we know what we mean by the n'th root of a positive real number, and from Euler's theorem we know how to evaluate

$$e^{i\theta/n} \equiv \cos \frac{\theta}{n} + i \sin \frac{\theta}{n} \quad (21)$$

However, we can also think of the n'th root as a solution of the polynomial equation in the variable w

$$w^n - z = 0. \quad (22)$$

The fundamental theorem of algebra proves that a polynomial of n'th degree has exactly n roots; this implies there are n distinct values of w that satisfy Eq. 22; i.e., there are n distinct roots of z . How can we generate them? Let us call the root shown above (in Eq. 20)

$$w_0 = r^{1/n} e^{i\theta/n}$$

Then we can multiply w_0 by a factor

$$\chi_k = e^{2\pi ik/n}, \quad k = 0, 1, \dots, n-1 \quad (23)$$

to obtain n different numbers,

$$w_k = w_0 \chi_k, \quad k=0, \dots, n-1$$

the n 'th power of each of which is z . Clearly, if k increases past $n-1$, the numbers w_k simply repeat⁸. Since it is neither possible nor desirable for a complex function to return all n roots of z , we choose the principal one, w_0 . All complex root-finding functions should obey this convention. In general the simplest algorithm to calculate the n 'th root of a positive real number is

$$r^{1/n} = e^{\ln(r)/n}. \quad (24)$$

Then to complete the job of evaluating w_0 we would need to calculate a sine and a cosine. Thus, 3 divisions, 2 multiplications and 4 transcendental function calls are generally needed to evaluate the n 'th root of a complex number.

However, for square roots ($n = 2$) there is a much more efficient method based on ordinary square roots, which we shall now describe.

§§1 Complex square roots

Our phase convention means the phase θ of the square root of z must lie between 0 and π , since that of z lies between 0 and 2π .

The algorithm can be understood using the half-angle formulae for sines and cosines (we used these to develop the trigonometric functions in Ch. 4 §6):

$$\cos\left(\frac{\theta}{2}\right) = \left(\frac{1 + \cos \theta}{2}\right)^{\frac{1}{2}} \quad (25a)$$

$$\sin\left(\frac{\theta}{2}\right) = \left(\frac{1 - \cos \theta}{2}\right)^{\frac{1}{2}} \quad (25b)$$

8. Recall $e^0 = e^{2\pi i 0} = 1$.

Now we use the fact that if $z = x + iy$, and if $w = a + ib$ is its square root, then their polar representations are

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} r \cos \theta \\ r \sin \theta \end{pmatrix} \quad (26a)$$

$$\begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} \sqrt{r} \cos(\theta/2) \\ \sqrt{r} \sin(\theta/2) \end{pmatrix} \quad (26b)$$

Therefore the principal root is

$$a = \operatorname{sgn}(y) \left[\frac{1}{2}(r + x) \right]^{\frac{1}{2}}, \quad (27a)$$

$$b = \left[\frac{1}{2}(r - x) \right]^{\frac{1}{2}}. \quad (27b)$$

That is, as we can easily see from the Argand diagram, Fig. 7-2 below, if $\operatorname{Im}(z)$ is negative, then $\operatorname{Re}(w)$ will be negative, and vice versa.

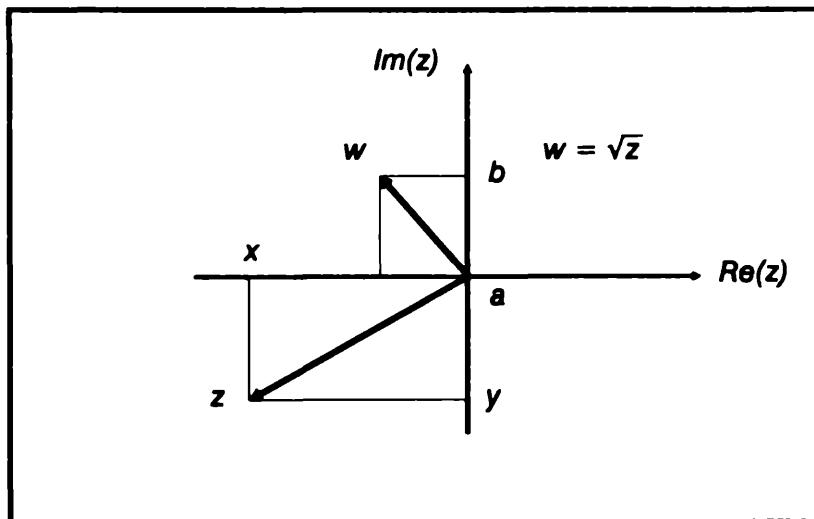


Fig. 7-2 Complex number with negative imaginary part, and its principal square root

§§2 The complex square root program

We now translate these equations into high-level FORTH with comments:

```
: XSQRT          ( 87: z -- z**1/2)
    FSWAP XDUP XABS      ( 87: -- y x r)
    FDUP F0=             \ return 0 if |z| = 0
    IF    FDROP XDROP X=0 EXIT THEN
        XDUP F+           ( 87: -- y x r r+x )
        F2X F-             ( 87: -- y r+x r-x )
        F2/ FSQRT          ( 87: -- y r+x b )
        F2X                 ( 87: -- b r+x )
        F0<                \ get sign of Im(z)
        F2/ FSQRT          ( -- f 87: -- b |a| )
        IF FNNEGATE THEN   \ fix sign of Re(z**1/2)
        FSWAP ;
```

We test to make sure $|z| > 0$, since we do not want the possibility that $|z| - x$ or $|z| + x$ works out to be negative (albeit small) through roundoff, thereby generating an error in the square root routine.

§5 Complex exponentials and trigonometric functions

Complex exponentials and trigonometric functions are nearly self-explanatory. They are based on Euler's theorem,

$$e^{i\theta} = \cos \theta + i \sin \theta.$$

Thus,

$$e^z \equiv e^{x+i y} \equiv e^x e^{i y} = e^x (\cos y + i \sin y) \quad (28)$$

Similarly,

$$\cos z = \frac{1}{2} (e^{ix-y} + e^{y-ix}) \quad (29a)$$

$$\sin z = \frac{1}{2i} (e^{ix-y} - e^{y-ix}). \quad (29b)$$

Hence the code for the complex trigonometric functions is

```

: FSINCOS      (87: x - - cos[x] sin[x] )
F2/ FTAN      (87: - - a = tan[x/2] )
FDUP F**2      (87: - - a a**2)
F=1 XDUP F+    (87: - - a a**2 1 1+a**2)
F2X F-        (87: - - a 1+a**2 1-a**2)
FOVER F/       (87: - - a 1+a**2 cos[x])
F-ROT F/ F2* ; (87: x - - cos[x] sin[x] )
\ note: FSINCOS is microcoded on the 80387

: XEXP          (87: x y - - e**x*cos[y] e**x*sin[y] )
FSINCOS FROT FEXP X*F ;

: X2/   F2/   FSWAP F2/   FSWAP ;

: XSIN          (87: x y - - sin[x]cosh[y] cos[x]sinh[y])
FNEGATE FEXP FSWAP FSINCOS F*X
XDUP 1/X X- X2/ FSWAP FNEGATE ;

: XCOS          (87: x y - - cos[x]cosh[y] -sin[x]sinh[y])
FNEGATE FEXP FSWAP FSINCOS F*X
XDUP 1/X X+ X2/ ;

```

§6 Logarithms

The logarithm of a complex number must be defined by the polar representation of the number. Thus, using the fact that

$$\log_e(ab) = \log_e(a) + \log_e(b), \quad (30)$$

and that

$$\log_e(e^z) = z, \quad (31)$$

we find it consistent to define the complex logarithm as

$$\log_e(z) = \log_e(r e^{i\theta}) \equiv \log_e(r) + i\theta. \quad (32)$$

Thus,

$$: XLOG (87: x y - - ln[r] atan[y/x])
>POLAR FSWAP FLN FSWAP ;$$

This completes our dissertation on complex arithmetic.

More Programming Examples

Contents

§1 Numerical integration	157
§§1 The integral of a function	158
§§2 The fundamental theorem of calculus	159
§§3 Monte-Carlo method	160
§§4 Adaptive methods	164
§§5 Adaptive integration on the real line	165
§§6 Adaptive integration in the Argand plane	179
§2 Fitting functions to data	181
§§1 Fast Fourier transform	184
§§2 Gram polynomials	193
§§3 Simplex algorithm	201
§3 Appendices	204
§§1 Gaussian quadrature	204
§§2 The trapezoidal rule	205
§§3 Richardson extrapolation	206
§§4 Linear least-squares: Gram polynomials	207
§§5 Non-linear least squares: simplex method	208

In this chapter we apply some of the FORTH tools we have been developing (complex arithmetic, typed data) to two standard problems in numerical analysis: numerical integration of a function over a definite interval; determining the function of a given form that most closely fits a set of data.

§1 Numerical Integration

We begin by defining the definite integral of a function $f(x)$. Then we discuss some methods for (numerically) approximating the integral. This process is called **numerical integration** or **numerical quadrature**. Finally, we write some FORTH programs based on the various methods we describe.

§§1 The integral of a function

The definite integral $\int_a^b f(x) dx$ is the area between the graph of the function and the x-axis as shown below in Fig. 8-1:

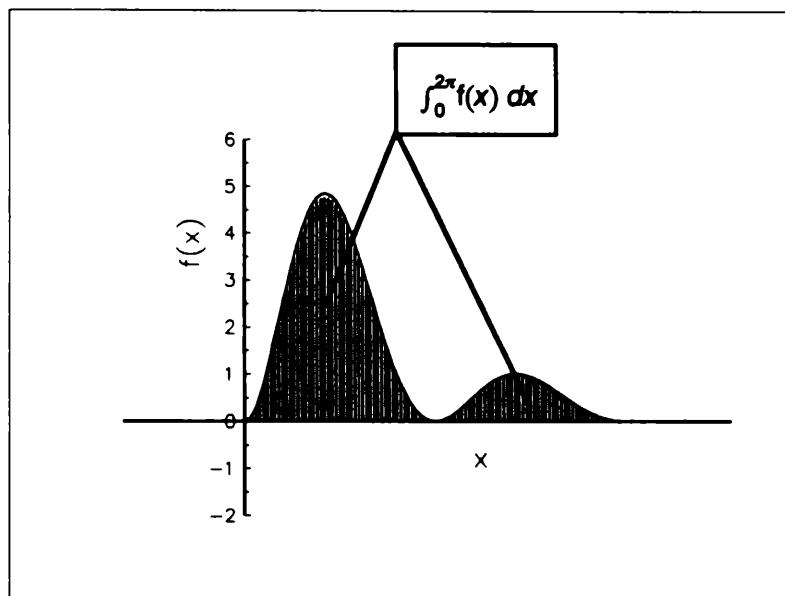


Fig. 8-1 The Integral of a function Is the area under the curve.

We estimate the integral by breaking up the area into narrow rectangles of width w that approximate the height of the curve at that point and then adding the areas of the rectangles¹. For rectangles of non-zero width the method gives an approximation. If we calculate with rectangles that consistently protrude above the curve (assume for simplicity the curve lies above the x-axis), and with rectangles that consistently lie below the curve, we capture the exact area between two approximations. We say that we have bounded the integral above and below. In mathematical language,

1. If a rectangle lies below the horizontal axis, its area is considered to be negative.

$$w \sum_{n=0}^{(b-a)/w} \min [f(a+nw), f(a+nw+w)] \leq \int_a^b f(x) dx \quad (1)$$

$$\leq w \sum_{n=0}^{(b-a)/w} \max [f(a+nw), f(a+nw+w)]$$

It is easy to see that each rectangle in the upper bound is about $w|f'(x)|$ too high² on the average, hence overestimates the area by about $\frac{1}{2}w^2|f'(x)|$. There are $(b-a)/w$ such rectangles, so if $|f'(x)|$ remains finite over the interval $[a, b]$ the total discrepancy will be smaller than

$$\frac{1}{2}w(b-a) \max_{a \leq x \leq b} |f'(x)|.$$

Similarly, the lower bound will be low by about the same amount. This means that if we halve w (by taking twice as many points), the accuracy of the approximation will double. The mathematical definition of $\int_a^b f(x) dx$ is the number we get by taking the limit as the width w of the rectangles becomes arbitrarily small. We know that such a limit exists because the actual area has been captured between lower and upper bounds that shrink together as we take more points.

§§2 The fundamental theorem of calculus

Suppose we think of $\int_a^b f(x) dx$ as a function – call it $F(b)$ – of the upper limit, b . What would happen if we compared the area $F(b)$ with the area $F(b + \Delta b)$: We see that the difference between the two is (for small Δb)

$$\Delta F(b) = F(b + \Delta b) - F(b) \approx f(b)\Delta b + O((\Delta b)^2) \quad (2)$$

2. $f'(x)$ is the slope of the line tangent to the curve at the point x . It is called the first derivative of $f(x)$.

so that

$$F'(b) = \lim_{\Delta b \rightarrow 0} \frac{1}{\Delta b} \left(\int_a^{b+\Delta b} dx - \int_a^b f(x) dx \right) \quad (3)$$

Equation 3 is a fancy way to say that integration and differentiation are **inverse operations** in the same sense as multiplication and division, or addition and subtraction.

This fact lets us calculate a definite integral using the differential equation routine developed in Chapter 6. We can express the problem in the following form:

Solve the differential equation

$$\frac{dF}{dx} = f(x) \quad (4)$$

from $x = a$ to $x = b$, subject to the initial condition

$$F(a) = 0.$$

The desired integral is $F(b)$.

The chief disadvantage of using a differential equation solver to evaluate a definite integral is that it gives us no **error criterion**. We would have to solve the problem at least twice, with two different step sizes, to be sure the result is sufficiently precise³.

§§3 Monte-Carlo method

The area under $f(x)$ is exactly equal to the average height \bar{f} of $f(x)$ on the interval $[a, b]$, times the length, $b-a$, of the interval⁴. How can we estimate \bar{f} ? One method is to sample $f(x)$ at random.

-
- 3. This is not strictly correct: one could use a differential equation solver of the “predictor/corrector” variety, with variable step-size, to integrate Eq. 4. See, e.g., Press, et al., *Numerical Recipes* (Cambridge University Press, Cambridge, 1986), pp. 102 ff.
 - 4. That is, this statement defines \bar{f} .

choosing N points in $[a,b]$ with a random number generator.
Then

$$\bar{f} = \frac{1}{N} \sum_{n=1}^N f(x_n) \quad (5)$$

and

$$\int_a^b f(x) dx \approx (b-a)\bar{f} \quad (6)$$

This random-sampling method is called the Monte-Carlo method (because of the element of chance).

§§3-1 Uncertainty of the Monte-Carlo method

The statistical notion of variance lets us estimate the accuracy of the Monte-Carlo method: The variance in $f(x)$ is

$$\begin{aligned} \text{Var}(f) &= \int_{-\infty}^{+\infty} \rho(f) (f - \bar{f})^2 df \\ &\approx \frac{1}{N} \sum_{n=1}^N (f(x_n) - \bar{f})^2 \end{aligned} \quad (7)$$

(here $\rho(f)df$ is the probability of measuring a value of f between f and $f + df$).

Statistical theory says the variance in estimating \bar{f} by random sampling is

$$\text{Var}(\bar{f}) = \frac{1}{N} \text{Var}(f) \quad (8)$$

i.e., the more points we take, the better estimate of \bar{f} we obtain. Hence the uncertainty in the integral will be of order

$$\Delta \left(\int_a^b f(x) dx \right) \approx \frac{(b-a)\sqrt{\text{Var}(f)}}{\sqrt{N}} \quad (9)$$

and is therefore guaranteed to decrease as $\frac{1}{\sqrt{N}}$.

It is easy to see that the Monte-Carlo method converges slowly.

Since the error decreases only as $\frac{1}{\sqrt{N}}$, whereas even so crude a rule as adding up rectangles (as in §§§1) has an error term that decreases as $1/N$, what is Monte-Carlo good for?

Monte-Carlo methods come into their own for multidimensional integrals, where they are much faster than multiple one-dimensional integration subroutines based on deterministic rules.

§§3–2 A simple Monte-Carlo program

Following the function protocol and naming convention developed in Ch. 6 §§§3.2, we invoke the integration routine *via*

```
USE( F.name % L.lim % U.lim % err )MONTE
```

We pass **)MONTE** the name **F.name** of the function $f(x)$, the limits of integration, and the absolute precision of the answer. The answer should be left on the ifstack. **L.lim**, **U.lim** and **err** stand for explicit floating point numbers that are placed on the 87stack by %.⁵ The word **%** appears explicitly because in a larger program — of which **)MONTE** could be but a portion — we might want to specify the parameters as numbers already on the 87stack. Since this is intended to be an illustrative program we keep the fstack simple by defining **SCALARs** to put the limits and precision into.

```
3 REAL*4 SCALARS A B-A E
```

The word **INITIALIZE** will be responsible for storing these numbers.

The program uses one of the pseudo-random number generators (**prng's**) from Ch. 3 §5. We need a word to transform prn's —uniformly distributed on the interval (0,1)— to prn's on the interval (A, B):

5. The word **%** pushes what follows in the input stream onto the 87stack, assuming it can be interpreted as a floating point number.

```
: NEW.X RANDOM B-A G@ F* A G@ F+ ;
```

The program is described by the simple flow diagram of Fig. 8-2 below:

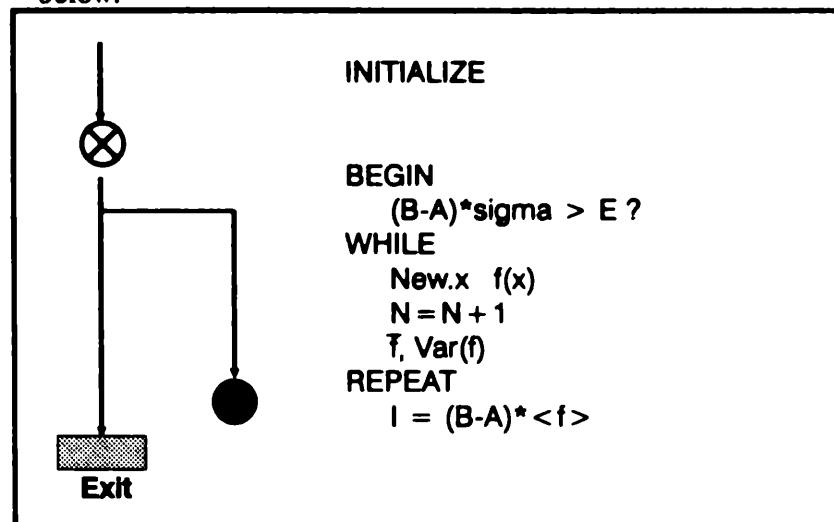


Fig. 8-2 Flow diagram of Monte Carlo integration

From the flow diagram we see we have to recompute \bar{f} and $\text{Var}(\bar{f})$ at each step. From Eq. 5 we see that

$$\bar{f}_{N+1} = \bar{f}_N + \frac{f(x_{N+1}) - \bar{f}_N}{N + 1}$$

and

$$\text{Var}_{N+1} = \text{Var}_N + \frac{(f_{N+1} - \bar{f}_N)(f_{N+1} - \bar{f}_{N+1}) - \text{Var}_N}{N + 1}$$

Writing the program is almost automatic:

```

: USE( [COMPILE] ' CFA LITERAL ; IMMEDIATE
3 REAL*4 SCALARS Av.F old.Av.F Var.F

: Do.Average ( n--n+1 87:f--f )
Av.F G@ old.Av.F G! \ save old.Av
FDUP 1+ ( --n+1 87:--ff )
dd.Av.F G@ ( 87:--ff old.Av.F)
FUNDER F.
DUP S->F F/ F+ ( 87:--f Av.F)
Av.F G! ; \ put away \ cont'd below

```

```

: Do.Variance      ( n--n 87:f-- )
    FDUP old.Av.F G@ ( 87:f f old.Av)
    FUNDER F- FSWAP
    Av.F G@ F- F*
    ( 87:[f-old.Av]*[f-Av] )
    Var.F G@ FUNDER F-
    DUP S->F F/ F+ ( 87:--Var' )
    Var.F G! ;

: INITIALIZE     (: adr-- 87:a b e --)
    IS adr.f
    E G!
    FOVER F- B-A G! A G!
    FINIT
    F=0 Var.F G!
    F=0 Av.F G!
    F=0 old.Av.F G!
    0 5 0 DO \ exercise 5 times

    NEWX adr.f EXECUTE
    Do.Average Do.Variance
    LOOP ;

    : Not.Converged? Var.F G@ FSQRT
        B-A G@ F* E G@ F> ;
    : DEBUG DUP
        10 MOD \ every 10 steps
    0= IF CR DUP.
        Av.F G@ F.
        Var.F G@ F. THEN ;
    : )MONTE
    INITIALIZE
    BEGIN DEBUG Not.Converged?
    WHILE NEWX adr.f EXECUTE
        Do.Average Do.Variance
    REPEAT
    DROP Av.F G@ B-A G@ F* ;

```

The word **DEBUG** is included to produce useful output every 10 points as an aid to testing. The final version of the program need not include **DEBUG**, of course. Also it would presumably be prudent to **BEHEAD** all the internal variables.

The chief virtue of the program we have just written is that it is easily generalized to an arbitrary number of dimensions. The generalization is left as an exercise.

§§4 Adaptive methods

Obviously, to minimize the execution time of an integration subroutine requires that we minimize the number of times the function $f(x)$ has to be evaluated. There are two aspects to this:

- First, we must evaluate $f(x)$ only once at each point x in the interval.
- Second, we evaluate $f(x)$ more densely where it varies rapidly than where it varies slowly. Algorithms that can do this are called **adaptive**.

To apply adaptive methods to Monte Carlo integration, we need an algorithm that biases the sampling method so more points are

chosen where the function varies rapidly. Techniques for doing this are known generically as **stratified sampling**⁶. The difficulty of automating stratified sampling for general functions puts adaptive Monte Carlo techniques beyond the scope of this book.

However, adaptive methods can be applied quite easily to deterministic quadrature formulae such as the **trapezoidal rule** or **Simpson's rule**. Adaptive quadrature is both interesting in its own right and illustrates a new class of programming techniques, so we pursue it in some detail.

§§5 Adaptive integration on the real line

We are now going to write an adaptive program to integrate an arbitrary function $f(x)$, specified at run-time, over an arbitrary interval of the x -axis, with an absolute precision specified in advance. We write the integral as a function of several arguments, once again to be invoked following Ch. 6 §1.3.2:

```
USE( F.name % L.limit % U.limit % err )INTEGRAL
```

Now, how do we ensure that the routine takes a lot of points when the function $f(x)$ is rapidly varying, but few when $f(x)$ is smooth? The simplest method uses **recursion**⁷.

§§5-1 Digression on recursive algorithms

We have so far not discussed recursion, wherein a program calls itself directly or indirectly (by calling a second routine that then calls the first).

Since there is no way to know *a priori* how many times a program will call itself, memory allocation for the arguments must be dynamic. That is, a recursive routine places its arguments on a

6. J.M. Hammersley and D.C. Handscomb, *Monte Carlo Methods* (Methuen, London, 1964).

7. See, e.g., R. Sedgewick, *Algorithms* (Addison-Wesley Publishing Company, Reading, MA, 1983), p. 85.

stack so each invocation of the program can find them. This is the method employed in recursive compiled languages such as Pascal, C or modern BASIC. Recursion is of course natural in FORTH since stacks are intrinsic to the language.

We illustrate with the problem of finding the greatest common divisor (gcd) of two integers. Euclid⁸ devised a rapid algorithm for finding the gcd⁹ which can be expressed symbolically as

$$\text{gcd } (u, v) = \begin{cases} u, & v=0 \\ \text{gcd } (v, u \text{ mod } v) & \text{else} \end{cases} \quad (10)$$

That is, the problem of finding the gcd of u and v can be replaced by the problem of finding the gcd of two much smaller numbers. A FORTH word that does this is¹⁰

```
: GCD           ( u v -- gcd)
?DUP 0>       \ stopping criterion
IF UNDER MOD RECURSE THEN ;
```

Here is a sample of **GCD** in action, using **TRACE**¹¹ to exhibit the rstack (in hex) and stack (in decimal):

8. An ancient Greek mathematician known for one or two other things!
9. See, e.g. Sedgewick, *op. cit.*, p. 11.
10. Most FORTHS do not permit a word to call itself by name; the reason is that when the compiler tries to compile the self-reference, the definition has not yet been completed and so cannot be looked up in the dictionary. Instead, we use **RECURSE** to stand for the name of the self-calling word. See Note 14 below.
11. **TRACE** is specific to HS/FORTH, but most dialects will support a similar operation. **SSTRACE** is a modification that single-steps through a program.

784 48 TRACE GCD

	rstack	stack
:GCD		784 48
DUP		784 48 48
0=		784 48 0
OBRANCH<8>0		784 48
UNDER		48 784 48
MOD		48 16
:GCD		48 16
DUP	4876	48 16 16
0=	4876	48 16 0
OBRANCH<8>0	4876	48 16
UNDER	4876	16 48 16
MOD	4876	16 0
:GCD	4876	16 0
DUP	4876 4876	16 0 0
0=	4876 4876	16 0 65535
OBRANCH<8>-1	4876 4876	16 0
DROP	4876 4876	16
EXIT	4876	16
EXIT		16

Note how **GCD** successively calls itself, placing the same address (displayed in hexadecimal notation) on the rstack, until the stopping criterion is satisfied.

Recursion can get into difficulties by exhausting the stack or rstack. Since the stack in **GCD** never contains more than three numbers, only the rstack must be worried about in this example.

Recursive programming possesses an undeserved reputation for slow execution, compared with nonrecursive equivalent programs¹². Compiled languages that permit recursion –e.g., BASIC, C, Pascal – generally waste time passing arguments to subroutines, i.e. recursive routines in these languages are slowed by parasitic calling overhead. FORTH does not suffer from this speed penalty, since it uses the stack directly.

12. For example, it is often claimed that removing recursion almost always produces a faster algorithm. See, e.g. Sedgewick, *op. cit.*, p. 12.

Nevertheless, not all algorithms should be formulated recursively. A disastrous example is the Fibonacci sequence

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \quad (11)$$

expressed recursively in FORTH as

```
: FIB          (: n -- F[n] )
  DUP 0> NOT
  IF DROP 0 EXIT THEN
  DUP 1 =
  IF DROP 1 EXIT THEN      \ n > 1
  1- DUP 1-               ( -- n-1 n-2 )
  RECURSE SWAP            ( -- F[n-2] n-1 )
  RECURSE + ;
```

This program is vastly slower than the nonrecursive version below, that uses an explicit **DO** loop:

```
: FIB          (: n -- F[n] )
  0 1 ROT      (: 0 1 n )
  DUP 0> NOT
  IF DDROP EXIT THEN
  DUP 1 =
  IF DROP PLUCK EXIT THEN
  1 DO UNDER + LOOP PLUCK ;
```

Why was recursion so bad for Fibonacci numbers? Suppose the running time for F_n is T_n ; then we have

$$T_n \approx T_{n-1} + T_{n-2} + \tau \quad (12)$$

where τ is the integer addition time. The solution of Eq. 12 is

$$T_n = \tau \left[\left(\frac{1 + \sqrt{5}}{2} \right)^n - 1 \right] \quad (13)$$

That is, the execution time increases exponentially with the size of the problem. The reason for this is simple: recursion managed to replace the original problem by two of nearly the same size, i.e. recursion nearly doubled the work at each step!

The preceding analysis of why recursion was bad suggests how recursion can be helpful: we should apply it whenever a given

problem can be replaced by –say– two problems of half the original size, that can be recombined in n or fewer operations. An example is *mergesort*, where we divide the list to be sorted into two roughly equal lists, sort each and then merge them:

```

subroutine sort(list[0,n])
    partition(list, list1, list2)
    sort(list1)
    sort(list2)
    merge(list1, list2, list)
end

```

In such cases the running time is

$$T_n \approx T_{n/2} + T_{n/2} + n = 2T_{n/2} + n \quad (14)$$

for which the solution is

$$T_n \approx n \log_2 (n) \quad (15)$$

(In fact, the running time for *mergesort* is comparable with the fastest sorting algorithms.) Algorithms that subdivide problems in this way are said to be of **divide and conquer** type.

Adaptive integration can be expressed as a divide and conquer algorithm, hence recursion can simplify the program. In pseudocode (actually QuickBasic®) we have the program shown below:

```

function simpson(f, a, b)
    c = (a + b)/2
    simpson = (f(a) + f(b) + 4*f(c)) * (b - a) /6
end function

function integral(f, a, b, error)
    c = (a + b)/2
    old.int = simpson(f, a, b)
    new.int = simpson(f, a, c) + simpson(f, c, b)
    if abs( old.int - new.int ) < error then
        integral = (16*new.int - old.int) /15
    else
        integral = integral(f, a, c, error/2) +
                    integral(f, c, b, error/2)
    end if
end function

```

Clearly, there is no obligation to use Simpson's rule on the sub-intervals: any favorite algorithm will do.

To translate the above into FORTH, we decompose into smaller parts. The name of the function representing the integrand (actually its execution address or **cfa**) is placed on the stack by **USE(**, as in Ch. 8 §1.3.2 above. Thence it can be saved in a local variable —either on the **rstack** or in a **VAR** or **VARIABLE** that can be **BEHEADED**— so the corresponding phrases

R@ EXECUTE	\	rstack
name EXECUTE	\	VAR
name EXECUTE@	\	VARIABLE

evaluate the integrand. Clearly the limits and error (or tolerance) must be placed on a stack of some sort, so the function can call itself. One simple possibility is to put the arguments on the 87stack itself. (Of course we then need a software fstack manager to extend the limited 87stack into memory, as discussed in Ch. 4 §7.) Alternatively, we could use the intelligent fstack (**ifstack**) discussed in Ch. 5 §2.5. We thus imagine the fstack to be as deep as necessary.

The program then takes the form¹³ shown on p. 171 below.

Note that in going from the pseudocode to FORTH we replaced **)INTEGRAL** by **RECURSE** inside the word **)INTEGRAL**. The need for this arises from an ideo-syncreny of FORTH: normally words do not refer to themselves, hence a word being defined is hidden from the dictionary search mechanism (compiler) until the final ; is reached. The word **RECURSE** unhides the current name, and compiles its **cfa** in the proper spot¹⁴.

- 13. For generality we do not specify the integration rule for sub-intervals, but factor it into its own word. If we want to change the rule, we then need redefine but one component (actually two, since the Richardson extrapolation —see Appendix 8.C— needs to be changed also).
- 14. We may define **RECURSE** (in reverse order) as


```
:RECURSE ?COMP LAST-CFA ;
: ?COMP STATE @ 0= ABORT"Compile only!";
: LAST-CFA LATEST PFA CFA ; IMMEDIATE
\ These definitions are appropriate for HS/FORTH
```

Note that **RECURSE** is called **MYSELF** in some dialects.

```

: USE( [COMPILE] ) CFA LITERAL ;
IMMEDIATE

: f(x) ( : cfa -- cfa ) DUP EXECUTE ;

: )Integral ( f: a b -- l )
  \ uses trapezoidal rule
  XDUP FR- F2/ ( 87: -- a b [b-a]/2 )
  F-ROT f(x) FSWAP f(x) F+ F* ;
: )Richardson \ R-extrap. for trap. rule
  3 S>F F/ F+ ; ( 87: l' l -- l" )

DVARIABLE ERR \ place to store err
CREATE OLD.I 10 ALLOT
  \ place to store [a,b]

: )INTEGRAL ( : adr-- 87: a b err -- l )
ERR R32!
XDUP )Integral ( 87: -- a b l0 )
OLD.I R80!
XDUP F+ F2/ ( 87: -- a b c = [a+b]/2 )
FUNDER FSWAP ( 87: -- a c c b )
XDUP )Integral ( 87: -- a c c b l1 )
F4P F4P ( 87: -- a c c b l1 a )
)Integral F+ ( 87: -- a c c b l1+l2 )
FDUP OLD.I R80@ F-
FDUP FABS ERR R32@ F<
IF )Richardson
  FPLUCK FPLUCK FPLUCK FPLUCK
ELSE FDROP FDROP ( 87: -- a c c b )
  ERR R32@ F2/
  F-ROT F2P ( 87: -- a c err/2 c b err/2 )
  RECURSE ( 87: -- a c err/2 l[c,b] )
  F3R F3R F3R RECURSE F+
THEN DROP ;

```

§§5-2 Disadvantages of recursion in adaptive integration

The main advantage of the recursive adaptive integration algorithm is its ease of programming. As we shall see, the recursive program is much shorter than the non-recursive one. For any reasonable integrand, the fstack (or ifstack) depth grows only as the square of the logarithm of the finest subdivision, hence never gets too large.

However, recursion has several disadvantages when applied to numerical quadrature:

- The recursive program evaluates the function more times than necessary.
- It would be hard to nest the function **)INTEGRAL** for multi-dimensional integrals.

Several solutions to these problems suggest themselves:

- The best, as we shall see, is to eliminate recursion from the algorithm.
- We can reduce the number of function evaluations with a more precise quadrature formula on the sub-intervals.

- We can use “open” formulas like Gauss-Legendre, that omit the endpoints (see Appendix 8.1).

§§5-3 Adaptive Integration without recursion

The chief reason to write a non-recursive program is to avoid any repeated evaluation of the integrand. That is, the optimum is not only the smallest number of points x_n in $[A, B]$ consistent with the desired precision, but to evaluate $f(x)$ once only at each x_n . This will be worthwhile when the integrand $f(x)$ is costly to evaluate.

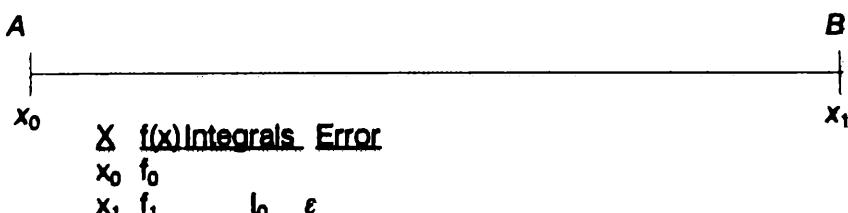
To minimize evaluations of $f(x)$, we shall have to save values $f(x_n)$ that can be re-used as we subdivide the intervals.

The best place to store the $f(x_n)$ ’s is some kind of stack or array. Moreover, to make sure that a value of $f(x)$ computed at one mesh size is usable at all smaller meshes, we must subdivide into two equal sub-intervals; and the points x_n must be equally spaced and include the end-points. Gaussian quadrature is thus out of the question since it invariably (because of the non-uniform spacings of the points) demands that previously computed $f(x_n)$ ’s are thrown away because they cannot be re-used.

The simplest quadrature formula that satisfies these criteria is the **trapezoidal rule** (see Appendix 8.2). This is the formula used in the following program.

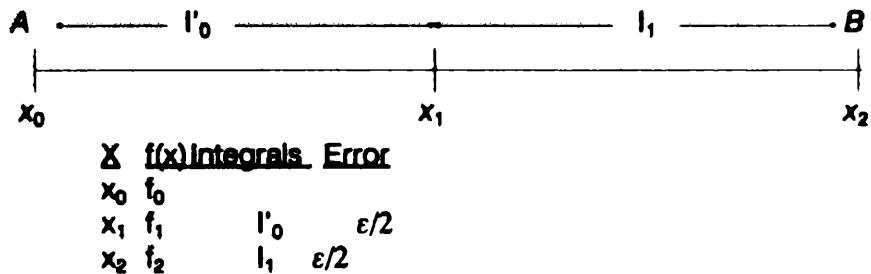
To clarify what we are going to do, let us visualize the interval of integration, and mark the mesh points (where we evaluate $f(x)$) with + :

Step 1: $N = 1$



We now save (temporarily) I_0 and divide the interval in two, computing I'_0 and I_1 on the halves, as shown. This will be one fundamental operation in the algorithm.

Step 2: $N = N + 1 = 2$



We next compare $I'_0 + I_1$ with I_0 . The results can be expressed as a branch in a flow diagram, shown below.

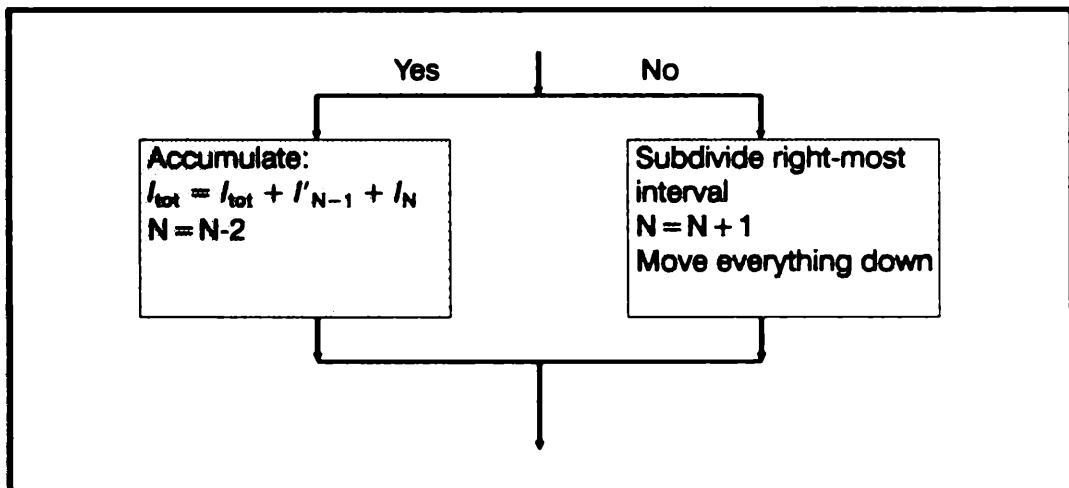
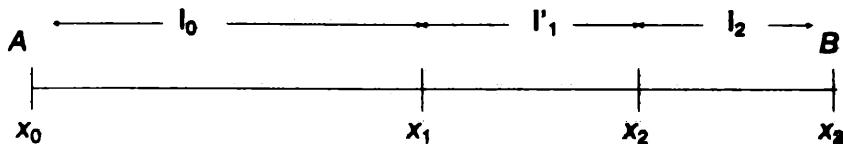


Fig. 8-3 SUBDIVIDE branch in adaptive integration

If the two integrals disagree, we subdivide again, as in Step 3 and Step 4 below:



Step 3: $N = N + 1 = 3$



X f(x) Integrals Error

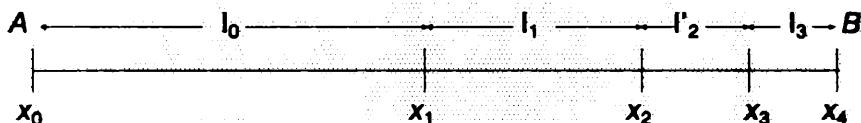
$x_0 f_0$

$x_1 f_1 \quad I_0 \quad \epsilon/2$

$x_2 f_2 \quad I'_1 \quad \epsilon/4$

$x_3 f_3 \quad I_2 \quad \epsilon/4$

Step 4: $N = N + 1 = 4$



X f(x) Integrals Error

$x_0 f_0$

$x_1 f_1 \quad I_0 \quad \epsilon/2$

$x_2 f_2 \quad I_1 \quad \epsilon/4$

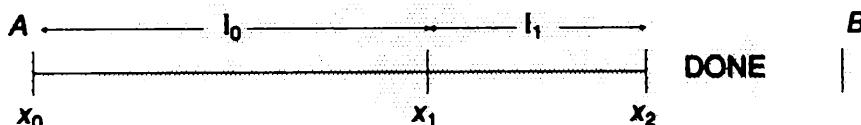
$x_3 f_3 \quad I'_2 \quad \epsilon/8$

$x_4 f_4 \quad I_3 \quad \epsilon/8$

Now suppose the last two sub-integrals ($I_3 + I'_2$) in step 4 agreed with their predecessor (I_2); we then accumulate the part computed so far, and begin again with the (leftward) remainder of the interval, as in Step 5:

Step 5: $N = N - 2 = 2$

$$I = I + (I'_2 + I_3) + (I'_2 + I_3 - I_2)/3$$



X f(x) Integrals Error

$x_0 f_0$

$x_1 f_1 \quad I_0 \quad \epsilon/2$

The flow diagram of the algorithm now looks like Fig. 8-4 below.

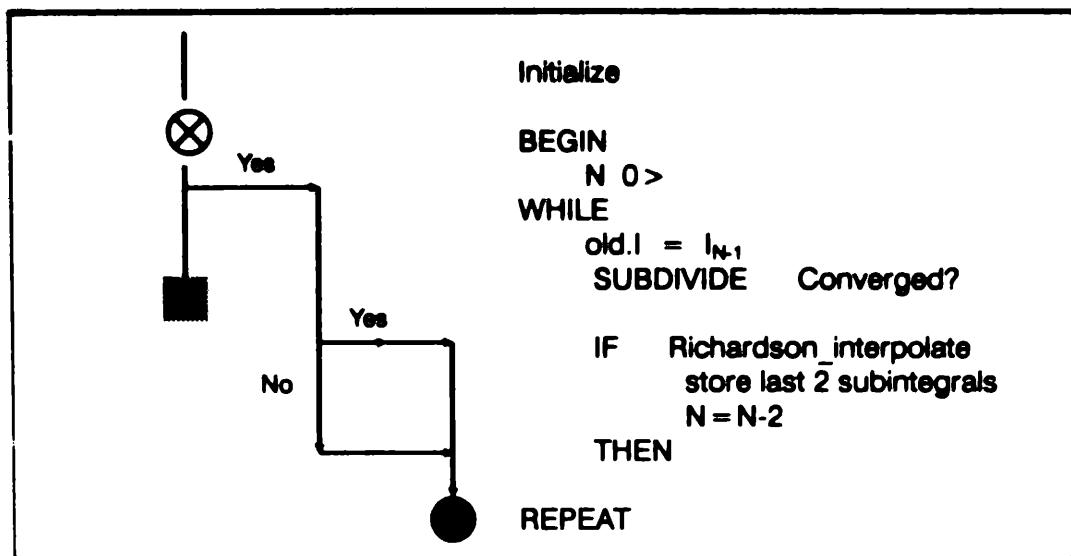


Fig. 8-4 Non-recursive adaptive quadrature

and the resulting FORTH program is ¹⁵:

-
15. We use the generalized arrays of Ch. 5 §3.4; A, B and E are fp #'s on the fstack, TYPE is the data-type of $f(x)$ and INTEGRAL.

```

\ COPYRIGHT 1991 JULIAN V. NOBLE
TASK INTEGRAL
FIND CP@L 0= ?( FLOAD COMPLEX )
\ define data-type tokens if not already
FIND REAL*4 0= ?((((
    0 CONSTANT REAL*4
    1 CONSTANT REAL*8
    2 CONSTANT COMPLEX
    3 CONSTANT DCOMPLEX )))

FIND 1ARRAY 0= ?( FLOAD MATRIX.HSF )
\ function usage
: USE( [COMPILE] ) CFA ; IMMEDIATE

\ BEHEADIng starts here
0 VAR N

: inc.N N 1+ IS N ;
: dec.N N 2- IS N ;

0 VAR type

\ define "stack"
    20 LONG REAL*8 1ARRAY X{
    20 LONG REAL*4 1ARRAY E{
    20 LONG DCOMPLEX 1ARRAY F{
    20 LONG DCOMPLEX 1ARRAY I{

2 DCOMPLEX SCALARS old.I final.I
:)integral ( n -- )\ trapezoidal rule
    X{ OVER } G@L
    X{ OVER 1- } G@L
    F- F2/
    F{ OVER } G@L
    F{ OVER 1- } G@L
    type 2 AND
    IF      X+ FROT X*F
    ELSE    F+ F* THEN
    I{ SWAP 1- } G!L ;
0 VAR f.name
:)f(x) f.name EXECUTE ;

```

```

: INITIALIZE
    IS type \ store type
    type F{ !
    type I{ !\ set types for function
    type 'old.I !
    type 'final.I ! \ and integral(s)
    type 1 AND X{ !
        \ set type for indep. var.
    E{ 0 } G!L \ store error
    X{ 1 } G!L \ store B
    X{ 0 } G!L \ store A
    IS f.name \ ! cfa of f(x)
    X{ 0 } G@L f(x) F{ 0 } G!L
    X{ 1 } G@L f(x) F{ 1 } G!L
    1 IS N
    N )integral
    type 2 AND IF F=0 THEN
    F=0 final.I G!L
    FINIT ;

: E/2 E{ N 1- } G@L F2/ E{ N 1- } G!L ;
: }move.down ( adr n -- )
    } #BYTES >R          ( -- seg off)
    DDUP R@ +
    ( -- s(seg s.off d(seg d.off)
    R> CMOVEV ;

: MOVE.DOWN
    E{ N 1- } move.down
    X{ N } move.down
    F{ N } move.down ;

: new.X ( 87: -- x' )
    X{ N } G@L X{ N 1- } G@L
    F+ F2/ FDUP X{ N } G!L ;
\ cont'd ...

```

```

\ INTEGRAL cont'd
\ debugging code
:GF 1 > IF FSWAP E. THEN E. ;
:F@. DUP>R G@L R> GF. ;
:STACKS CR ."N"
 8 CTAB ."X"
 19 CTAB ."Re[F(X)]"
 31 CTAB ."Im[F(X)]"
 45 CTAB ."Re[I]"
 57 CTAB ."Im[I]"
 71 CTAB ."E"
N 2 + 0 DO CR I .
 3 CTAB X{I} F@.
 16 CTAB F{I} F@.
 42 CTAB I{I} F@.
 65 CTAB E{I} F@.
LOOP
CR 5 SPACES ."old.I = " old.I F@.
5 SPACES ."final.I = " final.I F@. CR ;

CASE: <DEBUG> NEXT .STACKS ;CASE
0 VAR (DEBUG)
: DEBUG-ON 1 IS (DEBUG) 5 #PLACES ! ;
: DEBUG-OFF 0 IS (DEBUG) 7 #PLACES ! ;
: DEBUG (DEBUG) <DEBUG> ;

:SUBDIVIDE
N 19 > ABORT" Too many subdivisions!""
E/2 MOVE.DOWN
I{N 1-} DROP old.I #BYTES CMOVE
  newX f(x) F{N} G!L
N )integral N 1+ )integral ;

```

```

: CONVERGED? (87: -- I[N] + I'[N-1]-I[N-1] -- )
  I{N} G@L I{N 1-} G@L old.I G@L
  type 2 AND
  IF CP CP+ CP DUP CPABS
  ELSE F- F+ FDUP FABS
  THEN
  E{N 1-} G@L F2* F< ;

CASE: g*1 CP*F F* ;CASE
4 S->F 3 S->F F/ FCONSTANT F=4/3

:INTERPOLATE (87: I[N] + I'[N-1]-I[N-1] -- )
  F=4/3 type 2/ g*1
  old.I G@L final.I G@L
  type 2 AND
  IF CP+ CP+
  ELSE F+ F+ THEN
  final.I G!L ;
\ BEHEADing ends here

:)INTEGRAL (87: A B ERR -- I[A,B])
INITIALIZE
BEGIN N 0>
WHILE SUBDIVIDE DEBUG
  CONVERGED? Inc.N
  IF INTERPOLATE dec.N
  ELSE type 2 AND IF FDROP
  THEN FDROP
  THEN
  REPEAT final.I G@L ;
BEHEAD" N INTERPOLATE \optional
\ USE( F.name % A % B % E type )INTEGRAL

```

The nonrecursive program obviously requires *much* more code than the recursive version. This is the chief disadvantage of a nonrecursive method¹⁶.

16. The memory usage is about the same: the recursive method pushes limits, etc. onto the fstack.

§§5-4 Example of)INTEGRAL IN USE

The debugging code ("DEBUG-ON") lets us track the execution of the program by exhibiting the simulated stacks. Here is an example, $\int_1^2 dx \sqrt{x}$:

```
USE( FSQRT % 1. % 2. % 1.E-3 REAL*4 )INTEGRAL E.
```

N	X	F	I	E
0	1.0000E+00	1.0000E+00	5.5618E-01	5.0000E-04
1	1.5000E+00	1.2247E+00	6.5973E-01	5.0000E-04
2	2.0000E+00	1.4142E+00	1.4983E-01	1.2500E-04
	old.I = 1.2071E+00	final.I = 0.0000E+00		

0	1.0000E+00	1.0000E+00	5.5618E-01	5.0000E-04
1	1.5000E+00	1.2247E+00	3.1845E-01	2.5000E-04
2	1.7500E+00	1.3228E+00	3.4213E-01	2.5000E-04
3	2.0000E+00	1.4142E+00	1.7396E-01	1.2500E-04
	old.I = 6.5873E-01	final.I = 0.0000E+00		

0	1.0000E+00	1.0000E+00	5.5618E-01	5.0000E-04
1	1.5000E+00	1.2247E+00	3.1845E-01	2.5000E-04
2	1.7500E+00	1.3228E+00	1.6826E-01	1.2500E-04
3	1.8750E+00	1.3693E+00	1.7396E-01	1.2500E-04
4	2.0000E+00	1.4142E+00	0.0000E+00	0.0000E+00
	old.I = 3.4213E-01	final.I = 0.0000E+00		

0	1.0000E+00	1.0000E+00	5.5618E-01	5.0000E-04
1	1.5000E+00	1.2247E+00	1.5621E-01	1.2500E-04
2	1.6250E+00	1.2747E+00	1.6235E-01	1.2500E-04
3	1.7500E+00	1.3228E+00	1.7396E-01	1.2500E-04
	old.I = 3.1845E-01	final.I = 3.4226E-01		

N	X	F	I	E
0	1.0000E+00	1.0000E+00	2.6475E-01	2.5000E-04
1	1.2500E+00	1.1180E+00	2.9284E-01	2.5000E-04
2	1.5000E+00	1.2247E+00	1.6235E-01	1.2500E-04
	old.I = 5.5618E-01	final.I = 6.8087E-01		

0	1.0000E+00	1.0000E+00	2.6475E-01	2.5000E-04
1	1.2500E+00	1.1180E+00	1.4316E-01	1.2500E-04
2	1.3750E+00	1.1729E+00	1.4983E-01	1.2500E-04
3	1.5000E+00	1.2247E+00	1.7396E-01	1.2500E-04
	old.I = 2.9284E-01	final.I = 8.8087E-01		

0	1.0000E+00	1.0000E+00	1.2879E-01	1.2500E-04
1	1.1250E+00	1.0606E+00	1.3616E-01	1.2500E-04
2	1.2500E+00	1.1180E+00	1.4983E-01	1.2500E-04
old.I = 2.6475E-01	final.I = 9.5392E-01			
1.2189E+00	ok			

$$\int_1^2 dx \sqrt{x} = \frac{2}{3} (2^{3/2} - 1)$$

Notice that, although \sqrt{x} is perfectly finite at $x = 0$, its first derivative is not. This is not a problem in the above case, because the lower limit is 1.0.

It is an instructive exercise to run the above example with the limits (0.0, 1.0). The adaptive routine spends many iterations approaching $x = 0$ (25 in the range [0., 0.0625] vs. 25 in the range [0.0625, 1.0]). This is a concrete example of how an adaptive routine will unerringly locate the (integrable) singularities of a function by spending lots of time near them. The best answer to this problem is to separate out the bad parts of a function by hand, if possible, and integrate them by some other algorithm that takes the singularities into account. By the same token, one should always integrate up to, but not through, a discontinuity in $f(x)$.

§§6 Adaptive Integration in the Argand plane

We often want to evaluate the complex integral

$$I = \oint_{\Gamma} f(z) dz \quad (16)$$

where Γ is a contour (simple, closed, piecewise-continuous curve)¹⁷ in the complex z -plane, and $f(z)$ is an analytic¹⁷ function of z .

The easiest way to evaluate 16 is to parameterize z as a function of a real variable t ; as t runs from A to B, $z(t)$ traces out the contour. For example, the parameterization

$$z(t) = z_0 + R \cos(t) + iR \sin(t), \quad 0 \leq t \leq 2\pi \quad (17)$$

traces out a (closed) circle of radius R centered at $z = z_0$.

We assume that the derivative $\dot{z}(t) \equiv \frac{dz}{dt}$ can be defined; then the integral 16 can be re-written as one over a real interval, with a complex integrand:

$$I = \int_A^B \dot{z}(t) f(z(t)) dt \quad (18)$$

Now our previously defined adaptive function)INTEGRAL can be applied directly, with F.name the name of a *complex* function

$$g(t) = \dot{z}(t) f(z(t)), \quad (19)$$

of the *real* variable t .

Here is an example of complex integration: we integrate the function $f(z) = e^{1/z}$ around the unit circle in the counter-clockwise (positive) direction.

7. “Analytic” means the ordinary derivative $df(z)/dz$ exists. Consult any good text on the theory of functions of a complex variable.

The calculus of residues (Cauchy's theorem) gives

$$\oint_{|z|=1} dz e^{1/z} = 2\pi i \quad (20)$$

We parameterize the unit circle as $z(t) = \cos(2\pi t) + i \sin(2\pi t)$, hence $\dot{z}(t) = 2\pi i z(t)$, and we might as well evaluate

$$\int_0^1 dt z(t) e^{1/z(t)} \equiv 1. \quad (21)$$

For reasons of space, we exhibit only the first and last few iterations:

```
FIND FSINCOS 0 = ?( FLOAD TRIG )
: Z(T) F=PI F* F2* FSINCOS ;      (87: t -- )
: XEXP FSINCOS FROT FEXP X*X ;
    (87: x y -- e^x cos[y] e^x sin[y] )
: G(T) Z(T) XDUP 1/X XEXP X* ;
DEBUG-ON
USE( G(T) % 0 % 1 % 1.E-2 COMPLEX )INTEGRAL X.
```

N	X	F	I	E	N	X	F	I	E				
0	0.0000	2.7182	0.0000	.58760	0.0000	.0049999	0	0.0000	2.7182	0.0000	.58760	0.0000	.0049999
1	.50000	-.36787	-0.0000	.58760	0.0000	.0049999	1	.50000	-.36787	-0.0000	.059198	-.067537	.0024999
2	1.0000	2.7182	0.0000	.00000	.0000000		2	.75000	.84147	-.54030	.17886	-.043682	.0012499
	old.I = 2.7182 0.0000	final.I = 0.0000 0.0000					3	.87500	2.0219	-.15862	.14190	-.005745	.00062499
							4	.93750	2.5189	-.025229	.081022	-.0004467	.00031249
							5	.96875	2.6665	-.0033577	.08414	-.0000525	.00031249
							6	1.0000	2.7182	0.0000	.000000	.000000000	

N	X	F	I	E	N	X	F	I	E				
0	0.0000	2.7182	0.0000	.58760	0.0000	.0049999	0	0.0000	2.7182	0.0000	.58760	0.0000	.0049999
1	.50000	-.36787	-0.0000	.059198	-.067537	.0024999	1	.50000	-.36787	-0.0000	.059198	-.067537	.0024999
2	.75000	.84147	-.54030	.44496	-.067537	.0024999	2	.75000	.84147	-.54030	.17886	-.043682	.0012499
3	1.0000	2.7182	0.0000	.00000	.0000000		3	.87500	2.0219	-.15862	.14190	-.005745	.00062499
	old.I = .58760 0.0000	final.I = 0.0000 0.0000					4	.93750	2.5189	-.025229	.08102	-.0004467	.00031249
							5	.96875	2.6665	-.0033577	.04197	-.0000298	.00015624
							6	.98437	2.7052	-.000428	.042371	-.0000033	.00015624
							7	1.0000	2.7182	0.0000	.00000	0.0000	
							old.I = .084137 -.000062465	final.I = 0.0000 0.0000					

N	X	F	I	E	N	X	F	I	E				
0	0.0000	2.7182	0.0000	.58760	0.0000	.0049999	0	0.0000	2.7182	0.0000	.58760	0.0000	.0049999
1	.50000	-.36787	-0.0000	.059198	-.067537	.0024999	1	.50000	-.36787	-0.0000	.059198	-.067537	.0024999
2	.75000	.84147	-.54030	.17886	-.043682	.0012499	2	.75000	.84147	-.54030	.17886	-.043682	.0012499
3	.87500	2.0219	-.15862	.29626	-.009914	.0012499	3	.87500	2.0219	-.15862	.14190	-.005745	.00062499
4	1.0000	2.7182	0.0000	.00000	.0000000		4	.93750	2.5189	-.025229	.040020	-.0002833	.00015624
	old.I = .44496 -.067537	final.I = 0.0000 0.0000					5	.96875	2.6665	-.0033577	.04197	-.0000298	.00015624
							6	.98437	2.7052	-.000428	.042371	-.0000033	.00015624
							7	1.0000	2.7182	0.0000	.00000	0.0000	
							old.I = .084137 -.000062465	final.I = 0.0000 0.0000					

N	X	F	I	E	N	X	F	I	E				
0	0.0000	2.7182	0.0000	.58760	0.0000	.0049999	0	0.0000	2.7182	0.0000	.58760	0.0000	.0049999
1	.50000	-.36787	-0.0000	.059198	-.067537	.0024999	1	.50000	-.36787	-0.0000	.059198	-.067537	.0024999
2	.75000	.84147	-.54030	.17886	-.043682	.0012499	2	.75000	.84147	-.54030	.17886	-.043682	.0012499
3	.87500	2.0219	-.15862	.14190	-.005745	.00062499	3	.87500	2.0219	-.15862	.14190	-.005745	.00062499
4	.93750	2.5189	-.025229	.16368	-.000788	.00062499	4	.93750	2.5189	-.025229	.040020	-.0002833	.00015624
5	1.0000	2.7182	0.0000	.00000	.0000000		5	.96875	2.6665	-.0033577	.04197	-.0000298	.00015624
	old.I = .29626 -.0099138	final.I = 0.0000 0.0000					6	.98875	2.8685	-.003358	.042371	-.0000033	.00015624
							7	1.0000	2.7182	0.0000	.00000	0.0000	
							old.I = .081022 -.0044667	final.I = .084404 -.0000265					

N	X	F	I	E
0	0.0000	2.7182	0.0000	.29628 .0099138 .0012499
1	.12500	2.0219	.15862	.10763 .016479 .00062499
2	.18750	1.4180	.38873	.036711 .013045 .00031249
3	.21875	1.1224	.46620	.030888 .015726 .00031249
4	.25000	.94147	.54030	.000053670 .0079876 .00015624

old.I = .070942 .026407 final.I = .51343 -.050789N

N	X	F	I	E
0	0.0000	2.7182	0.0000	.29628 .0099138 .0012499
1	.12500	2.0219	.15862	.058518 .0088558 .00031249
2	.18750	1.7232	.26094	.049099 .0098388 .00031249
3	.18750	1.4180	.38873	.030888 .015726 .00031249

old.I = .10753 .016479 final.I = .58374 -.021892

N	X	F	I	E
0	0.0000	2.7182	0.0000	.16386 .00078842 .00062499
1	.062500	2.5189	.025229	.14180 .0057453 .00062499
2	.12500	2.0219	.15862	.049099 .0098388 .00031249

old.I = .29628 .0099138 final.I = .89139 -.0055268

N	X	F	I	E
0	0.0000	2.7182	0.0000	.16386 .00078842 .00062499
1	.062500	2.5189	.025229	.075223 .0015953 .00031249
2	.083749	2.2954	.078875	.067457 .0036796 .00031249
3	.12500	2.0219	.15862	.030888 .015726 .00031249

old.I = .14180 .0057453 final.I = .89139 -.0055268

N	X	F	I	E
0	0.0000	2.7182	0.0000	.16386 .00078842 .00062499
1	.062500	2.5189	.025229	.075223 .0015953 .00031249
2	.093749	2.2954	.078875	.034633 .0014942 .00015624
3	.10937	2.1632	.11439	.032886 .0021329 .00015624
4	.12500	2.0219	.15862	.0000537 .0079876 .00015624

old.I = .067457 .0036796 final.I = .89139 -.0055268

N	X	F	I	E
0	0.0000	2.7182	0.0000	.16386 .00078842 .00062499
1	.062500	2.5189	.025229	.038546 .00088464 .00015624
2	.078125	2.4180	.047044	.036800 .00088812 .00015624
3	.093749	2.2954	.078875	.032886 .0021329 .00015624

old.I = .075223 .0015953 final.I = .75894 -.0019171

N	X	F	I	E
0	0.0000	2.7182	0.0000	.084137 .000052465 .00031249
1	.031250	2.68685	.0033577	.081022 .00044687 .00031249
2	.083749	2.5189	.025229	.036800 .00088812 .00015624

old.I = .16386 .00078842 final.I = .83433 -.00040523

N	X	F	I	E
0	0.0000	2.7182	0.0000	.084137 .000052465 .00031249
1	.031250	2.68685	.0033577	.041173 .00011247 .00015624
2	.046875	2.6036	.011038	.040020 .00028334 .00015624
3	.083749	2.5189	.025229	.032886 .0021329 .00015624

old.I = .081022 .00044687 final.I = .83433 -.00040523

N	X	F	I	E
0	0.0000	2.7182	0.0000	.042371 .000003331 .00015624
1	.015625	2.7062	.00042642	.041886 .0000296 .000 5624
2	.031250	2.68685	.0033577	.040020 .00028334 .00015624

old.I = .084137 .000052465 final.I = .91558 -.000026373

.98889 .000000001 ok

Note:
answer = 1

§2 Fitting functions to data

One of the most important applications of numerical analysis is the representation of numerical data in functional form. This includes fitting, smoothing, filtering, interpolating, etc.

A typical example is the problem of table lookup: a program requires values of some mathematical function – $\sin(x)$, say – for arbitrary values of x . The function is moderately or extremely time-consuming to compute directly. According to the Intel tim-

ings for the 80x87 chip, this operation should take about 8 times longer than a floating point multiply. In some real-time applications this may be too slow.

There are several ways to speed up the computation of a function. They are all based on compact representations of the function — either in tabular form or as coefficients of functions that are faster to evaluate. For example, we might represent $\sin(x)$ by a simple polynomial¹⁸

$$\sin(x) \approx x (0.994108 - 0.147202x), \quad (22)$$

accurate to better than 1% over the range $-\frac{\pi}{2} \leq x \leq \frac{\pi}{2}$, the

requires but 3 multiplications and an addition to evaluate. This would be twice as fast as calculating $\sin(x)$ on the 80x87 chip.¹⁹

To achieve substantially greater speed requires table look-up. To locate data in an ordered table, we might employ binary search: that is, look at the x -value halfway down the table and see if the desired value is greater or less than that. On the average, $\log_2(N)$ comparisons are required, where N is the length of the table. For a table with 1% precision, we might need 128 entries, i.e. seven comparisons.

Binary search is unacceptably slow — is there a faster method? In fact, assuming an ordered table of equally-spaced abscissae the fastest way to locate the desired x -value is **hashing**, a method for computing the address rather than finding it using comparisons. Suppose, as before, we need 1% accuracy, i.e. a 128-point table with x in the range $[0, \pi/2]$. To look up a value, we multiply x by $256/\pi \cong 81.5$, truncate to an integer and quadruple it to get a (4-byte) floating point address. These operations — including fetch to the 87stack — take about 1.5-2 fp multiply times, hence the speedup is 4-fold.

The speedup factor does not seem like much, especially for a function such as $\sin(x)$ that is built into the fast co-processor. However, if we were speaking of a function that is considerably slower to evaluate (for example one requiring evaluation of an integral or solution of a differential equation) hashed table lookup with interpolation can be several orders of magnitude faster than direct evaluation.

We now consider how to represent data by mathematical functions. This can be useful in several contexts:

- The theoretical form of the function, but with unknown parameters, may be known. One might like to determine the parameters from the data. For example, one might have a lot of data on pendulums: their periods, masses, dimensions, etc. The period of a pendulum is given, theoretically, by

$$\tau = \left(\frac{2\pi L}{g} \right)^{1/2} f \left(\frac{L}{r}, \frac{m_{\text{bob}}}{m_{\text{string}}}, \dots \right) \quad (23)$$

where L is the length of the string, g the acceleration of gravity, and f is some function of ratios of typical lengths, masses and other factors in the problem. In order to determine g accurately, one generally fits a function of all the measured factors, and tries to minimize its deviation from the measured periods. That is, one might try

$$\tau_n = \left(\frac{2\pi L_n}{g} \right)^{1/2} \left[1 + \alpha \frac{r_n}{L_n} + \beta \left(\frac{m_{\text{bob}}}{m_{\text{string}}} \right)_n + \dots \right] \quad (24)$$

for the n 'th set of observations, with g, α, β, \dots the unknown parameters to be determined.

- Sometimes one knows that a phenomenon is basically smoothly varying; so that the wiggles and deviations in observations are noise or otherwise uninteresting. How can we filter out the noise without losing the significant part of the data? Several methods have been developed for this purpose, based on the same principle: the data are represented as a sum of functions from a complete set of functions, with unknown coefficients. That is, if $\varphi_m(x)$ are the functions, we say (y_n are the data)

$$y_n = \sum_{m=0}^{\infty} c_m \varphi_m(x_n) \quad (25)$$

Such representations are theoretically possible under general conditions. Then to filter we keep only a finite sum, retaining the first N (usually simplest and smoothest) functions from the set. An example of a complete set is monomials, $\varphi_m(x) = x^m$. Another is sinusoidal (trigonometric) functions,
 $\sin(2\pi mx)$, $\cos(2\pi mx)$, $0 \leq x \leq 1$,

used in Fourier-series representation. Gram polynomials, discussed below, comprise a third useful complete set.

The representation in Eq. 25 is called linear because the unknown coefficients c_m appear to their first power. Thus, if all the data were to double, we see immediately that the c_m 's would have to be multiplied by the same factor, 2. Sometimes, as in the example of the measurement of g above, the unknown parameters appear in more complicated fashion. The problem of fitting with these more general functional forms is called nonlinear for obvious reasons. The simplex algorithm of Ch. 8 §2.3 below is an example of a nonlinear fitting procedure.

We are now going to write programs to fit both linear and nonlinear functions to data. The first and conceptually simplest of these is the Fourier transform, namely representing a function as a sum of sines and cosines.

§§1 Fast Fourier transform

What is a Fourier transform? Suppose we have a function that is periodic on the interval $0 \leq x \leq 2\pi$:

$$f(x + 2\pi) = f(x) ;$$

Then under fairly general conditions the function can be expressed in the form

$$f(x) = a_0 + \sum_{n=1}^{\infty} (a_n \cos(nx) + b_n \sin(nx)) \quad (26)$$

Another way to write Eq. 26 is

$$f(x) = \sum_{-\infty}^{+\infty} c_n e^{inx}. \quad (27)$$

In either way of writing, the c_n are called **Fourier coefficients** of the function $f(x)$. Looking, e.g., at Eq. 27, we see that the orthogonality of the sinusoidal functions leads to the expression

$$c_n = \frac{1}{2\pi} \int_0^{2\pi} f(x) e^{-inx} dx. \quad (28)$$

Evaluating Eq. 28 numerically requires – for given n – at least $2n$ points²⁰. Naively, for each $n=0$ to $N-1$ we have to do a sum

$$c_n \approx \sum_{k=1}^{2N} f_k e^{-2\pi i n k / N}$$

which means carrying out $2N^2$ complex multiplications.

The **fast Fourier transform (FFT)** was discovered by Runge and König, rediscovered by Danielson and Lanczos and ~~re~~-rediscovered by Cooley and Tukey²¹. The FFT algorithm can be expressed as three steps:

- Discretize the interval, i.e. evaluate $f(x)$ only for

$$x_k = 2\pi \frac{k}{N}, \quad 0 \leq k \leq N-1.$$

Call $f(x_k) \equiv f_k$.

- Express the Fourier coefficients as

$$c_n = \sum_{k=0}^{N-1} f_k e^{-2\pi i n k / N}. \quad (29)$$

- With $w_n = e^{-2\pi i n / N}$, Eq. 29 is an $N-1$ 'st degree polynomial in w_n . We evaluate the polynomial using a fast algorithm.

20. to prevent aliasing.

21. See, e.g., D.E. Knuth, *The Art of Computer Programming*, v. 2 (Addison-Wesley Publishing Co., Reading, MA, 1981) p. 642.

To evaluate rapidly the polynomial

$$c_n = P_N(w_n) \equiv \sum_{k=0}^{N-1} f_k(w_n)^k$$

we divide it into two polynomials of order $N/2$, dividing each of those in two, etc. This procedure is efficient only for $N = 2^v$, with v an integer, so this is the case we attack.

How does dividing a polynomial in two help us? If we segregate the odd from the even powers, we have, symbolically,

$$P_N(w) = E_{N/2}(w^2) + w O_{N/2}(w^2). \quad (30)$$

Suppose the time to evaluate $P_N(w)$ is T_N . Then, clearly,

$$T_N = \lambda + 2T_{N/2} \quad (31)$$

where λ is the time to segregate the coefficients into odd and even, plus the time for 2 multiplications and a division. The solution of Eq. 31 is $\lambda(N-1)$. That is, it takes $O(N)$ time to evaluate a polynomial.

However, the discreteness of the Fourier transform helps us here. The reason is this: to evaluate the transform, we have to evaluate $P_N(w_n)$ for N values of w_n . But w_n^2 takes on only $N/2$ values as n takes on N values. Thus to evaluate the Fourier transform for all N values of n , we can evaluate the two polynomials of order $N/2$ for half as many points.

Suppose we evaluated the polynomials the old-fashioned way: it would take $2(N/2) = N$ multiplications to do both, but we need do this only $N/2$ times, and N more (to combine them) so we have $N^2/2 + N$ rather than N^2 . We have gained a factor 2. Obviously it pays to repeat the procedure, dividing each of the sub-polynomials in two again, until only monomials are left.

Symbolically, the number of multiplications needed to evaluate a polynomial for N (discrete) values of w is

$$\tau_N = N\lambda + 2\tau_{N/2} \quad (32)$$

whose solution is

$$\tau_N = \lambda N \log_2(N) . \quad (33)$$

Although the FFT algorithm can be programmed recursively, it almost never is. To see why, imagine how the coefficients would be re-shuffled by Eq. 30: we work out the case for 16 coefficients, exhibiting them in Table 8-1 below, writing only the indices:

Table 8-1 Bit-reversal for re-ordering discrete data

Start	Step 1	Step 2	Step 3	Bin_0	Bin_3
0	0	0	0	0000	0000
1	2	4	8	0001	1000
2	4	8	4	0010	0100
3	6	12	12	0011	1100
4	8	2	2	0100	0010
5	10	6	10	0101	1010
6	12	10	6	0110	0110
7	14	14	14	0111	1110
8	1	1	1	1000	0001
9	3	5	9	1001	1001
10	5	9	5	1010	0101
11	7	13	13	1011	1101
12	9	3	3	1100	0011
13	11	7	11	1101	1011
14	13	11	7	1110	0111
15	15	15	15	1111	1111

The crucial columns are “Start” and “Step 3”. Unfortunately, they are written in decimal notation, which conceals a fact that becomes glaringly obvious in binary notation. So we re-write them in binary in the columns Bin_0 and Bin_3 – and see that the final order can be obtained from the initial order simply by reversing the order of the bits, from left to right!

A standard FORTRAN program for complex FFT is shown below. We shall simply translate the FORTRAN into FORTH as expeditiously as possible, using some of FORTH’s simplifications.

One such improvement is a word to reverse the bits in a given integer. Note how clumsily this was done in the FORTRAN

```

SUBROUTINE FOUR1(DATA, NN, ISIGN)
C
C from Press, et al., Numerical Recipes, ibid., p. 304.
C
C ISIGN DETERMINES WHETHER THE FFT
C IS FORWARD OR BACKWARD
C
C DATA IS THE (COMPLEX) ARRAY OF DISCRETE INPUT
C COMPLEX W, WP, TEMP, DATA(N)
C REAL*8 THETA
C
J=0
DO 11 I=0,N-1      \ begin bit.reversal
IF (J.GT.I) THEN
  TEMP = DATA(J)
  DATA(J) = DATA(I)
  DATA(I) = TEMP
ENDIF
M=N/2
1  IF ((M.GE.1).AND.(J.GT.M)) THEN
  J=J-M
  M=M/2
  GO TO 1
ENDIF
J=J+M
11 CONTINUE      \ end bit.reversal

```

2 MMAX = 1 \ begin Danielson-Lanczos section
 IF (N.GT.MMAX) THEN
 ISTEP = 2^MMAX \ executed lg(N) times
 \ init trig recurrence

 THETA = 3.14159265358979D0/(ISIGN*MMAX)
 WP = CEXP(THETA)
 W = DCMLPX(1.0D0,0.0D0)
 DO 13 M = 1,MMAX,2 \ outer loop
 DO 12 I = M,N,ISTEP \ inner loop
 J = I + MMAX \ total = N times
 TEMP = DATA(J)*W
 DATA(J) = DATA(I)-TEMP
 DATA(I) = DATA(I) + TEMP
 12 CONTINUE \ end inner loop
 C
 W = W*WP \ trig recurrence
 C
 13 CONTINUE \ end outer loop
 MMAX = ISTEP
 GO TO 2
ENDIF \ end Danielson-Lanczos section
RETURN

program. Since practically every microprocessor permits right-shifting a register one bit at a time and feeding the overflow into another register from the right, **B.R** can be programmed easily in machine code for speed. Our fast bit-reversal procedure **B.R** may be represented pictorially as in Fig. 8-5 below.

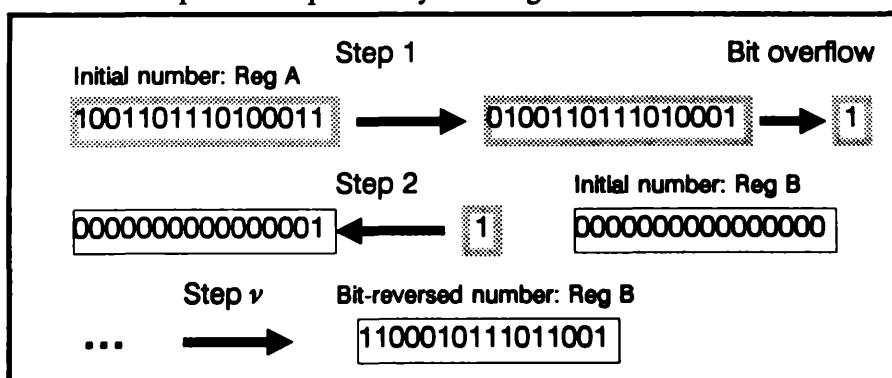


Fig. 8-5 Pictorial representation of bit-reversal

Bit-reversal can be accomplished in high-level FORTH via

```
: B.R      (n .. n')      \ reverse order of bits
    0 SWAP  (.. 0 n)      \ set up stack
    N.BITS 0 DO
        DUP 1 AND          \ pick out 1's bit
        ROT 2* +
        SWAP 2/             \ left shift 1, add 1's bit
    LOOP DROP;            \ right-shift n
```

| Note: **N.BITS** is a VAR, previously set to $\nu = \log_2(N)$

We will use **B.R** to re-order the actual data array (even though this is slightly more time-consuming than setting up a list of scrambled pointers, leaving the data alone). We forego indirection for two reasons: first, we have to divide by N (N steps) when inverse-transforming, so we might as well combine this with bit-reversal; second, there are N steps in rearranging and dividing by N the input vector, whereas the FFT itself takes $N\log_2(N)$ steps, i.e. the execution time for the preliminary N steps is unimportant.

Now, how do we go about evaluating the sub-polynomials to get the answer? First, let us write the polynomials (for our case $N = 16$) corresponding to taking the (bit-reversed) addresses off the stack in succession, as in Fig. 8-6 below.

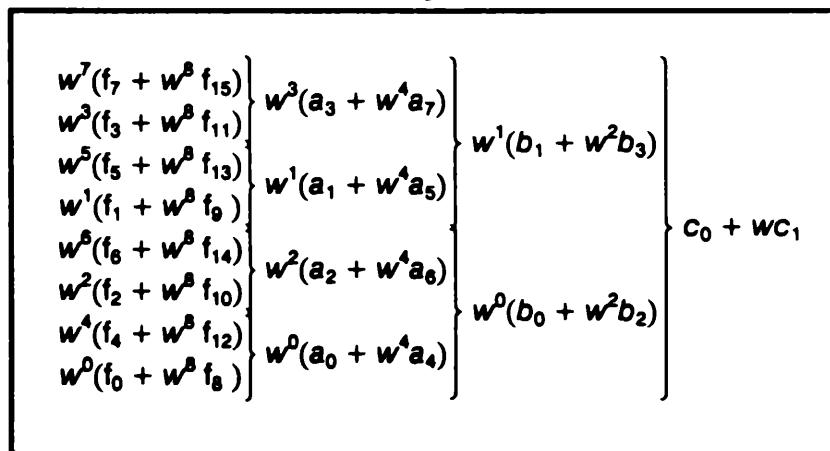


Fig. 8-6 The order of evaluating a 16 pt. FFT

We see that w_n^8 (for $N = 16$) has only two possible values, ± 1 . Thus we must evaluate not 16×8 terms like $f_i + w^8 f_{i+8}$, but only 2×8 . Similarly, we do not need to evaluate 16×4 terms of form $f_i + w^4 f_{i+4}$, but only 4×4 , since there are only 4 possible values of w_n^4 . Thus the total number of multiplications is

$$2 \times 8 + 4 \times 4 + 8 \times 2 + 16 \times 1 = 64 \equiv 16 \log_2 16,$$

as advertised. This is far fewer than $16 \times 16 = 256$, and the ratio improves with N — for example a 1024 point FFT is 100 times faster than a slow FT.

We list the FFT program on page 191 below. Since }FFT transforms a one-dimensional array we retain the curly braces notation introduced in Ch. 5. We want to say something like

V{ n.pts FORWARD }FFT

where **V{** is the name of the (complex) array to be transformed, **n.pts** (a power of 2) is the size of the array, and the flag-setting words **FORWARD** or **INVERSE** determine whether we are taking a FFT or inverting one.

Now we test the program. Table 8-2 on page 192 contains the weekly stock prices of IBM stock, for the year 1983 (the 52 values have been made complex numbers by adding $0i$, and the table padded out to 64 entries (the nearest power of 2) with complex zeros)²². The first two entries (2, 64) are the type and length of the file. (The file reads from left to right.)

We FFT Table 8-2 using the phrase **IBM{ 64 DIRECT }FFT**. The power spectrum of the resulting FFT (Table 8-3) is shown in Fig. 8-7 on page 192 below.

22. This example is taken from the article "FORTH and the Fast Fourier Transform" by Joe Barnhart, *Dr. Dobb's Journal*, September 1984, p. 34.

```

\ Complex Fast Fourier Transform
\ Usage: Vector.name{ N FORWARD ( INVERSE ) }FFT
: TASK FFT
\ check for presence of these extensions and load
: FIND C+ 0= ?( FLOAD COMPLEX )
: FIND 1ARRAY 0= ?( FLOAD MATRIX.HBF )
: FIND FILL 0= ?( FLOAD FILEIO.FTH )
: FIND TRIG 0= ?( FLOAD TRIG )

\ If not there
DECIMAL
-----
\ auxiliary words
: CODE SHR BX 1 SHR. END-CODE
: LG2 (n--lg2[n]) 0 SWAP (..0n) SHR
BEGIN ?DUP 0>
WHILE SHR SWAP 1+ SWAP REPEAT;

\ VAR DIRECTION?
: FORWARD 0 IS DIRECTION?;
: INVERSE -1 IS DIRECTION?;

0 VAR N.BITS           \ some VARS
0 VAR N
0 VAR MMAX
0 VAR I{

: C/N NS->F C/F;
: NORMALIZE DIRECTION?
  IF C/N CPSWAP C/N CPSWAP THEN;
\ end auxiliary words
-----
\ key bit-reversal routines!
0 VAR LR
: LR (n--n)           \ reverse order of bits
  0 SWAP (..0n)         \ set up stack
  N.BITS 0 DO DUP 1 AND \ pick out 1's bit
    ROT 2*+             \ double sum and add 1's bit
    SWAP 2/              \ n-n/2
  LOOP DROP;

: BIT.REVERSE 0 IS LR
  N 0 DO 1 LR IS LR
  LR 1 < NOT      (LR > -1)
  IF !(LR) G@L !{ } G@L NORMALIZE
    !{LR} GIL !{ } GIL THEN
  LOOP;
\ end bit-reversal (N times)

```

```

----- \ main algorithm
CODE C+ 2 FLD. 1 FXCH. 3 FSUBT.
  1 FADD. 1 FXCH. 3 PLD.
  1 FXCH. 4 FSUBT. 1 FADD. 1 FXCH. END-CODE
(87:w2--w2 w+z)

: THETA F=PI MMAX S>F F/ DIRECTION? FSIGN;

CREATE WP 16 ALLOT OKLW
: INIT.TRIG FINIT THETA EXP(IPHI) WP DCPI C=1;

: NEW.W (87:w--w') WP DCPI C*;

0 VAR ISTEP

: DO.INNER.LOOP
  DO MMAX I + IS LR
    CPDUP !{LR} G@L C* !{ } G@L
    CPSWAP C+ !{ } GIL !{LR} GIL
  ISTEP +LOOP;

: }FFT (adr n--) IS N IS !{
  1 IS MMAX
  N LG2 IS N.BITS
  FINIT
  BIT.REVERSE
  BEGIN
    N MMAX >
  WHILE
    INIT.TRIG MMAX 2* IS ISTEP
    MMAX 0 DO
      N 1 DO.INNERLOOP NEW.W
    LOOP
    ISTEP IS MMAX
  REPEAT CPDROP;

: POWER 0 DO !{ } G@L CABS CR I. F. LOOP;
\ power spectrum of FFT          \ end of fft code
-----
\ an example
64 LONG COMPLEX 1ARRAY A{
: INIT.A A($ IBM.EX OPEN INPUT FILL CLOSE INPUT;
INIT.A
A{ 64 DIRECT }FFT
\ end of example
-----
```

How do we know the FFT program actually worked? The simplest method is to inverse-transform the transform, and compare with the input file. The FFT and inverse FFT are given, respectively, in Tables 8-3 and 8-4 on page 193 below. Within roundoff error, Table 8-4 agrees with Table 8-2 on page 192.

Table 8-2 Weekly IBM common stock prices, 1983

2 64			
96.63 0.0	99.13 0.0	94.63 0.0	97.38 0.0
97.38 0.0	96.38 0.0	98.63 0.0	100.38 0.0
102.25 0.0	100.75 0.0	99.88 0.0	102.13 0.0
101.63 0.0	103.88 0.0	110.13 0.0	117.25 0.0
117.00 0.0	117.63 0.0	116.50 0.0	110.63 0.0
113.00 0.0	114.00 0.0	114.25 0.0	121.13 0.0
123.00 0.0	121.00 0.0	121.50 0.0	120.13 0.0
124.38 0.0	120.38 0.0	119.75 0.0	118.50 0.0
122.50 0.0	117.83 0.0	119.75 0.0	122.25 0.0
123.13 0.0	126.63 0.0	126.88 0.0	132.25 0.0
131.75 0.0	127.00 0.0	128.00 0.0	122.25 0.0
126.88 0.0	123.50 0.0	121.00 0.0	117.88 0.0
122.25 0.0	120.88 0.0	123.63 0.0	122.00 0.0
0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0
0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0
0.0 0.0	0.0 0.0	0.0 0.0	0.0 0.0

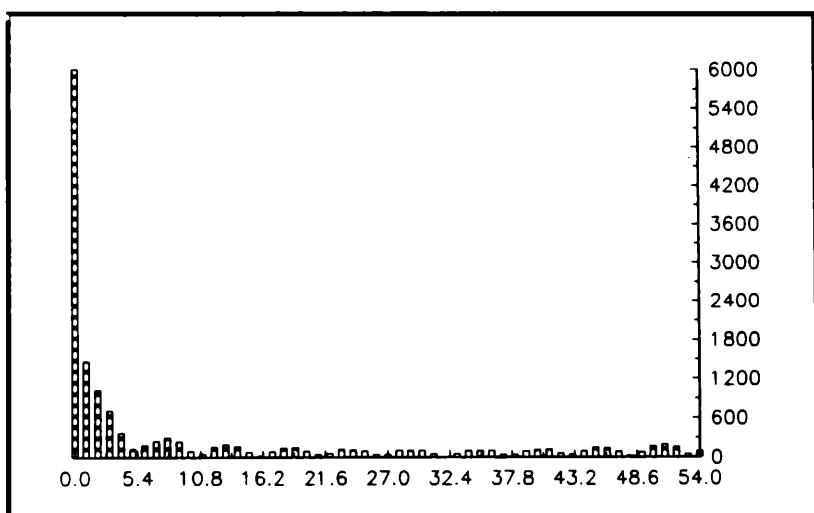


Fig. 8-7 Power spectrum of FFT of 1983 IBM prices (from Table 8-3)

Table 8-3 FFT of IBM weekly stock prices, 1983

0 5089.4589 0.0000000	32 3.1801882 0.0000000	16 7.2500000 -15.170043	48 7.2500000 15.170043
1 -1356.9239 562.94860	33 13.450087 44.888729	17 -27.548482 81.081804	49 69.775169 29.838867
2 -374.28417 956.84282	34 89.178887 32.348035	18 88.084281 117.80709	50 148.10401-64.187637
3 305.86314 634.45819	35 98.788884-25.042837	19 119.02930 71.422482	51 84.502433-165.21282
4 327.40542 177.80118	36 39.788885-63.410888	20 87.788884-24.258874	52 -49.510889-147.46129
5 125.21042 -8.0063304	37 -30.826173-14.841442	21 1.8497881 -37.884341	53 -49.475231 5.4212093
6 -178.28804 39.630813	38 19.283210 29.322804	22-11.327768 52.313049	54 80.279841 33.341926
7 -90.525482 228.27159	39 88.785382 22.908825	23 44.172229 110.77301	55 191.18481-128.15400
8 150.24284 280.91508	40 107.71731-34.418077	24 107.71731 34.418077	56 150.24284-280.91508
9 191.18481 128.15400	41 44.172229-110.77301	25 88.785382 -22.908825	57 -90.525482-228.27159
10 80.279841 -33.341926	42 -11.327768-52.313049	26 19.283210 -29.322804	58 -178.28804-39.630813
11 -49.475231 -5.4212093	43 1.8497881 37.884341	27-30.826173 14.841442	59 125.21042 8.0063304
12 -49.510889 147.46129	44 87.788885 24.258874	28 39.788885 83.410888	60 327.40542-177.80118
13 84.502433 165.21282	45 119.02930-71.422482	29 88.788884 25.042837	61 305.86314-634.45819
14 148.10401 64.187637	46 88.084281-117.80709	30 89.178887 -32.348035	62 -374.28417-956.84282
15 69.775169 -29.838867	47 -27.548482-81.081804	31 13.450087 -44.888729	63 -1356.9239-562.94860

Table 8-4 Reconstructed IBM prices (inverse FFT)

0 98.630 0.0000	32 122.50 0.0000	16 117.00 -0.00000000	48 122.25 .000000000
1 99.129 .000000705	33 117.82 -0.00000245	17 117.62 -0.00000518	49 120.87 .000000132
2 94.829 .000000223	34 119.75 .000000148	18 116.50 -0.00000119	50 123.63 -0.00000052
3 97.379 .000000227	35 122.24 -0.00000379	19 110.62 .000001375	51 121.99 -0.00000944
4 97.380 .000000385	36 123.13 -0.00000385	20 113.00 .000001076	52 .000012884 -0.000001076
5 98.379 -0.00000362	37 126.62 .000000796	21 114.00 .000000975	53 -.000001277 -0.000001434
6 98.630 .000001697	38 128.88 -0.00000851	22 114.25 .000000378	54 -.000002880 -0.00001224
7 100.38 .000002158	39 132.25 -0.00001661	23 121.12 .000000317	55 -.000005683 -0.00000315
8 102.25 -0.000000000	40 131.75 -0.000000000	24 123.00 .000000000	56 -.000001602 -0.000000000
9 100.75 -0.00000799	41 127.00 -0.00000373	25 121.00 .000001130	57 -.000002985 -0.000001630
10 99.880 -0.00000271	42 128.00 .000000401	26 121.50 -0.00001469	58 -.0000010348 .000001339
11 102.12 -0.00000078	43 122.24 -0.00001011	27 120.12 .000002522	59 -.0000058928 -0.000001711
12 101.62 .000000316	44 126.87 -0.00000316	28 124.37 .00000027	60 -.000003016 -0.00000027
13 103.88 -0.00000043	45 123.50 .000000380	29 120.38 .000001281	61 -.000002253 -0.000001583
14 110.13 -0.000001615	46 121.00 .000001937	30 119.75 -0.00000618	62 -.000003121 .000000295
15 117.25 -0.00002237	47 117.87 .000001164	31 118.49 .000000573	63 -.000003713 .000000500

§§2 Gram polynomials

Gram polynomials are useful in fitting data by the linear least-squares method. The usual method is based on the following question: What is the “best” polynomial,

$$P_N(x) = \sum_{n=0}^N \gamma_n x^n, \quad (34)$$

(of order N) that I can use to fit some set of M pairs of data points,

$$\left\{ \begin{array}{l} x_k \\ f_k \end{array} \right\}, \quad k=0, 1, \dots, M-1$$

(with M > N) where $f(x)$ is measured at M distinct values of the independent variable x ?

The usual answer, found by Gauss, is to minimize the **squares** of the **deviations** (at the points x_k) of the fitting function $P_N(x)$ from the data – possibly weighted by the uncertainties of the data. That is, we want to minimize the **statistic**

$$\chi^2 = \sum_{k=0}^{M-1} \left(f_k - \sum_{n=0}^N \gamma_n x_k^n \right)^2 \frac{1}{\sigma_k^2} \quad (35)$$

with respect to the N + 1 parameters γ_n .

From the differential calculus we know that a function's first derivative vanishes at a minimum, hence we differentiate χ^2 with respect to each γ_n independently, and set the results equal to zero. This yields N + 1 linear equations in N + 1 unknowns:

$$\sum_m A_{nm} \gamma_m = \beta_n, \quad n=0, 1, \dots, N \quad (36)$$

where (the symbol $\hat{=}$ means “is defined by”)

$$A_{nm} \hat{=} \sum_{k=0}^{M-1} (x_k)^{n+m} \frac{1}{\sigma_k^2} \quad (37a)$$

and

$$\beta_n \hat{=} \sum_{k=0}^{M-1} x_k^n f_k \frac{1}{\sigma_k^2} \quad (37b)$$

In Chapter 9 we develop methods for solving linear equations. Unfortunately, they cannot be applied to Eq. 36 for $N \geq 9$ because the matrix A_{nm} approximates a Hilbert matrix,

$$H_{nm} = \frac{\text{const.}}{n+m+1},$$

a particularly virulent example of an exponentially ill-conditioned matrix. That is, the roundoff error in solving 36 grows exponentially with N , and is generally unacceptable. We can avoid roundoff problems by expanding in polynomials rather than monomials:

$$\chi^2 = \sum_{k=0}^{M-1} \left(f_k - \sum_{n=0}^N \gamma_n p_n(x_k) \right)^2 \frac{1}{\sigma_k^2}. \quad (38)$$

The matrix then becomes

$$A_{nm} = \sum_{k=0}^{M-1} p_n(x_k) p_m(x_k) \frac{1}{\sigma_k^2} \quad (39a)$$

and the inhomogeneous term is now

$$\beta_n = \sum_{k=0}^{M-1} p_n(x_k) f_k \frac{1}{\sigma_k^2} \quad (39b)$$

Is there any choice of the polynomials $p_n(x)$ that will eliminate roundoff? The best kinds of linear equations are those with nearly diagonal matrices. We note the sum in Eq. 39a is nearly an integral, if M is large. If we choose the polynomials so they are orthogonal with respect to the weight function

$$w(x) = \frac{1}{\sigma_k^2} \theta(x_k - x) \theta(x - x_{k-1}),$$

where

$$\theta(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

then A_{nm} will be nearly diagonal, and well-conditioned.

Orthogonal polynomials play an important role in numerical analysis and applied mathematics. They satisfy orthogonality relations²³ of the form

$$\int_A^B dx w(x) p_n(x) p_m(x) = \delta_{nm} \equiv \begin{cases} 1, & m=n \\ 0, & m \neq n \end{cases} \quad (40)$$

where the weight function $w(x)$ is positive.

For a given $w(x)$ and interval $[A, B]$, we can construct orthogonal polynomials using the Gram-Schmidt orthogonalization process.

Denote the integral in Eq. 40 by (p_n, p_m) to save having to write it many times. We start with

$$p_{-1} = 0, \\ p_0(x) = \left(\int_A^B dx w(x) \right)^{-1/2} = \text{const.},$$

and assume the polynomials satisfy the 2-term upward recursion relation

$$p_{n+1}(x) = (a_n + xb_n) p_n(x) + c_n p_{n-1}(x) \quad (41)$$

Now apply Eq. 41: assume we have calculated p_n and p_{n-1} and want to calculate p_{n+1} . Clearly, the orthogonality property gives

$$(p_{n+1}, p_n) = (p_{n+1}, p_{n-1}) = (p_n, p_{n-1}) = 0,$$

and the assumed normalization gives

$$(p_n, p_n) = 1.$$

These relations yields two equations for the three unknowns, a_n , b_n and c_n :

23. Polynomials can be thought of as vectors in a space of infinitely many dimensions ("Hilbert" space). Certain polynomials are like the vectors that point in the (mutually orthogonal) directions in ordinary 3-dimensional space, and so are called orthogonal by analogy.

$$a_n + b_n (p_n, x p_n) = 0$$

$$c_n + b_n (p_n, x p_{n-1}) = 0$$

We express a_n and c_n in terms of b_n to get

$$\begin{aligned} p_{n+1}(x) &= b_n \left[(x - (p_n, x p_n)) p_n(x) \right. \\ &\quad \left. - (p_n, x p_{n-1}) p_{n-1}(x) \right] \end{aligned} \tag{42}$$

We determine the remaining parameter b_n by again using the normalization condition:

$$(p_{n+1}, p_{n+1}) = 1.$$

In practice, we pretend $b_n = 1$ and evaluate Eq. 42; then we calculate

$$b_n = (\bar{p}_{n+1}, \bar{p}_{n+1})^{-1/2}, \tag{43}$$

multiply the (un-normalized) \bar{p}_{n+1} by b_n , and continue.

The process of successive orthogonalization guarantees that p_n is orthogonal to all polynomials of lesser degree in the set. Why is this so? By construction, $p_{n+1} \perp p_n$ and p_{n+1} . Is it $\perp p_{n-2}$? We need to ask whether

$$(p_n, (x - \alpha_n) p_{n-2}) = 0.$$

But we know that any polynomial of degree $N-1$ can be expressed as a linear combination of independent polynomials of degrees $0, 1, \dots, N-1$. Thus

$$(x - \alpha_n) p_{n-2} \equiv \sum_{k=0}^{n-1} \mu_k p_k(x) \tag{44}$$

and (by hypothesis) $p_n \perp$ every term of the rhs of Eq. 44, hence it follows (by mathematical induction) that

$$p_{n+1} \perp \{p_{n-2}, p_{n-3}, \dots\}.$$

Let us illustrate the process for Legendre polynomials, defined by weight $w(x) = 1$, interval $[-1,1]$:

$$p_0 = \left(\frac{1}{2}\right)^{1/2},$$

$$p_1 = \left(\frac{3}{2}\right)^{1/2} x,$$

$$p_2 = \left(\frac{5}{2}\right)^{1/2} \left(\frac{3}{2}x^2 - \frac{1}{2}\right),$$

.....

These are in fact the first three (normalized) Legendre polynomials, as any standard reference will confirm.

Now we can discuss Gram polynomials. While orthogonal polynomials are usually defined with respect to an integral as in Eq. 40, we might also define orthogonality in terms of a sum, as in Eq. 39a. That is, suppose we define the polynomials such that

$$\sum_{k=0}^{M-1} p_n(x_k) p_m(x_k) \frac{1}{\sigma_k^2} \equiv \delta_{nm} = \begin{cases} 1, & m=n \\ 0, & m \neq n \end{cases} \quad (45)$$

Then we can construct the Gram polynomials, calculating the coefficients by the algebraic steps of the Gram-Schmidt process, except now we evaluate sums rather than integrals. Since $p_n(x)$ satisfies 45 by construction, the coefficients γ_n in our fitting polynomial are simply

$$\gamma_n = \sum_{k=0}^{M-1} p_n(x_k) f_k \frac{1}{\sigma_k^2}; \quad (46)$$

they can be evaluated without solving any coupled linear equations, ill-conditioned or otherwise. Roundoff error thus becomes irrelevant.

The algorithm for fitting data with Gram polynomials may be expressed in flow-diagram form:

Read in points f_k , x_k and $w_k \equiv 1/\sigma_k^2$.

DO $n = 1$ to $N-1$ (outer loop)

Construct a_n , c_n :

DO $k = 0$ to $M-1$ (inner loop)

$$p_{n+1}(x_k) = (x_k - a_n)p_n(x_k) - c_n p_{n-1}(x_k)$$

$$\text{sum} = \text{sum} + (p_{n+1}(x_k))^2 w_k$$

$$c_{n+1} = c_{n+1} + f_k p_{n+1}(x_k) w_k$$

LOOP (end inner loop)

$$c_{n+1} = c_{n+1} / \text{sum}$$

)

DO $k = 0$ to $M-1$ (normalize)

$$p_{n+1}(x_k) = p_{n+1}(x_k) / \sqrt{\text{sum}}$$

)

LOOP

Fig. 8-5 Construction of Gram polynomials

The required storage is 5 vectors of length M to hold x_k , $p_n(x_k)$, $p_{n-1}(x_k)$, f_k and $w_k \equiv 1/\sigma_k^2$. We also need to store the coefficients a_n , c_n and the normalizations b_n — that is, 3 vectors of length $N < M$ — in case they should be needed to interpolate. The time involved is approximately $7M$ multiplications and additions for each n , giving $7NM$. Since N can be no greater than $M-1$ (M data determine at most a polynomial of degree $M-1$), the maximum possible running time is $7M^2$, which is much less than the time to solve M linear equations.

In practice, we would never wish to fit a polynomial of order comparable to the number of data, since this would include the noise as well as the significant information.

We therefore calculate a statistic called $\chi^2/(degree\ of\ freedom)$ ²⁴. With M data points and an N'th order polynomial, there are M-N-1 degrees of freedom. That is, we evaluate Eq. 38 for fixed N, and divide by M-N-1. We then increase N by 1 and do it again. The value of N to stop at is the one where

$$\sigma_{M,N}^2 = \frac{\chi_{M,N}^2}{M-N-1}$$

stops decreasing (with N) and begins to increase.

The best thing about the $\chi_{M,N}^2$ statistic is we can increase N without having to do any extra work:

$$\begin{aligned}\chi_{M,N}^2 &= \sum_{k=0}^{M-1} \left(f_k - \sum_n \gamma_n p_n(x_k) \right)^2 w_k \\ &\equiv \sum_{k=0}^{M-1} (f_k)^2 - \sum_{n=0}^N (\gamma_n)^2\end{aligned}\tag{47}$$

The first term after \equiv in Eq. 47 is independent of N, and the second term is computed as we go. Thus we could turn the outer loop (over N) into a **BEGIN ... WHILE ... REPEAT** loop, in which N is incremented as long as $\sigma_{M,N}^2$ is larger than $\sigma_{M,N+1}^2$. (Incidentally, Eq. 47 guarantees that as we increase N the fitted curve deviates less and less, on the average, from the measured points. When N = M-1, in fact, the curve goes through the points. But as explained above, this is a meaningless fit, since all data contain measurement errors. A fitted curve that passes closer than σ_k to more than about $1/3$ of the points is suspect.)

The code for Gram polynomials is relatively easy to write using the techniques developed in Ch. 5. The program is displayed in full in Appendix 8.4.

24. That is, "chi-squared per degree of freedom".

§§3 Simplex algorithm

Sometimes we must fit data by a function that depends on parameters in a nonlinear manner. An example is

$$f_k = \frac{F}{1 + e^{\alpha(x_k - X)}} \quad (48)$$

Although the dependence on the parameter F is linear, that on the parameters α and X is decidedly nonlinear.

One way to handle a problem like fitting Eq. 48 might be to transform the data, to make the dependence on the parameters linear. In some cases this is possible, but in 48 no transformation will render linear the dependence on all three parameters at once.

Thus we are frequently confronted with having to minimize numerically a complicated function of several parameters. Let us denote these by $\theta_0, \theta_1, \dots, \theta_{N-1}$, and denote their possible range of variation by \mathbf{R} . Then we want to find those values of $\{\theta\} \subset \mathbf{R}$ that minimize a positive function:

$$\chi^2(\theta_0, \dots, \theta_{N-1}) = \min_{\{\theta\} \subset \mathbf{R}} \chi^2(\theta_0, \dots, \theta_{N-1}) \quad (49)$$

One way to accomplish the minimization is *via* calculus, using a method known as *steepest descents*. The idea is to differentiate the function χ^2 with respect to each θ_k , and to set the resulting N equations equal to zero, solving for the $N \theta$'s. This is generally a pretty tall order, hence various approximate, iterative techniques have been developed. The simplest just steps along in θ -space, along the direction of the local downhill gradient $-\nabla \chi^2$, until a minimum is found. Then a new gradient is computed, and a new minimum sought²⁵.

Aside from the labor of computing $-\nabla \chi^2$, steepest descents has two main drawbacks: first, it only guarantees to find *a* minimum, not necessarily *the* minimum – if a function has several local

25. This is not by itself very useful. Useful modifications can be found in Press, et al., *Numerical Recipes*, *ibid.*, p. 301ff.

minima, steepest descents will not necessarily find the smallest. Worse, consider a function that has a minimum in the form of a steep-sided gulley that winds slowly downhill to a declivity — somewhat like a meandering river's channel. Steepest descents will then spend all its time bouncing up and down the banks of the gulley, rather than proceeding along its bottom, since the steepest gradient is always nearly perpendicular to the line of the channel.

Sometimes the function χ^2 is so complex that its gradient is too expensive to compute. Can we find a minimum *without* evaluating partial derivatives? A standard way to do this is called the **simplex method**. The idea is to construct a **simplex** — a set of $N+1$ distinct and **non-degenerate** vertices in the N -dimensional θ -space ("non-degenerate" means the geometrical object, formed by connecting the $N+1$ vertices with straight lines, has non-zero N -dimensional volume; for example, if $N=2$, the simplex is a triangle.)

We evaluate the function to be minimized at each of the vertices, and sort the table of vertices by the size of χ^2 at each vertex, the best (smallest χ^2) on top, the worst at the bottom. The simpler algorithm then chooses a new point in θ -space by the a strategy, expressed as the flow diagram, Fig. 8-8 on page 203 below, that in action somewhat resembles the behavior of an amoeba seeking its food. The key word **)MINIMIZE** that implements the complex decision tree in Fig. 8-8 (given here in pseudocode) is

```
: )MINIMIZE ( n.iter -- 87: rel.error -- )
INITIALIZE
BEGIN done? NOT N N.max < AND .
WHILE
    REFLECT r>=best?
    IF r>=2worst?
        IF r<worst? IF STORE.X THEN
            HALVE r<worst?
            IF STORE.X ELSE SHRINK THEN
        ELSE STORE.X THEN
    ELSE DOUBLE r>=best?
        IF STORE.XP ELSE STORE.X THEN
        THEN
        N 1+ IS N SORT
REPEAT ;
```

used in the format

USE(f.name 20 % 1.E-4)MINIMIZE

Fleshing out the details is a — by now — familiar process, so we leave the program *per se* to Appendix 8.5. We also include there a FORTRAN subroutine for the simplex algorithm, taken from

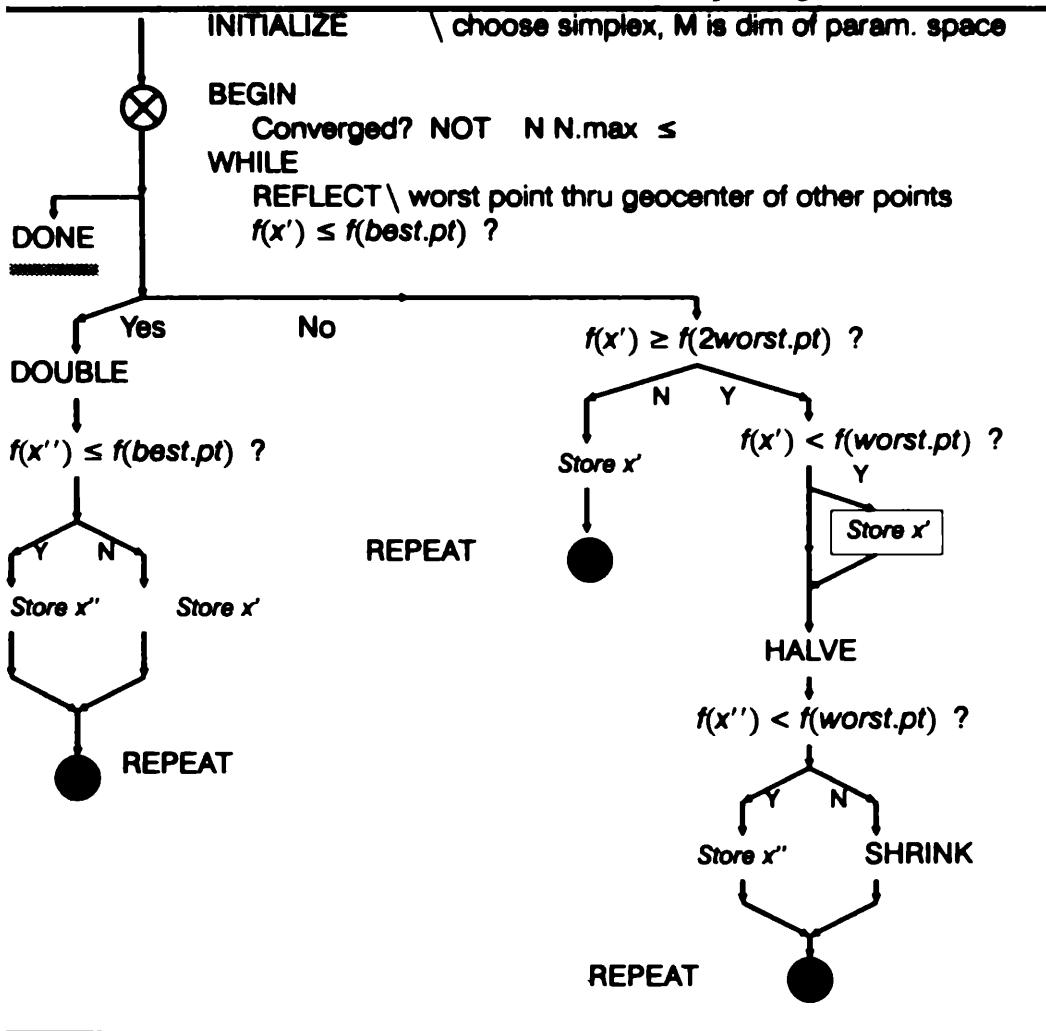


Fig. 8-8 Flow diagram of the simplex algorithm

*Numerical Recipes*²⁶, as an example of just how indecipherable traditional languages can be.

§3 Appendices

§§1 Gaussian quadrature

Gaussian quadrature formulae are based on the following idea: if we let the points ξ_n and weights w_n be $2N$ free parameters (n runs from 1 to N), what values of them most accurately represent an integral by the formula

$$I = \int_A^B dx \sigma(x) f(x) \approx \sum_{n=1}^N w_n f(\xi_n) ? \quad (50)$$

In Eq. 50 $\sigma(x)$ is a (known) positive function and $f(x)$ is the function we want to integrate. This problem can actually be solved, and leads to tables of points ξ_n and weight coefficients w_n specific to a particular interval $[A,B]$ and weight function $\sigma(x)$. Gauss-Legendre integration pertains to $[-1, +1]$ and $\sigma(x) = 1$. (Note any interval can be transformed into $[-1, +1]$.)

The interval $[0, \infty)$ and $\sigma(x) = e^{-x}$ leads to Gauss-Laguerre formulae, whereas the interval $(-\infty, +\infty)$ and $\sigma(x) = e^{-x^2}$ leads to Gauss-Hermite formulae.

Finally, we note that the more common integration formulae such as Simpson's rule or the trapezoidal rule can be derived on the same basis as the Gauss methods, except that the points are specified in advance to be equally spaced and to include the endpoints of the interval. Only the weights w_n can be determined as free fitting parameters that give the best approximation to the integral.

For given N Gaussian formulae can be more accurate than equally-spaced rules, as they have twice as many parameters to play with.

26. Press, et al., *Numerical Recipes*, *ibid.*, p. 289ff.

Some FORTH words for 5-point Gauss-Legendre integration:

% 0.906170045938884	FCONSTANT	x2	:)Integral	(87: A B -- I)
% 0.538469310105683	FCONSTANT	x1	scale	(87: A B -- [A+B]/2 [B-A]/2)
% 0.568888888888889	FCONSTANT	w0	FOVER	F(X) w0 F* F-ROT (87: -- I a b)
% 0.478628670499388	FCONSTANT	w1	XDUP	x1 rescale F(X) w1 F* F3R +
% 0.2369268865056189	FCONSTANT	w2	XDUP	x1 FNNEGATE rescale
: scale (87: A B -- [A+B]/2 [B-A]/2)			F(X)	w1 F* F3R +
FOVER	F-	F2/	XDUP	x2 rescale F(X) w2 F* F3R +
: rescale (87: a b x -- a+b*x)	F*	F+ ;	XDUP	x2 FNNEGATE rescale
: F3R + (87: a b c x -- a+x b c)	F3R	F+ F-ROT ;	F(X)	w2 F* F3R +

§§2 The trapezoidal rule

Recall (Ch. 8, §1.1) how we approximated the area under a curve by capturing it between rectangles consistently higher- and lower than the curve; and calculating the areas of the two sets of rectangles. In practice we use a better approximation: we average the rectangular upper and lower bounds. The errors tend to cancel, resulting in²⁷

$$\begin{aligned} & \frac{w}{2} \sum_{n=0}^{(B-A)/w} (f(A+nw) + f(A+nw+w)) \\ & = \int_A^B dx f(x) + \frac{1}{12} \left(\frac{B-A}{w} \right) w^3 \max_{A < x < B} |f''(x)| \end{aligned} \quad (51)$$

Now the error is much smaller²⁸ – of order w^2 – so if we double the number of points, we decrease the error four-fold. Yet this so-called trapezoidal rule²⁹ requires no more effort (in terms of the number of function evaluations) than the rectangle rule.

27. See, e.g., Abramowitz and Stegun, *HMF*, p. 885.

28. $f''(x)$ is the second derivative of $f(x)$, i.e. the first derivative of $f'(x)$.

29. Called so because the curve $f(x)$ is approximated by straight line segments between successive points $x, x + w$. Thus we evaluate the areas of trapezoids rather than rectangles.

§§3 Richardson extrapolation

In the program)INTEGRAL in Ch.8 §§5.3 the word **INTERPOLATE** performs Richardson extrapolation for the trapezoidal rule. The idea is this: If we use a given rule, accurate to order w^n , to calculate the integral on an interval $[a,b]$, then presumably the error of using the formula on each half of the interval and adding the results, will be smaller by 2^{-n} . For the trapezoidal rule, $n=2$, hence we expect the error from summing two half-intervals to be $4\times$ smaller than that from the whole interval.

Thus, we can write $(I_0 = \int_a^b, I'_0 = \int_a^{(a+b)/2}, I_1 = \int_{(a+b)/2}^b)$

$$I_0 = I_{\text{exact}} + R \quad (52a)$$

$$I'_0 + I_1 = I_{\text{exact}} + \frac{R}{4} \quad (52b)$$

Equation 52b is only an approximation because the R that appears in it is not exactly the same as R in Eq. 52a. We will pretend the two R 's are equal, however, and eliminate R from the two equations (52a,b) ending with an expression for I_{exact} :

$$I_{\text{exact}} \approx \frac{4}{3}(I'_0 + I_1 - I_0) \quad (53)$$

Equation 53 is exactly what appears in **INTERPOLATE**.

§4 Linear least-squares: Gram polynomials

Here is a FORTH program for implementing the algorithm derived in §2§2 above.

```

: ASK.GRAM1
CODE G(N+1) 4 FMUL. FCOS. 2 FLD. 6 FSUBB.
2 FMUL. 1 FADDP.
DX DS MOV. DS POP. R84 DS: [BX] FST.
DS DX MOV. BX POP. END-CODE
seg off -- 87: s a b w x g[n] g[n-1] .. s a b w x g[n] g[n+1])

CODE B(N+1) 2 FXCH. 2 FMUL. 3 FMUL.
1 FXCH. 1 FMUL. DX DS MOV. DS POP.
R84 DS: [BX] FADD. DS: [BX] FSTP.
DS DX MOV. BX POP. END-CODE
seg off --
87: s a b w x g[n] g[n+1] .. s a b w g[n+1] w x g[n+1])

CODE A(N+1) 1 FMUL. DX DS MOV. DS POP.
R84 DS: [BX] FADD.
DS: [BX] FSTP. DS DX MOV. BX POP. END-CODE
seg off -- 87: s a b w g[n+1] w g[n+1] .. s a b w g[n+1])

CODE C(N+1) 1 FXCH. 2 FMUL*. 2 FMUL.
2 FXCH. 1 FMULP.
DX DS MOV. DS POP.
R84 DS: [BX] FADD. DS: [BX] FSTP.
DS DX MOV. BX POP. 3 FADDP. END-CODE
seg off -- 87: s a b w g[n+1] f .. s = s + w g[n+1]**2 a b)

-----Gram polynomial coding
: REAL SCALAR DELTA
: VAR Nmax
: usage: A{ B{ C{ G{ Nmax }FIT

: FIRST.AB's F=0 a{0} G! F=0 b{0} G! ;

INIT.DELTA (87:--g{{11}}) F=0 F=0
M 0 DO w{1} G@ y{1} G@ F**2 FOVER F*
(87:s'--ss'ww^4^2)
FROT F+ F-ROT F+ FSWAP
(87:--s=s+w s'=s'+w^4^2)
LOOP DELTA G! FSORT 1/F ;

: SECOND.AB's (87:g{{11}}..g{{11}})
F=0 b{10} G! FDUP g{{11}} G! LOOP ;
F=0 M 0 DO w{10} G@ x{10} G@ F* F+ LOOP
F* a{10} G!;

: FIRST.& SECOND.C's (87:g{{11}}..)
F=0 c{00} G! F=0
M 0 DO w{10} G@ y{10} G@ F* F+ LOOP
F* c{10} G!;

: INITIALIZE FINIT IS Nmax IS g{{11}} IS b{{11}} IS a{{11}}
FIRST.AB's INIT.DELTA FIRST.G's SECOND.AB's
FIRST.& SECOND.C's ;

: VAR N 0 VAR N+1
: inc.N N+1 DUP IS N 1+ IS N+1;
: DISPOSE R DDUP R@ G@ R;
: inc.OFF #BYTES DROP +;
: ZERO.L DDUP F=0 R84IL;

: START.Next.G
( -- [c{n+1}] [a{n+1}] [b{n+1}] 87:--s=0 a{n} b{n})
FINIT F=0 c{N+1} DROP ZERO.L
a{N} DISPOSE inc.OFF ZERO.L
b{N} DISPOSE inc.OFF ZERO.L;

: SET.FSTACK w{10} G@ x{10} G@ g{{N1}} G@ g{{N1-1}} G@ ;
: )@*1 (adr n-- 87:x--x) FDUP 0} DISPOSE F* G!;

: NORMALIZE (87:sum--)
1/F a{N+1}@*1 FSORT
b{N+1}@*1 c{N+1}@*1
M 0 DO FDUP g{{N+1}} DISPOSE F* G! LOOP
FDROP;

CODE 6DUP OPT* 6 PICK 6 PICK 6 PICK
6 PICK 6 PICK 6 PICK * END-CODE
CODE 6DROP OPT* DDROP DDROP DDROP * END-CODE

```

```
\ GRAM POLYNOMIAL LEAST-SQUARES (CONT'D)

Next.G START.Next.G
M 0 DO 8DUP SET.FSTACK
    g{{ N+1 }} DROP G(N+1) B(N+1) A(N+1)
    y{10} G@ C(N+1)
LOOP 8DROP FDROP FDROP NORMALIZE;

: New.DELTA ( :: - old.delta new.delta)
    DELTA G@ FDUP c{ N 0} G@ F**2 F- FDUP
    DELTAG!;

: NOT.ENUF.G? New.DELTA M N+1- S-F
    FDUP F=1 F-
(CR .FS ." NEXT ITERATION?" ?YN 0= IF ABORT THEN )
    ( :: - d d' m-n-1 m-n-2 )
    FROT F\ F-ROT F/ ( :: - d'/(m-n-2) d/(m-n-1) )
    FOVER F0 IF F ELSE FDROP FDROP 0 THEN ;

: }FIT ( X{ Y{ S{ Nmax A{ B{ C{ G{{ - }
    INITIALIZE 1 IS N 2 IS N+1
    BEGIN NOT.ENUF.G? N Nmax AND
    WHILE Next.G Inc.N
    REPEAT FINIT;
\----- end of code

: RECONSTRUCT M 0 DO CR x{10} G@ F. y{10} G@ F.
    F=0 N+1 1 DO
        c{10} G@ g{{ IJ }} G@ F* F+
    LOOP F.
    LOOP ;
```

§§5 Non-linear least squares: simplex method

A FORTRAN program for the simplex method is given below on page 209. The FORTH version, as discussed in §§3 given on pages 210 and 211.

```

SUBROUTINE AMOEBA(P,Y,MP,NP,NDIM,FTOL,FUNK,ITER)
PARAMETER (NMAX = 50,ALPHA = 1.0,
           BETA = 0.5,GAMMA = 2.0,ITMAX = 500)
DIMENSION P(MP,NP),Y(MP),PR(NMAX),PBAR(NDIM)
MPTS = NDIM + 1
ITER = 0
ILO = 1
IF(Y(1).GT.Y(2))THEN
  IH = 1
  NH = 2
ELSE
  IH = 2
  NH = 1
ENDIF
DO 11 I = 1,MPTS
  IF(Y(I).LT.Y(ILO)) ILO = I
  IF(Y(I).GT.Y(IH))THEN
    NH = IH
    IH = I
  ELSE IF(Y(I).GT.Y(NH))THEN
    IF(.NE.IH) NH = I
  ENDIF
  1 CONTINUE
  RTOL = 2 * ABS(Y(NH)-Y(ILO))/(ABS(Y(NH))+ABS(Y(ILO)))
  IF(RTOL.LT.FTOL)RETURN
  IF(ITER.EQ.ITMAX) PAUSE 'Amoeba exceeding maximum iterations.'
  ITER = ITER + 1
  DO 12 J = 1,NDIM
    PBAR(J) = 0.
  2 CONTINUE
  DO 14 I = 1,MPTS
    IF(.NE.IH)THEN
      DO 13 J = 1,NDIM
        PBAR(J) = PBAR(J) + P(I,J)
    3 CONTINUE
    ENDIF
  4 CONTINUE
  DO 15 J = 1,NDIM
    PBAR(J) = PBAR(J)/NDIM
    PR(J) = (1. + ALPHA)*PBAR(J)-ALPHA*P(IH,J)
  5 CONTINUE
  YPR = FUNK(PR)
  IF(YPR.LE.Y(ILO))THEN
    DO 16 J = 1,NDIM
      PR(J) = GAMMA*PR(J) + (1.-GAMMA)*PBAR(J)
  6 CONTINUE

```

```

  YPRR = FUNK(PR)
  IF(YPRR.LT.Y(ILO))THEN
    DO 17 J = 1,NDIM
      P(IH,J) = PR(J)
    17 CONTINUE
    Y(IH) = YPRR
  ELSE
    DO 18 J = 1,NDIM
      P(IH,J) = PR(J)
    18 CONTINUE
    Y(IH) = YPR
  ENDIF
  ELSE IF(YPRR.GE.Y(NH))THEN
    IF(YPRR.LT.Y(IH))THEN
      DO 19 J = 1,NDIM
        P(IH,J) = PR(J)
    19 CONTINUE
    Y(IH) = YPRR
  ENDIF
  DO 21 J = 1,NDIM
    PR(J) = BETA*P(IH,J) + (1.-BETA)*PBAR(J)
  21 CONTINUE
  YPRR = FUNK(PR)
  IF(YPRR.LT.Y(IH))THEN
    DO 22 J = 1,NDIM
      P(IH,J) = PR(J)
    22 CONTINUE
    Y(IH) = YPRR
  ELSE
    DO 24 I = 1,MPTS
      IF(.NE.ILO)THEN
        DO 23 J = 1,NDIM
          PR(J) = 0.5*(P(I,J) + P(ILO,J))
          P(I,J) = PR(J)
        23 CONTINUE
        Y(I) = FUNK(PR)
      ENDIF
    24 CONTINUE
    ENDIF
  ELSE
    DO 25 J = 1,NDIM
      P(IH,J) = PR(J)
    25 CONTINUE
    Y(IH) = YPRR
  ENDIF
  GO TO 1
END

```

```
\MINIMIZATION BY THE SIMPLEX METHOD
\VERSION OF 20:51:19 4/30/1991

TASK AMOEBA

\----- FUNCTION NOTATION
VARIABLE <F>
:USE( [COMPILE] ' CFA <F> );
:F() EXECUTE@ ;
BEHEAD' <F>
\----- END FUNCTION NOTATION

\----- DATA STRUCTURES
3 VAR Ndim
0 VAR N
0 VAR N.max

CREATE SIMPLEX{{ Ndim 4 (bytes) * Ndim 1+ (# points)
* ALLOT
CREATE F{ Ndim 1+ 4 (bytes) * ALLOT \residuals
CREATE index Ndim 1+ 2* ALLOT \array for scrambled indices

:>index (I - I') 2* index + @ ;
DVARIABLE Residual
DVARIABLE Residual'
DVARIABLE Epsilon
CREATE X{ Ndim 4 (bytes) * ALLOT \trial point
CREATE XP{ Ndim 4 (bytes) * ALLOT \2nd trial point
CREATE Y{ Ndim 4 (bytes) * ALLOT \geocenter

:} (adr n - adr + 4n) 4* + ; \part of array notation
:} (adr m n - adr + [m*Ndim + n]*4) SWAP Ndim * + } ;
\----- END DATA STRUCTURES
\----- ACTION WORDS
:RESIDUALS
Ndim 1+ 0 DO SIMPLEX{{ 10 }} F(X) F(1) R32!
LOOP ;
:<index> Ndim 1+ 0 DO I index 12* + ! LOOP ; \fill index

:ORDER <index>
Ndim 1+ 0 DO F{ 1 >index } R32@
I 1+ BEGIN Ndim 1+ OVER
WHILE F( OVER >index ) R32@ FOVER FOVER F>
IF FSWAP
I >index OVER >index
Index 12* + ! Index 3 PICK 2* +
THEN FDROP 1+
REPEAT FDROP DROP
LOOP ;
CENTER ( - ) FINIT Ndim S-F (87: - Ndim)
Ndim 0 DO F=0 \loop over components
Ndim 0 DO \average over vectors

SIMPLEX{{ 1 Index J }} R32@ F+
LOOP FOVER F/

```

```
Y{1} R32! \put away
LOOP FDROP ;
\ note: Worst.Point is SIMPLEX{{ Ndim Index J }}
\ -- excluded!

: DONE! CR ."We're finished." ;
: TOO.MANY CR ."Too many iterations." ;

: VMOVE (src.adr dest.adr -) \move vector
Ndim 0 DO OVER 1) OVER 1) 2 MOVE
LOOP DDROP ;
: STORE (adr1 adr2-- ) 0 }
SIMPLEX{{ Ndim Index 0 }} VMOVE
F{ Ndim Index } 2 MOVE ;
: STOREX Residual X{ STORE ;
: STOREXP Residual' XP{ STORE ;

: New.F X{ F() Residual R32! ;
: EXTRUDE (87: scale.factor-- )
\extend pseudopod
Ndim 0 DO DUP 1) R32@ (87: -- s.fx)
Y{1} R32@ FUNDER F- (87: -- s.fy x.y)
FROT FUNDER F* (87: -- y s.f [x-y]*s.f)
FROT F+ (87: -- s.fy + [x-y]*s.f)
X{1} R32!
LOOP DROP FDROP New.F ;

: F=1/2 F=1 FNEGATE F=1 FSCALE FPLUCK ;
: F=2 F=1 FDUP FSCALE FPLUCK ;

\----- DEBUGGING CODE
0 VAR DBG
: DEBUG-ON -1 IS DBG ;
: DEBUG-OFF 0 IS DBG ;
: .V 3 SPACES Ndim 0 DO DUP 1) R32@ F.
LOOP DROP ;
:.M (--)
Ndim 1+ 0 DO DBG CR
IF 1. 2 SPACES THEN
SIMPLEX{{ 1 Index 0 }} .V
DBG IF F{ 1 Index } R32@ F. THEN
LOOP CR
DBG IF CR X{ V Residual R32@ F. THEN ;

:F Ndim 1+ 0 DO CR F{ 1 Index } R32@ F.
LOOP ;
\----- END DEBUGGING CODE
:REFLECT CENTER
F=1 FNEGATE \scale.factor = -1
SIMPLEX{{ Ndim Index 0 }} \worst pt.
EXTRUDE \calculate x, f(x)
DBG IF CR ."REFLECTING" THEN .M ;

```

ANOESIA CONT'D

```

DOUBLE Residual Residual' 2 MOVE
  \ save residual
  X( XP( V.MOVE          \ save point
  FINIT F=2 FNNEGATE (?) \ scale.factor = -2
  SIMPLEX({ Ndim Index 0 }) \ worst pt.
  EXTRUDE           \ calculate x, f(x)
  DBG IF CR." DOUBLING" THEN .M ;
HALVE FINIT F=1/2      \ scale factor = 0.5
  SIMPLEX({ Ndim Index 0 }) \ worst pt.
  EXTRUDE           \ calculate x, f(x)
  DBG IF CR." HALVING" THEN .M ;
SHRINK SIMPLEX({ 0 >Index 0 }) \ best pt.
  Y( V.MOVE          \ save it
  Ndim 1+ 1 DO        \ by vector
    Ndim 0 DO          \ by component
      SIMPLEX({ J >Index 1 }) DUP
      R32@ Y(1) R32@ FUNDER F-
      F=1/2 F* F+ R32!
    LOOP
  LOOP RESIDUALS ORDER
  DBG IF CR." SHRINKING" THEN .M ;
----- END ACTION WORDS
----- TEST WORDS
test) FINIT Residual R32@
  F{ SWAP >Index } R32@ F> NOT ;
  >=best? 0 (test) ;

<worst? Ndim (test) NOT ;
  >=2worst? Ndim 1- (test) ;

tans? F{ Ndim Index } R32@
  F{ 0 Index } R32@
  FOVER FOVER F- F2*
  F-ROT F+ F/ FABS Epsilon R32@ F> ;
----- END TEST WORDS

```

```

: MINIMIZE ( n,iter .. 87: error .. )
IS Number Epsilon R32@ 0 IS N          \ Initialize
RESIDUALS          \ compute residuals
ORDER              \ locate best, 2worst, worst
BEGIN              \ start iteration
  done? NOT N N,max < AND
WHILE
  REFLECT r>=best?
  IF r>=2worst?
    IF r<worst? IF STOREX THEN
      HALVE r<worst?
      IF STOREX ELSE SHRINK THEN
      ELSE STOREX THEN
    ELSE DOUBLE
      r>=best?
      IF STOREXP ELSE STOREX THEN
    THEN
      N 1+ IS N ORDER
    REPEAT DONE! ;
----- EXAMPLE FUNCTIONS
:F1   (adr - 87: F) DUP 0 } R32@ FDUP F**2
                  DUP 1 } R32@ F**2 F2* F+
                  2 } R32@ F**2 F2* F2* F+
                  36 S-F F- F**2 F2/ FSWAP 6 S-F F* F-
                  \ f1 = .5 * (x*x + 2*y*y + 4*z*z - 36) ^ 2 - 6*x
:F2   (adr - 87: F) DUP DUP 0 }
  R32@ FDUP F**2 1 } R32@ F**2
  F2* F+ 36 S-F F- F**2 F2/ FSWAP 6 S-F F* F-
  F**2 F+ } R32@ 0 } R32@ F/ FATAN F2* FCOS
  \ f2 = [(x^2+y^2-36)^2 - 6*x] * cos(2*atan(y/x)) ^ 2
  \ Usage: USE(MYFUNC 10 1.E-3)MINIMIZE
FLOATS
5. SIMPLEX({ 0 0 }) R32!
3. SIMPLEX({ 0 1 }) R32!
7. SIMPLEX({ 0 2 }) R32!
5. SIMPLEX({ 1 0 }) R32!
3. SIMPLEX({ 1 1 }) R32!
-1.5 SIMPLEX({ 1 2 }) R32!
-10. SIMPLEX({ 2 0 }) R32!
1. SIMPLEX({ 2 1 }) R32!
3. SIMPLEX({ 2 2 }) R32!
5. SIMPLEX({ 3 0 }) R32!
3. SIMPLEX({ 3 1 }) R32!
3. SIMPLEX({ 3 2 }) R32!

```


Linear Algebra

Contents

§1 Simultaneous linear equations	214
§§1 Theory of linear algebraic equations	214
§§2 Eigenvalue problems	215
§2 Solving linear equations	218
§§1 Cramer's rule	218
§§2 Pivotal Elimination	221
§§3 Testing	225
§§4 Implementing pivotal elimination	227
§§5 The program	227
§§6 Timing	238
§3 Matrix inversion	242
§§1 Linear transformations	242
§§2 Matrix multiplication	243
§§3 Matrix inversion	244
§§4 Why invert matrices, anyway?	245
§§5 An example	245
§4 LU decomposition	246

Two common problems in scientific programming are the numerical solution of simultaneous linear algebraic equations and computing the inverse of a given square matrix. This is such an important subject As an exercise in FORTH program development we shall now write programs to solve linear equations and invert matrices.

§1 Simultaneous linear equations

We begin with a dose of mathematics (linear algebra), and then develop programs. Several actual programming sessions are reproduced *in toto* — mistakes and all — to give the flavor of program development and debugging in FORTH.

§§1 Theory of linear algebraic equations

We begin by stating the problem. Given the equations

$$\sum_{n=0}^{N-1} A_{mn} x_n = b_m. \quad (1)$$

— more compactly written $\vec{A} \cdot \vec{x} = \vec{b}$ — with known coefficient matrix A_{mn} and known inhomogeneous term b_m : under what conditions can we find unique values of x_n that simultaneously satisfy the N equations 1?

The theory of simultaneous linear equations tells us that if not all the b_m 's are 0, the necessary and sufficient condition for solvability is that the determinant¹ of the matrix \vec{A} should not be 0. Contrariwise, if $\det(\vec{A}) = 0$, a solution with $\vec{x} \neq 0$ can be found when $\vec{b} = 0$.

1. The determinant of an N'th-order square matrix \vec{A} — denoted by $\det(\vec{A})$ or $||\vec{A}||$ — is a number computed from the elements A_{mn} by applying rules familiar from linear algebra. These rules define $||\vec{A}||$ recursively in terms of determinants of matrices of square submatrices of \vec{A} . See §§2.1.

§§2 Eigenvalue problems

Many physical systems can be represented by systems of linear equations. Masses on springs, pendula, electrical circuits, structures², and molecules are examples. Such systems often can oscillate sinusoidally. If the amplitude of oscillation remains bounded, such motions are called **stable**. Conversely, sometimes the motions of physical systems are unbounded – the amplitude of any small disturbance will increase exponentially with time. An example is a pencil balanced on its point. Exponentially growing motions are called – for obvious reasons – **unstable**.

Clearly it can be vital to know whether a system is stable or unstable. If stable, we want to know its possible frequencies of free oscillation; whereas for unstable systems we want to know how rapidly disturbances increase in magnitude. Both these problems can be expressed as the question: do linear equations of the form

$$\mathbf{A} \cdot \vec{x} = \lambda \mathbf{P} \cdot \vec{x} \quad (2)$$

have solutions? Here λ is generally a complex number, called the **eigenvalue** (or **characteristic value**) of Eq. 2, and \mathbf{P} is often called the **mass matrix**. Frequently \mathbf{P} is the unit matrix \mathbf{I} ,

$$I_{mn} = \begin{cases} 1, & m = n \\ 0, & m \neq n \end{cases}, \quad (3)$$

but in any case, \mathbf{P} must be **positive-definite** (we define this below). A non-trivial solution, $\vec{x} \neq 0$, of the equation

$$\mathbf{A} \cdot \vec{x} = 0 \quad (4)$$

exists if and only if $\|\mathbf{A}\| = 0$. This fact is useful in solving eigenvalue problems such as Eq. 2 above.

2. buildings, cars, airplanes, bridges ...

The secular equation (or determinantal equation)

$$\|\mathbf{A} - \lambda \mathbf{P}^* \| = 0 \quad (5)$$

is a polynomial of degree N in λ , hence has N roots (either real or complex)³. When $\mathbf{P}^* = \mathbf{I}$, these roots are called the **eigenvalues** (or "characteristic values") of the matrix \mathbf{A}^* .

Eigenvalue problems arising in physical contexts usually involve a restricted class of matrices, called **real-symmetric** or **Hermitian** (after the French mathematician Hermite) matrices, for which $A_{mn}^* = A_{nm}$. (The superscript * denotes complex conjugation – see Ch. 7.) All the eigenvalues of Hermitian matrices are **real** numbers. How do we know? We simply consider Eq. 3 and its **complex conjugate**:

$$\sum_n A_{mn} x_n = \lambda \sum_n \rho_{mn} x_n ; \quad (6a)$$

$$\sum_n x_n^* A_{nm}^* = \lambda^* \sum_n x_n^* \rho_{nm}^* ; \quad (6b)$$

Equation 6b can be rewritten (using the fact that \mathbf{A}^* and \mathbf{P}^* are Hermitian)

$$\sum_m x_m^* A_{mn} = \lambda^* \sum_n x_m^* \rho_{mn} \quad (6b')$$

Multiply 6b' by x_m and 6a by x_m^* , sum both over m and subtract this gives

$$0 = (\lambda^* - \lambda) \sum_{n,m} x_m^* \rho_{mn} x_n . \quad (7)$$

However, as noted above, ρ is **positive-definite**, i.e.

3. This follows from the **fundamental theorem of algebra**: a polynomial equation, $p(z) = a_0 + a_1 z + a_2 z^2 + \dots + a_n z^n = 0$, of degree n (in a complex variable z) has exactly n solutions.

$$\mathbf{x}^T \cdot \rho \cdot \mathbf{x} = \sum_{n,m} x_m^* \rho_{mn} x_n > 0 \quad (8)$$

for any non-zero vector \mathbf{x} . Thus, Eq. 7 $\Rightarrow \lambda^* = \lambda$, that is, λ is real.

In vibration problems, the eigenvalue λ usually stands for the square of the (angular) vibration frequency: $\lambda = \omega^2$. Thus, a positive eigenvalue λ corresponds to a (double) real value, $\pm\omega$, of the angular frequency. Real frequencies correspond to sinusoidal vibration with time-dependence $\sin(\omega t)$ or $\cos(\omega t)$.

Conversely, a negative λ corresponds to an imaginary frequency, $\pm i\omega$ and hence to a solution that grows exponentially in time, as

$$\begin{aligned} \sin(i\omega t) &= i \sinh(\omega t) \\ \cos(i\omega t) &= \cosh(\omega t) \end{aligned} \quad (9)$$

There are many techniques for finding eigenvalues of matrices. If only the largest few are needed, the simplest method is iteration: make an initial guess $\mathbf{x}^{(0)}$ and let

$$\mathbf{x}^{(n)} = \frac{\mathbf{A} \mathbf{x}^{(n-1)}}{\left(\mathbf{x}^{(n-1)}, \rho \mathbf{x}^{(n-1)} \right)^{1/2}}$$

Assuming the largest eigenvalue is unique, the sequence of vectors $\mathbf{x}^{(n)}$, $n = 1, 2, \dots$, is guaranteed to converge to the vector corresponding to that eigenvalue, usually after just a few iterations.

If all the eigenvalues are wanted, then the only choice is to solve the secular equation 5 for all N roots.

§2 Solving linear equations

The test-and-development cycle in FORTH is short compared with most languages. It is usually easy to create a working program that subsequently can be tuned for speed, again much

4. For clarity we now omit the vector " \rightarrow " and dyad " \leftrightarrow " symbols from vectors and matrices.

more rapidly than with other languages. For me this is the chief benefit of the language.

§§1 Cramer's rule

Cramer's rule is a constructive method for solving linear equations by computing determinants. It is completely impractical as a computer algorithm because it requires $O(N!)$ steps to solve N linear equations, whereas pivotal elimination (that we look at below) requires $O(N^3)$ steps, a much smaller number. Nevertheless Cramer's rule is of theoretical interest because it is a closed-form solution.

Consider a square $N \times N$ matrix \mathbf{A} . Pretend for the moment we know how to compute the determinant of an $(N-1) \times (N-1)$ matrix. The determinant of \mathbf{A} is defined to be

$$\det(\mathbf{A}) = \sum_{n=0}^{N-1} A_{mn} a_{nm}$$

where the a_{mn} 's are called **co-factors** of the matrix elements A_{mn} and are in fact determinants of specially selected $(N-1) \times (N-1)$ sub-matrices of \mathbf{A} .

The sub-matrices are chosen by striking out of \mathbf{A} the n 'th column and m 'th row (leaving an $(N-1) \times (N-1)$ matrix). To illustrate, consider the 3×3 matrix

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 5 \\ 3 & 2 & 4 \\ 1 & 1 & 6 \end{pmatrix} \tag{11}$$

and produce the co-factor of A_{12} :

$$a_{21} = (-1)^{2+1} \begin{vmatrix} & 0 & 1 & 2 \\ 0 & 1 & 0 & 5 \\ 1 & 3 & 2 & 4 \\ 2 & 1 & 1 & 6 \end{vmatrix} \quad (12)$$

Column labels
Row labels

We also attach the factor $(-1)^{m+n}$ to the determinant of the submatrix when we compute a_{nm} .

A determinant changes sign when any two rows or any two columns are interchanged. Thus, a determinant with two identical rows or columns is exactly zero⁵. What would happen to Eq. 11 if instead of putting A_{mn} in the sum we put A_{kn} where $k \neq m$? By inspection we realize that this is the same as evaluating a determinant in which two rows are the same, hence we get zero. Thus Eq. 11 can be rewritten more generally

$$\sum_{n=0}^{N-1} A_{kn} a_{nm} = \|A\| \delta_{km} . \quad (13)$$

This feature of determinants lets us solve the linear equation $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ by construction: try

$$x_n = \frac{1}{\det(A)} \sum_{m=0}^{N-1} a_{nm} b_m . \quad (14)$$

We see from Eq. 13 that Eq. 14 solves the equation. Equation 14 also makes clear why the solution cannot be found if $\det(A) = 0$.

A determinant also vanishes when a row is a *linear combination* of any of the other rows. Suppose row 0 can be written

5. The only number equal to its negative is 0.

$$a_{0k} \equiv \sum_{m=1}^{N-1} \beta_m a_{mk};$$

that is, the 0'th equation can be derived from the other N-1 equations, hence it contains no new information. We do not really have N equations for N unknowns, but at most N-1 equations. The N unknowns therefore cannot be completely specified, and the determinant tells us this by vanishing.

As an example, we now use Cramer's rule to evaluate the determinant of Eq. 11. We will write

$$\|A\| = A_{00} a_{00} + A_{01} a_{10} + A_{02} a_{20}$$

$$a_{10} = \begin{vmatrix} 2 & 4 \\ 1 & 6 \end{vmatrix}$$

$$a_{10} = \begin{vmatrix} 3 & 4 \\ 1 & 6 \end{vmatrix} (-1)$$

$$a_{20} = \begin{vmatrix} 3 & 2 \\ 1 & 1 \end{vmatrix}$$

The determinant of a 1×1 matrix is just the matrix element, hence

$$a_{00} = 2 \cdot 6 + (-1) \cdot 4 \cdot 1 = 8$$

$$a_{10} = -(18 - 4) = -14$$

$$a_{20} = (3 - 2) = 1$$

$$\|A\| = 1 \cdot 8 + 0 \cdot (-14) + 5 \cdot 1 = 13.$$

How many operations does it take to evaluate a determinant? We see that a determinant of order N requires N determinants of order N-1 to be evaluated, as well as N multiplications and N-1

additions. If the addition time plus the multiplication time is τ , then

$$T_N = N(\tau + T_{N-1}).$$

It is easy to see^{6,7} the solution to this is

$$T_N = N! \tau \sum_{n=0}^N \frac{1}{n!} \xrightarrow{N \rightarrow \infty} N! \tau e.$$

In other words, the time required to solve N linear equations by Cramer's rule increases so rapidly with N as to render the method thoroughly impractical.

§§2 Pivotal Elimination

The algorithm we shall use for solving linear equations is elimination, just as we were taught in high school algebra. However we modify it to take into account the experience of 40 years's solving linear equations on digital computers (not to mention a previous 20 years's worth on mechanical calculators!), to minimize the buildup of round-off error and consequent loss of precision⁸.

The necessary additional step involves pivoting – selecting the largest element in a given column to normalize all the other

6. Professors always say this, hee, hee! See R. Sedgewick, *Algorithms* (Addison-Wesley Publishing Company, Reading, MA 1983).
7. $N!$ means $N \times (N-1) \times (N-2) \times \dots \times 2 \times 1$. The base of the “natural” logarithms is $e = 2.7182818\dots$
8. A computer stores a number with finite precision – say 6-7 decimal places with 32-bit floating-point numbers. This is enough for many purposes, especially in science and engineering, where the data are rarely measured to better than 1% relative precision. Suppose, however, that two numbers, about 10^2 in magnitude, are multiplied. Their product is of order 10^4 and is known to six significant figures. Now add it to a third number of order unity. The result will be that third number $\pm 10^4$. Later, a fourth number – also of order unity – is subtracted from this sum. The result will be a number of order 10^4 , but now known only to two significant figures. Matrix arithmetic is full of multiplications and additions. The lesson is clear – to minimize the (inevitable) loss of precision associated with round-off, we must try to keep the magnitudes of products and sums as close as possible.

elements. This will be clearer with a concrete illustration rather than further description: Consider the 3×3 system of equations:

$$\begin{pmatrix} 1 & 0 & 5 \\ 3 & 2 & 4 \\ 1 & 1 & 6 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 4 \\ 2 \end{pmatrix} \quad (15)$$

We check that the determinant is $\neq 0$; in fact $\det(\mathbf{A}) = 13$. The first step in solving these equations is to transpose rows in \mathbf{A} and in \mathbf{b} to bring the largest element in the first column to the A_{00} position.

Notes:

- The x 's are not relabeled by this transposition.
- We choose the row ($n = 1$ — second row) with the largest (in absolute value) element A_{n0} because we are eventually going to divide by it, and want to minimize the accumulation of roundoff error in the floating point arithmetic.

Transposition gives

$$\begin{pmatrix} 3 & 2 & 4 \\ 1 & 0 & 5 \\ 1 & 1 & 6 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4 \\ 0 \\ 2 \end{pmatrix} \quad (16)$$

Now divide row 0 by the new A_{00} (in this case, 3) to get

$$\begin{pmatrix} 1 & 2/3 & 4/3 \\ 1 & 0 & 5 \\ 1 & 1 & 6 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4/3 \\ 0 \\ 2 \end{pmatrix} \quad (17)$$

Subtract row 0 times A_{n0} from rows with $n > 0$

$$\begin{pmatrix} 1 & 2/3 & 4/3 \\ 0 & -2/3 & 1/3 \\ 0 & 1/3 & 14/3 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 4/3 \\ -4/3 \\ 2/3 \end{pmatrix} \quad (18)$$

Since $|A_{11}| > |A_{21}|$ we do not bother to switch rows 1 and 2, but divide row 1 by $A_{11} = -2/3$, getting

$$\begin{pmatrix} 1 & \frac{2}{3} & \frac{4}{3} \\ 0 & 1 & -\frac{1}{2} \\ 0 & \frac{1}{3} & \frac{4}{3} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \frac{4}{3} \\ 2 \\ \frac{1}{3} \end{pmatrix} \quad (19)$$

We now multiply row 1 by $A_{21} = 1/3$ and subtract it from row 2, and also divide through by A_{22} to get

$$\begin{pmatrix} 1 & \frac{2}{3} & \frac{4}{3} \\ 0 & 1 & -\frac{1}{2} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \frac{4}{3} \\ 2 \\ 0 \end{pmatrix} \quad (20)$$

The resulting transformed system of equations now has 0's to the left and below the principal diagonal, and 1's along the diagonal. Its solution is almost trivial, as can be seen by actually writing out the equations:

$$1x_0 + \frac{2}{3}x_1 + \frac{4}{3}x_2 = \frac{4}{3} \quad (21.0)$$

$$0x_0 + 1x_1 + (-\frac{1}{2})x_2 = 2 \quad (21.1)$$

$$0x_0 + 0x_1 + 1x_2 = 0 \quad (21.2)$$

That is, from Eq. 21.2, $x_2 = 0$. We can back-substitute this in 21.1, then solve for x_1 to get

$$x_1 = 2 - (-\frac{1}{2}) \cdot (0) = 2$$

and similarly, from 21.0 we find

$$x_0 = \frac{4}{3} - \frac{2}{3}(2) + \frac{4}{3}(0) = 0.$$

We test to see whether this is the correct solution by direct trial:

$$\begin{aligned} 1 \cdot 0 + 0 \cdot 2 + 5 \cdot 0 &= 0 \\ 3 \cdot 0 + 2 \cdot 2 + 4 \cdot 0 &= 4 \\ 1 \cdot 0 + 2 \cdot 2 + 6 \cdot 0 &= 2 \end{aligned}$$

This works – we have indeed found the solution.

How much time is needed to perform pivotal elimination? We concentrate on the terms that dominate as $N \rightarrow \infty$.

The pivot has to be found once for each row; this takes $N-k$ comparisons for the k 'th row. Thus we make $\approx N/2$ comparisons of 2 real numbers. (For complex matrices we compare the squared moduli of 2 complex numbers, requiring two multiplications and an addition for each modulus.)

We have to divide the k 'th (pivot) row by the pivot, at a point in the calculation when the row contains $N-k$ elements that have not been reduced to 0. We have to do this for $k=0, 1, \dots, N-1$, requiring $\approx N^2/2$ divisions.

The back-substitution requires 0 steps for x_{N-1} , 1 multiplication and 1 addition for x_{N-2} , 2 each for x_{N-3} , etc. That is, it requires

$$\sum_{k=N-1}^{k=0} (N-1-k) \approx N^2/2$$

multiplications and additions.

The really time-consuming step is multiplying the k 'th row by A_{jk} , $j > k$, and subtracting it from row j . Each such step requires $N-k$ multiplications and subtractions, for $j=k+1$ to $N-1$, or $(N-k) \cdot (N-k-1)$ multiplications and subtractions. This has to be repeated for $k=0$ to $N-2$, giving approximately $N^3/3$ multiplications and subtractions. In other words, the leading contribution to the time is $\tau N^3/3$, which is a lot better than $\tau eN!$ as with Cramer's rule.

When we optimize for speed, only the innermost loop — requiring $O(N^3/3)$ operations needs careful tuning; the $O(N^2/2)$ operations — comparing floating point numbers, dividing by the pivot, and back-substituting — need not be optimized because for large N they are overshadowed by the innermost loop⁹.

9. The exception to this general rule, where more complete optimization would pay, would be an application that requires solving many sets of equations of relatively small order.

9.3 Testing

Since we have worked out a specific 3×3 set of linear equations, we might as well use it for testing and debugging. Begin with some test words that do the following jobs:

- Create matrix and vector (inhomogeneous term)
- Initialize them to the values in the example
- Display the matrix at any stage of the calculation
- Display inhomogeneous term at any stage

```
\ Display words
\ V{ or M{{ stands for what an array puts on stack
GU: G. F. X. ;
: .M      ( M{{ -- )  FINIT DUP
      D.LEN 0 DO CR DUP
      D.LEN 0 DO DUP  ( -- M{{ M{{ )
                          J}{1} } DUP>R G@ G.
      LOOP
      LOOP DROP ;

: .V      ( V{ -- ) DUP D.LEN
      0 DO CR DUP 1}{0} G@ G. LOOP DROP ;
```

We define a word to put ASCII numbers into matrices:

```
: GET-F# BL TEXT PAD $->F ;
```

This word takes the next string¹⁰ from the input stream (set off by ASCII blank, BL) and uses the HS/FORTH word \$->F to convert the string to floating point format and put it on the 87stack.

10. The word TEXT inputs a counted string, using the FORTH-79 standard word WORD, and places it at PAD. This is why PAD appears in the definition of GET-F#. A definition of TEXT might be : TEXT (delimiter --) WORD DUP C@ 1+ PAD SWAP CMOVE ; Note that all words prefaced with C are byte operations by FORTH convention.

GET-F# is used in the following:

```
: <DO-VEC> 0 DO GET-F# DUP 10} G!
      LOOP DROP ;
: TAB->VEC DUP D.LEN <DO-VEC> ;
: TAB->MAT DUP D.LEN DUP * <DO-VEC> ;
```

The file EX.3 will be found in the accompanying program diskette. The explanatory notes below refer to EX.3.

Notes:

- The word .(emits everything up to a terminating) .
- HS/FORTH, because of its segmented dictionary, uses a word **TASK** to define a task-name, and **FORGET-TASK** to **FORGET** everything following the task-name. The FORTH word **FORGET** fails in HS/FORTH when it has to reach too deeply into the dictionary.
- Ex.3 uses the word }row{ defined below.

This is what it looks like when we load EX.3:

FLOAD EX.3 Loading EX.3

Read a dimension 3 vector and 3x3 matrix from a table, then display them.

Now displaying vector:

```
V{ .V CR
0.0000000
4.0000000
2.0000000
```

Now displaying matrix:

```
A{{ .M CR
1.0000000 0.0000000 5.0000000
3.0000000 2.0000000 4.0000000
1.0000000 1.0000000 6.0000000
```

say EX.3 FORGET-TASK to gracefully FORGET these words ok

§§4 Implementing pivotal elimination

Now we have to define the FORTH words that will implement this algorithm. In pseudocode, we can express pivotal elimination as shown in Fig. 9-1 below.

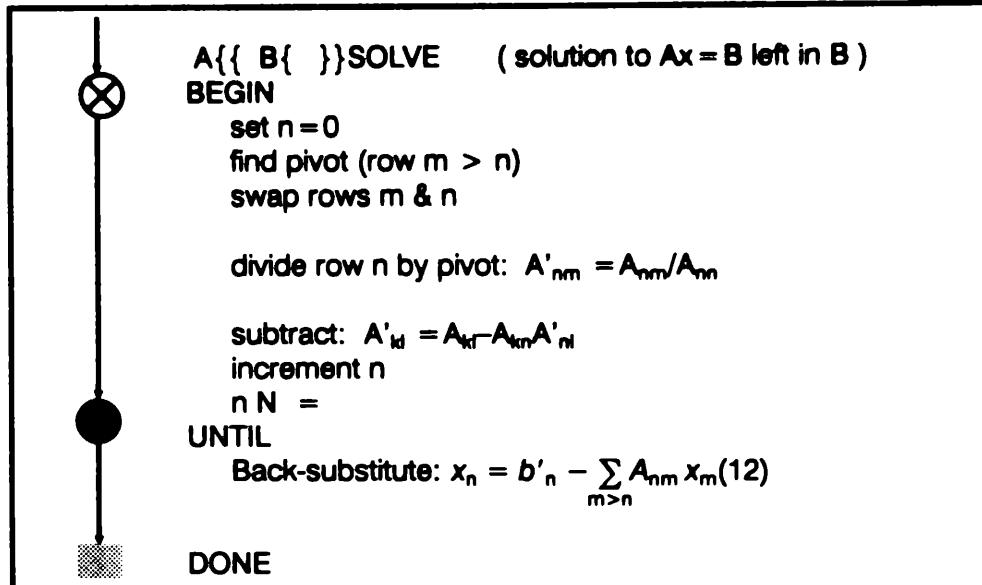


Fig. 9-1 Pseudocode for pivotal elimination

§§5 The program

Whenever we write a complex program we are faced with the problem "Where do we begin?" FORTH emphasizes the construction of components, and generally cares little which component we define first.

For example, we must swap two rows of a matrix. The most obvious way moves data:

```

row1      ->      temp
row2      ->      row1
temp      ->      row2

```

Although data moves are relatively fast, up to $N^2/2$ swaps may be needed, each row containing N data; row-swapping is an $O(N^3)$ operation which must be avoided. Indirection, setting up a list of row pointers and exchanging those, uses $O(N^2)$ time.

We anticipate computing the address of an element of $A\{\}$ in a DO loop via the phrase

$A\{\{ J \} I \}$ (J is row, I is col)

Suppose we have an array of row pointers and a word }row{ that looks them up. Then the (swapped) element would be

$A\{\{ J \} \text{row}\{ I \}\}$.

To implement this syntax we define¹¹

```
: swapper      ( n -- )
    CREATE 0 DO I , LOOP
    DOES> OVER + + @ ;
    \ this is a defining word
```

We would define the array of row indices via

$A\{\{ D.\text{LEN} \text{ swapper } \} \text{row}\{ \ \backslash \text{create array } \} \text{row}\{$

We have initialized the vector }row{ by filling it with integers from 0 to N-1 while defining it.

Suppose we wanted to re-initialize the currently vectored row-index array: this is accomplished via

$A\{\{ D.\text{LEN} ' \} \text{row}\{ \text{ refill}$

where

```
: refill ( n adr -- )
    SWAP 0 DO I 2* DDUP + !
    2 +LOOP DROP;
```

11. The HS/FORTH words **VAR** and **IS** define a data structure that can be changed, like a FORTH VARIABLE: **0 VAR X 3 IS X** but has the run-time behavior of a CONSTANT: **X .3 ok.**

To swap the two rows

```
: ADRS >R 2* OVER + SWAP R > 2* + ;
( a m n - a + 2m a + 2n)
0 VAR ?SWAP \ to keep track of swaps

: )SWAP ( a m n - ) DDUP =
  IF      DDROP DROP \ no swap - clean up
  ELSE   ADRS DDUP      ( - 1 2 1 2 )
        @ SWAP @          ( - 1 2 [2] [1])
        ROT ! SWAP !
        -1 ?SWAP XOR IS ?SWAP \ ~ ?SWAP
  THEN ;
```

Test this with a 3-dimensional row-index array:

```
3 swapper }row{
: TEST 0 DO 1 }row{ CR . LOOP ;
3 TEST
0
1
2 ok
```

Now swap rows 1 and 2:

```
' }row{ 1 2 }SWAP 3 TEST
0
2
1 ok
```

and back again:

```
' }row{ 1 2 }SWAP 3 TEST
0
1
2 ok
```

Next, we need a word to find the pivot element in a given column, searching from a given n. The steps in this procedure are shown in Fig. 9-2 on page 230 below.

We anticipate using generic operations **Gx** defined in Chapter 5 to perform fetch, store and other useful manipulations on the matrix elements, without specifying their type until run-time. It

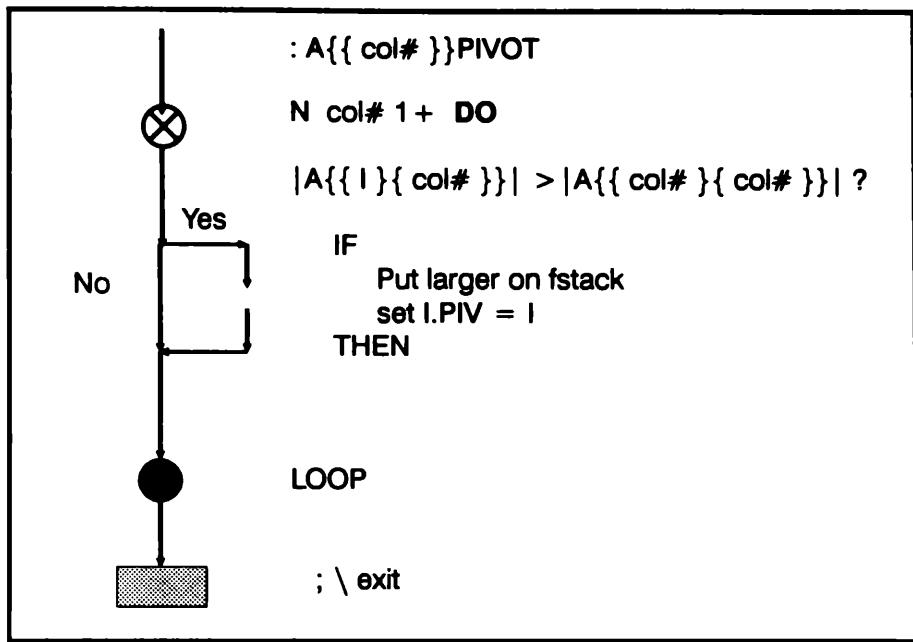


Fig. 9-2 Pseudocode for finding the pivot element

is thus useful to have a place to store the type (other than the second cell of the array data structure). We arrange this *via*

0 VAR T \ a place to store data type

The rest of the definition is rendered self-explanatory by the vertical commenting style (a must for long¹² words):

```

0 VAR Length \ length of matrix
0 VAR COL \ current col#
0 VAR I.PIV \ I.PIV is used to return the result
0 VAR a{{ \ a{{ stores the address of M{{

: INITIALIZE ( M{{ -- :: - )
IS a{{}
a{{ type@ IS T \ initialize T
a{{ LEN@ IS Length ; \ length -> Length

```

12. While Brodie (*TF*, p.180) quotes Charles Moore, FORTH's inventor, as offering 1-line definitions as the goal in FORTH programming, sometimes this is just not possible.

```

: } } PIVOT      ( M{ { col -- :: -- } }
IS COL  COL IS I.PIV
INITIALIZE
a{ { COL }row{ COL } }   ( -- seg.off[M{ { col,col } } ] t )
>FS GABS FS>F        ( 87: -- | 1st_elt |
R>  COL 1+             \ loop limits: L, COL + 1
DO                      \ begin loop
a{ { 1 }row{ COL } }   ( -- seg.off[M{ { i,col } } ] t )
>FS GABS FS>F        ( 87: -- | old_elt | new_elt |
XDUP F<               \ test if new_elt > old_elt
IF FSWAP I IS I.PIV   THEN FDROP
LOOP                   \ end loop
FDROP :                \ clean up fstack

```

\ Usage: A{ { 2 } }PIVOT

Notes:

- We avoid filling up the parameter stack with addresses that have to be juggled, by putting arguments in named storage locations (VARs). We anticipate setting all the parameters in the beginning using INITIALIZE which can be extended later if necessary.
- We have used a word from the COMPLEX arithmetic lexicon, namely XDUP (FOVER FOVER) to double-copy the contents of the fstack, since the test F< drops both arguments and leaves a flag.
- There is little to be gained by optimizing (and if we did we should have had to avoid the generic Gx words because they cannot be optimized by the HS/FORTH recursive-descent optimizer) because only $N^2/2$ time is used, negligible (for large N) compared with the $N^3/3$ in the innermost loop.

The Gx operations are found in the file GLIB.FTH.

To test the word } }PIVOT we can add some lines to the test file:

```

CR CR .( find the pivot row for columns 0, 1, 2 )
CR
CR .(A{ { 0 } }PIVOT IPIV .) BL EMIT A{ { 0 } }PIVOT IPIV .
CR .(A{ { 1 } }PIVOT IPIV .) BL EMIT A{ { 1 } }PIVOT IPIV .
CR .(A{ { 2 } }PIVOT IPIV .) BL EMIT A{ { 2 } }PIVOT IPIV .
CR CR

```

The result is the additional screen output

```
A{{ 0 }}PIVOT IPIV 1
A{{ 1 }}PIVOT IPIV 1
A{{ 2 }}PIVOT IPIV 2
ok
```

From our pseudocode expression (see Fig. 9-1, p. 227) of the pivotal elimination algorithm we see that the next operation is to multiply a row by a constant. To do this we define

```
0 VAR ISTEP
  \ include phrase
  \ T #BYTES DROP IS ISTEP
  \ in INITIALIZE

: DO(ROW*X)LOOP  ( seg off l.fin l.beg -- :: x - - x)
  DO DDUP I + DDUP T >FS G*NP T FS>
  ISTEP /LOOP DDROP ;
  \ G*NP means “(generic) multiply, no pop”

: } }ROW*X      ( M{{ row - :: x - x )
  UNDER }row{ 0 } }      ( -- r seg off t )
  DROP ROT      ( - -seg off r )
  ISTEP * Length ISTEP * SWAP \ loop indices
  DO(ROW*X)LOOP ;      \ loop

\ Ex: A{{ 2 }}ROW*X
```

Notes:

- While **DO(ROW*X)LOOP** and **} }ROW*X** clear the parameter stack in approved FORTH fashion, they leave the constant multiplier **x** on the fstack. That is, we anticipate next multiplying the corresponding row of the inhomogeneous term **b** (in the matrix equation **A · x = b**) by the same constant **x**.
- We assume **INITIALIZE** (including **INIT.ISTEP**) has been invoked before **} }PIVOT** or **} }ROW*X**.
- Anticipating the (possible) need to optimize, we factor the loop right out of the word. We also compute the addresses fast by putting base addresses on the stack and then incrementing them by the cell-size, in bytes.

Subtracting row I (times a constant) from row J is quite similar to multiplying a row by a constant. The process can be broken down into the pseudocoded actions in Fig. 9-3 below.

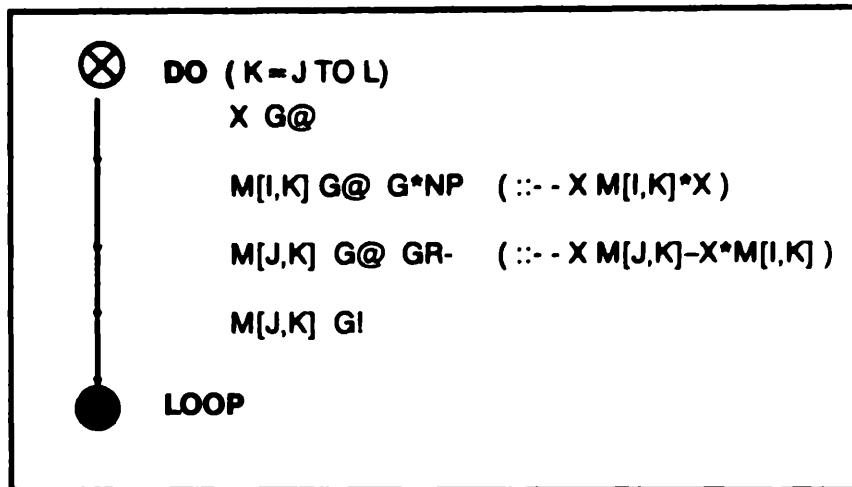


Fig. 9-3 Steps in } } R1-R2*X

A first attempt might look as follows:

```

\ auxiliary words
:4DUP DOVER DOVER ;
(a b c d -- a b c d a b c d)
: + + (s1.o1 s2.o2 n - s1.o1 + n s2.o2 + n)
DUP>R + ROT R> + -ROT ;

: } } R1-R2*X (r1 r2 -- :: x - - x)
>R >R
a{{ R> }row{ 0 }} DROP \ M{{ r1 0 }}
a{{ R@ }row{ 0 }} DROP \ M{{ r2 0 }}
Length ISTEP * \ L*ISTEP (upper limit)
R> ISTEP * \ r2*ISTEP (lower limit)
DO 4DUP I ++ \ duplicate & inc base addrs
T>FS G*N P ( :: - x x*M[r2,k] )
DDUP T>FS GR-
T>FS \ ! result
ISTEP /LOOP DDROP DDROP ;
\ INITIALIZE is assumed
    
```

As with } } ROW*X we avoid the execution-speed penalty of calculating addresses within the loop by *not* using the phrase

```
a{{ I }row{ J }} .
```

However, we are doing a lot of unnecessary work inside the loop by making 5 choices based on datatype. There are also a lot of unnecessary moves to/from the ifstack. This is an example where speed has been sacrificed to compactness of code: one program solves equations of any ("scientific") data format – REAL*4, REAL*8, COMPLEX*8 and COMPLEX*16.

Conversely, we can both eliminate the extra work and accelerate execution simply by defining a *separate* inner loop for each datatype, letting the choice take place once, outside the inner loop. This multiplies the needed code fourfold (or more-fold, if we optimize).

The 4 inner loops are

```
: Re.LP  ( s1.o1 s2.o2 L*iStep r2*iStep -- :: x--x)
      FS>F >R
      DO 4DUP I ++ R32@L F*NP
          DDUP R32@L FR- R32!L
      4 /LOOP    R> F>FS ;

: DRe.LP ( s1.o1 s2.o2 L*iStep r2*iStep -- :: x--x)
      FS>F >R
      DO 4DUP I ++ R64@L F*NP
          DDUP R64@L FR- R64!L
      8 /LOOP    R> F>FS ;

: X.LP   ( s1.o1 s2.o2 L*iStep r2*iStep -- :: x--x)
      FS>F >R
      DO 4DUP I ++ CP@L X*NP
          DDUP CP@L CPR- CPI!
      8 /LOOP    R> F>FS ;

: DX.LP  ( s1.o1 s2.o2 L*iStep r2*iStep -- :: x--x)
      FS>F >R
      DO 4DUP I ++ DCP@L X*NP
          DDUP DCP@L CPR- DCPI!
      16 /LOOP   R> F>FS ;
```

Not surprisingly, the loops contain some duplicate code, but this is a small price to pay for the speed increase. A significant further

increase can be obtained easily, using the HS/FORTH recursive-descent optimizer to define these inner loops, or by redefining the loops in assembler (for ultimate speed).

Now we can redefine $\{\}R1-R2*X$ using vectored execution, via HS/FORTH's CASE: ... ;CASE construct, or the equivalent high-level version given here:

```

: CASE: CREATE ]
    DOES> OVER + + @ EXECUTE ;
: ;CASE [COMPILE] ; ; IMMEDIATE

CASE: DO(R1-R2*X)LOOP
    Re.LP  DRe.LP  X.LP  DX.LP  ;CASE
    : } } R1-R2*X  ( r1 r2 - - :: x - - x )
        > R > R
        a{{ R > }row{ 0 }} DROP          \ M{{ r1 0 }}
        a{{ R@ }row{ 0 }} DROP          \ M{{ r2 0 }}
        Length I.STEP *             \ L*I.STEP (upper limit)
        R > I.STEP *              \ r2*I.STEP (lower limit)
        T  DO(R1-R2*X)LOOP  DDROP DDROP ;
    \ We assume INITIALIZE has been invoked

```

Another word is needed to perform the same manipulation on the inhomogeneous term. Since this latter process runs in $O(N^2)$ time we need not concern ourselves unduly with efficiency.

```

: }V1-V2*X      ( r1 r2 - - :: x - - )
    SWAP >R >R
    b{ 0.0 } DROP DDUP  \ V[0] V[0]
    R > }row{ ISTEP * + >FS G*
    R > }row{ ISTEP * + DDUP >FS GR- FS > ;

```

Note:

- After $\{\}V1-V2*X$ executes, the multiplier x is no longer needed, so we drop it here by using G^* rather than G^*NP .

We now combine the words $\{\}SWAP$, $\{\}PIVOT$, $\{\}ROW*X$, $\{\}R1-R2*X$ and $\{\}V1-V2*X$ to implement the triangularization portion of pivotal elimination.

Since the Gaussian elimination method makes it very easy to compute the determinant of the matrix as we go, we might as well do so, especially as it is only an $O(N)$ process. By evaluating the determinants of simple cases, we realize the determinant is simply the product of all the pivot elements, multiplied by $(-1)^{\#swaps}$. Hence we need to keep track of swaps, as well as to multiply the determinant (at a given stage) by the next pivot element. The need to do these things has been anticipated in defining }SWAP above.

Computing the determinant also lets us test as we go, that the equations can be solved: if at any stage the determinant is zero, (at least) two of the equations must have been equivalent. Should it be necessary to test the condition¹³ of the equations, this too can be found as we proceed, by computing the determinant.

Here is a simple recipe for computing the determinant, with checking to be sure it does not vanish identically. We use the intelligent fstack (ifstack) defined in Ch. 7.

```

DCOMPLEX SCALAR DET      \ room for any type
: INIT_DET      T 'DET !
T G=1  DET G! ;          \ set Type
                           \ set det = 1

% 1.E-10 FCONSTANT CONDITION

: DETERMINANT ( -- ::x--x )
DET >FS  G*NP
?SWAP IF  GNEGATE THEN
FS.DUP  GABS FS>F CONDITION F<
ABORT" determinant too small!"  DET FS> ;

```

13. The condition of a system of linear equations refers to how accurately they can be solved. Equations can be hard to solve precisely if the inverse matrix A^{-1} has some large elements. Thus, the error vector for a calculated solution, $\delta = b - A \cdot x_{\text{Calc}}$ can be small, but the difference between the exact solution and the calculated solution can be large, since (see §9§3 below)

$$x_{\text{Exact}} - x_{\text{Calc}} = A^{-1} \cdot \delta.$$

A test for ill-condition is whether the determinant gets small compared with the precision of the arithmetic. See, e.g., A. Ralston, *A First Course in Numerical Analysis* (McGraw-Hill Book Co., New York, 1965).

Now we define the word that triangularizes the matrix:

```

0 VAR b{           \ to keep the stack short
: INITIALIZE      ( M{{ V{ -- :: -- )
IS b{
IS a{{{
a{{ D.TYPE IS T
a{{ D.LEN IS Length
a{{ D.TYPE #BYTES DROP IS ISTEP
INIT.DET ;        \ set det = 1

: }/PIVOT }row{ } DROP DDUP T >FS G* T FS>;
( seg off t-- :: x-- )

: TRIANGULARIZE ( M{{ V{ -- )
INITIALIZE
Length 0 DO          \ loop 1 - by rows
    a{{ I }}PIVOT      \ find pivot in col I
    ' }row{ I I.PIV }SWAP \ exchange rows
    a{{ I }}row{ I }) >FS \ pivot- > ifstack
    DETERMINANT        \ calc det
    1/G a{{ I }}ROW*X   \ row I /pivot
    b{ I }/PIVOT        \ inhom. term /pivot
    Length 1 I+ DO      \ loop 2 - by rows
        a{{ I }}row{ J }) >FS \ x -> ifstack
        a{{ I J }}R1-R2*X
        \ row[i] = row[i]-row[j]*x
        b{ I J }V1-V2*X
        \ same for b{ and drop x
    LOOP                \ end loop 2
    LOOP ;              \ end loop 1
\ Usage: A{{ B{ TRIANGULARIZE

```

Now at last we can back-solve the triangularized equations to find the unknown x 's. The word for this is }BACK-SOLVE, defined as follows:

```

: }BACK-SOLVE      ( -- )
  0 LENGTH 2- DO
    T G=0
    LENGTH I 1+ DO
      a{{ J }row{ I }} >FS
      b{ I }row{ } >FS G* G+
      LOOP
      b{ I }{ } DROP DDUP T >FS
      GR- T FS>
      -1 +LOOP ;
  ] outer loop
  ] inner loop
  ] inner loop
] outer loop

```

Putting the entire program together we have the linear equation solver given in the file SOLVE1. Examples of solving dense 3×3 and 4×4 systems are included for testing purposes.

§§6 Timing

We should like to know how much time it will take to solve a given system. (Of course it is also useful to know whether the solution is correct!) We time the solution of 4 sets of N equations, with 4 different values of N. The running time can be expressed as a cubic polynomial in N with undetermined coefficients:

$$T_N = a_0 + a_1 N^1 + a_2 N^2 + a_3 N^3 \quad (19)$$

Evaluating 19 for 4 different values of N, and measuring the 4 times T_{N_i} , we produce 4 inhomogeneous linear equations in 4 unknowns: a_i , $i = 0, 1, 2, 3$. As luck would have it, we just happen to have on hand a linear equation solver, and are thus in a position to determine these coefficients numerically.

For this timing and testing chore we need an exactly soluble dense system of linear equations, of arbitrary order. The simplest such system involves a rank-1 matrix, that is, a matrix of the form¹⁴:

14. We employ the standard notation that a vector u is a column and an adjoint vector v^\dagger is a row. Their outer product, uv^\dagger , is a matrix. Given a column v , we construct its adjoint (row) v^\dagger by taking the complex conjugate of each element and placing it in the corresponding position in a row-vector.

$$\mathbf{A} = \mathbf{I} - \mathbf{u} \mathbf{v}^\dagger \quad (20)$$

In terms of matrix elements,

$$A_{ij} = \delta_{ij} - u_i v_j^* \quad (20')$$

The solution of the system $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ is simple: using the standard notation

$$\mathbf{v}^\dagger \cdot \mathbf{x} \equiv (\mathbf{v}, \mathbf{x}) = \sum_i v_i^* x_i \quad (21)$$

we have

$$x_i = b_i + (\mathbf{v}, \mathbf{x}) u_i \quad (22)$$

The coefficient (\mathbf{v}, \mathbf{x}) is just a (generally complex) number; we compute it from 22 via

$$(\mathbf{v}, \mathbf{x}) = (\mathbf{v}, \mathbf{b}) + (\mathbf{v}, \mathbf{u}) (\mathbf{v}, \mathbf{x}) \quad (23)$$

or

$$(\mathbf{v}, \mathbf{x}) = \frac{(\mathbf{v}, \mathbf{b})}{1 - (\mathbf{v}, \mathbf{u})} . \quad (24)$$

Substituting 24 back in 22, we have

$$x_i = b_i + \frac{(\mathbf{v}, \mathbf{b})}{1 - (\mathbf{v}, \mathbf{u})} u_i \quad (25)$$

Everything in 25 is determined, hence the solution is known in closed form.

An example that embodies the above idea is given in the file EX.20, where we make the special choices

$$\mathbf{u} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ \dots \\ \dots \\ \dots \end{pmatrix}, \quad \mathbf{v} = \frac{1}{2N} \mathbf{u}, \quad \mathbf{b} = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ \dots \\ \dots \end{pmatrix}, \quad \mathbf{x} = \begin{pmatrix} 1.5 \\ 0.5 \\ 1.5 \\ 0.5 \\ \dots \\ \dots \end{pmatrix}, \quad N \text{ even}$$

We give the times only for the case of a highly optimized inner loop (written in assembler), for the type **REAL*4**:

Table 9-1 Execution times for various N (9.54 MHz 8086 + 8087 machine)

N	Time
20	1.20
50	7.75
75	19.6
100	38.8

The coefficients of the cubic polynomial extracted from the above are (in seconds, 9.54 MHz 8086 machine, machine-coded inner loop)

$$\begin{aligned} a_0 &= 0.32727304 \\ a_1 &= -0.01084850 \\ a_2 &= 0.00241636 \\ a_3 &= 0.00001539 \end{aligned}$$

from them we can extrapolate the time to solve a 350×350 real system to be about 16 minutes. On a 25 MHz 80386/80387 machine with 32-bit addressing implemented, the time should decrease five- to tenfold¹⁵.

It is interesting to explore a bit further the extracted value of a_3 , which is $\frac{1}{3}$ the time needed to evaluate the innermost loop (defined above as **Re.LP** and hand-coded in assembler for ultimate speed). Converting to clock cycles on an IBM-PC compatible (running at 9.54 MHz) we have an average of 440 clock cycles per traversal (of this loop). Recall the operations that are needed: a 32-bit memory fetch, a floating-point multiply, another 32-bit memory fetch, a floating-point subtraction, and a 32-bit store. The initial fetch-and-multiply can be compressed into a single co-processor operation, as can the fetch-and-subtract. The times in cycles for these basic operations are¹⁶

Table 9-2 Execution times of innermost loop operations

Operation	Time (cpu clocks)
memory FMUL	133
memory FSUBR	128
memory FSTP	100
overhead	61
Total	422

-
- 15. When I think that solving 100×100 systems – key to my Ph.D. research in 1966 – took the better part of an hour on an IBM 7094, these results seem incredible.
 - 16. See 8037P.

We see from Table 9-2 above that the time computed from the CPU and coprocessor specifications (422 clocks) is close to the measured time (440 clocks). The slight difference doubtless comes both from measurement error and from the fact that the timing of CISC chips is not an exact art (for example, there are periodic interruptions for dynamic memory refreshment and for the system clock).

Finally, we confirm that little is to be gained by optimizing outer loops. Suppose, e.g., we could halve a_2 by clever programming; then we should cut the $N = 350$ time from 16 to 13 minutes, and the time for $N = 1000$ by some 20 minutes in 5 hours, or 6%.

§3 Matrix inversion

We introduce this subject with a brief discourse on linear algebra of square matrices.

§§1 Linear transformations

Suppose A is a square ($N \times N$) matrix and x is an N -dimensional vector (a column, or $N \times 1$ matrix). We can think of the symbolic operation

$$y = A \cdot x \quad (26)$$

as a linear transformation of the column x to a new column y . In terms of matrix elements and components,

$$y_m = \sum_{n=0}^{N-1} A_{mn} x_n \quad (27)$$

The transformation is linear because if x is the sum of 2 columns (added component by component)

$$x = x^{(1)} + x^{(2)}, \quad (28)$$

we can calculate $A \cdot x$ either by first adding the two vectors and then transforming, written

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{A} \cdot (\mathbf{x}^{(1)} + \mathbf{x}^{(2)}) \quad (29)$$

or we could transform first and then add the transformed vectors.
The identity of the results is called the **distributive law**

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{A} \cdot (\mathbf{x}^{(1)} + \mathbf{x}^{(2)}) = \mathbf{A} \cdot \mathbf{x}^{(1)} + \mathbf{A} \cdot \mathbf{x}^{(2)} \quad (30)$$

of linear transformations.

§§2 Matrix multiplication

Now, suppose we had several square matrices, \mathbf{A} , \mathbf{B} , \mathbf{C}, \dots . We could imagine performing successive linear transformations on a vector \mathbf{x} via

$$\mathbf{y} = \mathbf{A} \cdot \mathbf{x} \quad (31a)$$

$$\mathbf{z} = \mathbf{B} \cdot \mathbf{y} \quad (31b)$$

$$\mathbf{w} = \mathbf{C} \cdot \mathbf{z} \quad (31c)$$

These can conveniently be written

$$\mathbf{w} = \mathbf{C} \cdot \left(\mathbf{B} \cdot (\mathbf{A} \cdot \mathbf{x}) \right). \quad (32)$$

The concept of successive transformations leads to the idea of multiplying two matrices to obtain a third:

$$\mathbf{D} = \mathbf{B} \cdot \mathbf{A} \quad \mathbf{E} = \mathbf{C} \cdot \mathbf{D} \quad (33)$$

In terms of matrix elements we have, for example

$$D_{ik} = \sum_{j=0}^{N-1} B_{ij} A_{jk} \quad (34)$$

The important point is that the (matrix) multiplications may be performed in any order, so long as the left-to-right ordering of the factors is maintained:

$$\mathbf{C} \cdot (\mathbf{B} \cdot \mathbf{A}) \equiv (\mathbf{C} \cdot \mathbf{B}) \cdot \mathbf{A} \quad (35)$$

Equation 35 is known as the **associative law** of matrix multiplication. Finally, we note that —as hinted above— the left-to-right order of factors in matrix multiplication is significant. That is, in general,

$$\mathbf{A} \cdot \mathbf{B} \neq \mathbf{B} \cdot \mathbf{A} \quad (36)$$

We say that, unlike with ordinary or even complex arithmetic, matrix multiplication —even of square matrices— does not in general obey the commutative law¹⁷.

§§3 Matrix Inversion

With this introduction, what does it mean to invert a matrix? First of all, the concept can apply only to a square matrix. Given an $N \times N$ matrix \mathbf{A} , we seek another $N \times N$ matrix \mathbf{A}^{-1} with the property that

$$\mathbf{A}^{-1} \cdot \mathbf{A} \equiv \mathbf{A} \cdot \mathbf{A}^{-1} = \mathbf{I} \quad (37)$$

where \mathbf{I} is the unit matrix defined in the beginning of this chapter (Eq. 4 — 1's on the main diagonal, 0's everywhere else). We have implied in Eq. 37 that a matrix that is an inverse with respect to left-multiplication is also an inverse with respect to right-multiplication. Put another way, we imply that

$$(\mathbf{A}^{-1})^{-1} \equiv \mathbf{A} \quad (38)$$

The condition that —given \mathbf{A} — we can construct \mathbf{A}^{-1} is the same as the condition that we should be able to solve the linear equation

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b};$$

namely, $\det(\mathbf{A}) \neq 0$.

§§4 Why invert matrices, anyway?

We have developed a linear equation solver program already – why should we be interested in a matrix inversion program?

Here is why: The time needed to solve a single system (that is, with a given inhomogeneous term) of linear equations is conditioned by the number of floating-point multiplications required: about $N^3/3$. The number needed to invert the matrix is about N^3 , roughly $3 \times$ as many, meaning roughly $3 \times$ as long to invert as to solve, if the matrix is large. Clearly there is no advantage to inverting unless we want to solve a number of equations with the same coefficient matrix but with different inhomogeneous terms. In this case, we can write

$$\mathbf{x} = \mathbf{A}^{-1} \cdot \mathbf{b} \tag{39}$$

and just recalculate for each \mathbf{b} . Clearly this breaks even – relative to solving 3 sets of equations – for 3 different \mathbf{b} 's and is superior to re-solving for more than 3 \mathbf{b} 's.

§§5 An example

Let us now calculate the inverse of our 3×3 matrix from before.

The equation is (let $\mathbf{C} = \mathbf{A}^{-1}$ be the inverse¹⁸)

$$\begin{pmatrix} 1 & 0 & 5 \\ 3 & 2 & 4 \\ 1 & 1 & 6 \end{pmatrix} \begin{pmatrix} C_{00} & C_{01} & C_{02} \\ C_{10} & C_{11} & C_{12} \\ C_{20} & C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{40}$$

18. Note: if \mathbf{C} is a right-inverse, $\mathbf{AC} = \mathbf{I}$, it is also a left-inverse, $\mathbf{CA} = \mathbf{I}$.

It is easy to see that Eq. 40 is like 3 linear equations, the unknowns being each column of the matrix \mathbf{C} . The brute-force method to calculate \mathbf{A} is then to work on the right-hand-side all at once and we triangularize, and back-solve. It is easy to see that triangularization by pivotal elimination leads to

$$\begin{pmatrix} 1 & \frac{v_3}{3} & \frac{v_3}{3} \\ 0 & 1 & -\frac{v_2}{2} \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} C_{00} & C_{01} & C_{02} \\ C_{10} & C_{11} & C_{12} \\ C_{20} & C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} 0 & v_3 & 0 \\ -\frac{v_2}{2} & v_2 & 0 \\ v_{13} & -v_{13} & \frac{v_{13}}{3} \end{pmatrix} \quad (41)$$

To construct the inverse we replace the right-hand side with the results of back-solving, column by column. This is N times as much work as back-solving a single set of equations, hence the total time required for brute-force inversion is $\approx \frac{4}{3}N^3$. By keeping track of zero elements we could reduce this time by another $\frac{1}{3}N^3$, thereby obtaining the theoretical minimum (for Gaussian elimination) of $O(N^3)$. However, the brute-force method is unsatisfactory for another reason: it takes twice the storage of more sophisticated algorithms. Modern matrix packages therefore use LU decomposition for both linear equations and matrix inversion.

§4 LU decomposition

We now investigate the LU decomposition algorithm¹⁹. Suppose a given matrix \mathbf{A} could be rewritten

$$\mathbf{A} = \begin{pmatrix} a_{00} & a_{01} & \dots \\ a_{10} & a_{11} & \dots \\ \dots & \dots & \dots \end{pmatrix} = \mathbf{L} \cdot \mathbf{U} = \begin{pmatrix} \lambda_{00} & 0 & 0 \\ \lambda_{10} & \lambda_{11} & 0 \\ \dots & \dots & \dots \end{pmatrix} \begin{pmatrix} \mu_{00} & \mu_{01} & \dots \\ 0 & \mu_{11} & \dots \\ 0 & 0 & \dots \end{pmatrix} \quad (42)$$

then the solution of

19. See, e.g., W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vetterling, *Numerical Recipes* (Cambridge University Press, Cambridge, 1986), p. 31ff.

$$(L \cdot U) \cdot x = L \cdot (U \cdot x) = b \quad (43)$$

can be found in two steps: First, solve

$$L \cdot y = b \quad (44)$$

for

$$y = U \cdot x \quad (45)$$

via

$$\begin{aligned} \lambda_{00} y_0 &= b_0 \\ \lambda_{10} y_0 + \lambda_{11} y_1 &= b_1 \\ \lambda_{20} y_0 + \lambda_{21} y_1 + \lambda_{22} y_2 &= b_2 \\ \dots \text{etc.} \dots \end{aligned} \quad (46)$$

which can be solved successively by forward substitution. Next solve 45 successively (by back-substitution) for x :

$$\begin{aligned} \mu_{N-1 N-1} x_{N-1} &= y_{N-1} \\ \mu_{N-2 N-2} x_{N-2} + \mu_{N-2 N-1} x_{N-1} &= y_{N-2} \\ \dots \text{etc.} \dots \end{aligned} \quad (47)$$

The n 'th term of Eq. 46 requires n multiplications and n additions. Since we must sum n from 0 to $N-1$, we find $N(N-1)/2 \approx N^2/2$ multiplications and additions to solve all of 46. Similarly, solving 47 requires about $N^2/2$ additions and multiplications. Thus, the dominant time in solving must be the time to decompose according to Eq. 42.

The decomposition time is $\approx N^3/3$, and the method for decomposing is described clearly in *Numerical Recipes*. The equations to be solved are

$$\sum_{k=0}^{N-1} \lambda_{mk} \mu_{kn} = A_{mn} \quad (48)$$

constituting $N^2 + N$ equations for N^2 unknowns. Thus we may arbitrarily choose $\lambda_{kk} = 1$.

The equations 48 are easy to solve if we do so in a sensible order.
Clearly,

$$\begin{aligned}\lambda_{mk} &\equiv 0, \quad m > k \\ \mu_{kn} &\equiv 0, \quad k < n\end{aligned}\tag{49}$$

so we can divide up the work as follows: for each n , write

$$\begin{aligned}\mu_{mn} &= A_{mn} - \sum_{k=0}^{m-1} \lambda_{mk} \mu_{kn}, \quad m = 0, 1, \dots, n \\ \lambda_{mn} &= \frac{1}{\mu_{nn}} \left(A_{mn} - \sum_{k=0}^{n-1} \lambda_{mk} \mu_{kn} \right), \quad m = n+1, n+2, \dots, N-1\end{aligned}\tag{50}$$

Inspection of Eq. 50 makes clear that the terms on the right side are always computed before they are needed. We can store the computed elements λ_{mk} and μ_{kn} in place of the corresponding elements of the original matrix (on the diagonals we store μ_{nn} since $\lambda_{kk} = 1$ is known).

To limit roundoff error we again pivot, which amounts to permuting so the row with the largest diagonal element is the current one. Much of the code developed for the Gauss elimination method is applicable, as the file LU.FTH shows.

Strings and I/O in FORTH

Contents

§1 Standard I/O in FORTH	249
§2 Standard I/O in Scientific FORTH	250
§§1 Disk file input	251
§§2 Disk file output	253
§3 Logging screen activity	254
§§1 Redirection by re-vectoring	254
§§2 Redirection using MS-DOS resources	255
§§3 Redirection while DEBUGging	257

This chapter describes some input and output (I/O) functions and string handling facilities needed for scientific computing. We will illustrate with words that capture screen output to a disk file, and with a word that translates a FORTRAN expression – entered from keyboard or read in from a file – to FORTH code.

§1 Standard I/O in FORTH

One area where more conventional languages excel FORTH in convenience for the casual technical user/programmer is standardized I/O. FORTRAN offers all the needed functionality – number formatting, strings, *etc.* – albeit in a somewhat rigid form.

BASIC, a more modern language, offers even more convenient built-in I/O facilities than FORTRAN.

I confess I personally do not find the practices of C or Pascal congenial, but most programmers seem able to live with them.

FORTH possesses no standardized I/O words except for writing to the screen and inputting from the keyboard. Thus there are no

required words for redirecting output to printer, disk file or plotter.

FORTH lacks standard I/O functions for several reasons:

- Historically, FORTH incorporated its own (rather primitive) operating system (OS) on many machines. The aim was compactness rather than sophistication, so it eschewed built-in OS or BIOS ("Basic Input/Output System) functions and/or device drivers.
- FORTH has been implemented on so many machines that hardware independence in I/O has seemed an unattainable goal.

FORTH developers have therefore deemed it better to let each programmer develop his own favorite techniques and specialized device drivers. The result is Bedlam.

§2 Standard I/O in Scientific FORTH

While the freedom to custom tailor I/O is very nice when unusual problems crop up, no one likes having to roll his own subroutines to do routine tasks (95% of all I/O is routine). A scientific FORTH dialect needs to standardize this routine 95%. This book is intended for dialects like HS/FORTH that run under an operating system¹ and are file oriented. Such dialects necessarily contain a reasonable spectrum of words for disk, screen and printer I/O, equivalent to those in HS/FORTH that I will employ in this chapter.

Some FORTHS are screen oriented. That is, they are designed to read and write blocks of 1024 bytes, displayed as 16 lines of 64 characters per line. This was a great idea for primitive machines like the Timex/Sinclair ZX81 or Jupiter Ace, but in my opinion is thoroughly inappropriate for modern work stations with sophisticated displays.

1. Such as MS-DOS, OS/2, Pick, Unix, etc.

I strongly recommend users of screen oriented FORTHS to update to file oriented systems. Failing that, they should consult any of the numerous introductory books, or articles in journals such as *Dr. Dobb's Journal*, for suggestions on using 1K blocks to store and input numbers and text.

§§1 Disk file Input

We now tackle the problem of inputting data from disk files. HS/FORTH provides the words **OPEN-INPUT**, **CLOSE-INPUT**, **<FILE**, **\$"**, **G#**, **G\$**, **NR**, and **\$->F** that will suffice for our needs². Their effects are:

OPEN-INPUT	\ open a file for input, whose name is a string whose address is on the stack.
CLOSE-INPUT	\ close an open input file
<FILE	\ get input from the open file
\$"	\ put a string at PAD (terminate with ")
G#	\ get a 16-bit number from open device
G\$	\ get a counted string from open device (will not read past a CR-LF)
NR	\ skip CR-LF to next record
\$->F	\ convert the string at PAD to a floating point number on the 87stack.

Note: Most operating systems handle sequential files by maintaining a pointer into the file. After a datum is read, the pointer is advanced. Suppose the file *dat.fil* has 3 numbers in it: 17, 32 and -8. Then the phrase

\$" dat.fil" <FILE G#

will place 17 on the stack. A repetition will place the number 32 on the stack, and a third repetition will place -8 on the stack.

-
2. Voltaire once remarked that if Satan did not exist it would be necessary for God to create him. If your FORTH system lacks words with these functions, by all means, invent them. A MS-DOS programmer's manual (see below) is vital for this task.

We shall use the following data file convention for arrays:

- The first number is the type of the array.
- The second is the order N.
- The third is the number of entries— N^2 for a matrix (2ARRAY), N for a vector (1ARRAY).
- One or more ASCII blanks (20h, 32d) separate the numbers in the file.
- The entries in a 2ARRAY are stored row-wise.

These conventions let one word serve for 1ARRAYs and 2ARRAYs.

Coding from the top down, we plan to enter the following phrase to load the file:

```
IN SUPERSEG 100 LONG REAL 1ARRAY A{
A{ $"data.fil" FILL <FILE
```

To minimize our labor let us plan to have the word **FILL <FILE** close *data.fil* after reading it in. For safety we also should check whether the type and order of the array we are filling, and that of the array stored in the file are commensurate.

To check whether the file is too large or of the wrong type, we have

```
0 VAR FILE.TYPE
0 VAR FILE.LEN
: READ.FILE.STATS <FILE
  G# IS FILE.TYPE
  G# IS FILE.LEN ;

: MAT>=FILE?
  DUP D.TYPE OVER D.LEN      ( a{ - a{ )
    FILE.LEN >                ( -- a{ T L )
    SWAP FILE.TYPE =          ( -- a{ T f1 )
    AND NOT                  ( -- a{ f1 f2 )
    ABORT" File bigger than array" ;
```

The rest of the definition will look something like this:

```
:COMPLEX? FILE.TYPE 10 > ; \ is it complex?  
:EOR? <FILE G$ COUNT 0 = ;  
:F#.NR EOR?  
    IF <FILE NR G$ DROP THEN \ start new record  
        PAD $->F DROP ; \ convert to floating  
  
:FILL <FILE  
    OPEN-INPUT  
    MAT > =FILE?  
    FILE.LEN 0  
    DO DUP 10) ( A{ $filespec -- )  
        COMPLEX?  
        IF F#.NR THEN \ enter 2 #'s  
            F#.NR GI \ put away  
    LOOP DROP  
    CLOSE-INPUT ; \ clean up
```

§§2 Disk file output

Suppose one wants to write an array to a file. We can use

A{{ \$" data.fil" MAT>FILE

or

B{ \$" data.fil" VEC>FILE

where we define⁴

```
:MAT>FILE ( a{{ $adr -- )  
    MAKE-OUTPUT \ $adr = file.spec  
    >FILE DUP D.TYPE .  
    DUP D.LEN DUP . **2 .  
.M CLOSE-OUTPUT ;
```

-
4. We use .M and .V from Chapter 9. The words >FILE, MAKE-OUTPUT and CLOSE-OUTPUT are the output analogues of the input words <FILE, etc.

```
: VEC>FILE          ( V{ $filespec -- )  
MAKE-OUTPUT  
>FILE D.TYPE .  
D.LEN DUP .  
.V CLOSE-OUTPUT ;
```

Logging screen activity

Most FORTHS handle output to a device by vectoring through a user variable called **EMIT**. This allows relatively easy output redirection. Hence, for example, everything sent to the screen can be echoed to the printer (in an MS-DOS system operating on a PC-clone this is trivial: just press ctrl-PrtSc to toggle printer logging on or off).

The terminal sessions reproduced in preceding chapters were captured by logging development sessions to disk files. This is far safer than logging to a memory segment or ram-disk because it produces a record that remains even if one's experiments crash the system. Logging to the printer is also permanent, but the results are not easily inserted into a book!

HS/FORTH permits (at least!) two simple methods for redirecting output, which we now describe.

§§1 Redirection by re-vectoring

The first word in this set of definitions creates the logging file.

```
: MAKE.LOG.FILE  
CR ." Insert a formatted disk in drive A:"  
CR ." Press any key to continue, Esc to abort ..."  
KEY 27 = ABORT" ABORTED!"  
$" A:LOG.TXT" MAKE-OUTPUT ;  
  
: REOPEN.LOG.FILE  
$" A:LOG.TXT" OPEN-OUTPUT ;
```

```

: FWEMIT      \ a word to replace standard EMIT
    DUP        \ duplicate character
    FEMIT      \ emit to open file
    OUT 1-I    \ decrement the output count
    WEMIT ;   \ emit to the CRT (this increments OUT)

: FCRT CFA' FWEMIT IS EMIT; \ vector new routine

: LOG-ON   FCRT      \ all I/O to file & CRT
    IO-STAY ;       \ normally FCRT would stop
                      \ after word executes.
                      \ IO-STAY leaves FCRT on.

: LOG-OFF  CLOSE-OUTPUT IO-RESET ;
                      \ IO-RESET restores I/O
                      \ to standard device

```

§§2 Redirection using MS-DOS resources

Assembly language programmers familiar with MS-DOS⁵ versions 2.x and higher will be aware that the operating system includes many useful functions. Those employed here are invoked by means of an "interrupt" (meaning the CPU halts what it was doing and performs the system function).

The interrupt code to invoke functions is 21h (hex). The system functions are usually specified by placing the function number in register AH, and other needed information is placed in other registers. Then an INT 21 instruction is issued.

For example, to create a new file (or to overwrite an old one of the same name), a program operating under MS-DOS would include the following code:

MOV DX, FILESPEC	; FILESPEC is address of ; a string ending with ASCII 0 ; containing [d:]\[path\]file.ext
MOV AH, 3CH	; the function number of "create"
MOV CX, 0H	; the file attribute
INT 21H	; now perform the interrupt

Consulting the MS-DOS Programmer's Reference⁶ reveals that Function 3Ch either leaves an error code in AX (if there was an error — depicted by having the carry flag bit CF = 1); or leaves the file handle number (if all went well — depicted by CF = 0) in AX.

The HS/FORTH word **MKFILE** expects the address of a counted string on the stack. This string contains the drive, path and filename of the new file. Then **MKFILE** performs the interrupt (using code akin to that above) to invoke the function, issues an error message if something went wrong, and leaves the **file handle number** on the stack if all went well.

This **file handle number** is MS-DOS's way to refer to the new file internally. All subsequent references — say to read or write — are made to this number rather than to the **FILESPEC** (which would be much more tedious). The word **MAKE-OUTPUT** used above contains **MKFILE** as a component.

Now what does all this have to do with the price of bananas in the Maldives Islands? One of the features of MS-DOS is to treat *everything* like a file, including the printer, CRT, serial ports, etc. Thus output intended for the printer can be sent to a file by fooling DOS into thinking the file was the printer. This can be done by placing the file's file handle number where the printer's should be.

But MS-DOS actually has functions we can use to splice two file handle numbers so that they are subsequently treated as a single file. These are Function 45h (duplicate a file handle specified in BX: take an already open file and return a new handle that refers to the same file at the same position) and Function 46h (force a duplicate of a file handle specified in BX: take an already open file and force another handle in CX to refer to the same file at the same position; if the file handle specified in CX was open, close it).

Function 45h is used in the HS/FORTH word **DUPH**, and 46h is used in **SPLICEH**. We can combine them with the (already-

6. Radio Shack Cat. No. 26-5403.

defined) printer-logging word **PCRT**, as well as with **IO-STAY** and **CRT**, to let DOS perform the redirection of output:

```

0 VAR LOG-HANDLE
0 VAR P-H#           \ VARs for storing handles
: LOG-ON $" A:LOG.TXT" MKFILE
    IS LOG-HANDLE      \ make file, leave handle#
    4 DUPH IS P-H#     \ store file handle#
    LOG-HANDLE          \ make new printer handle#
    4 SPLICEH          \ and store it
    PCRT IO-STAY ;     \ splice the handle#'s
                        \ redirect output to
                        \ printer & CRT & leave on

: LOG-OFF P-H# DUP
    4 SPLICEH          \ splice printer back to printer
    LOG-HANDLE CLOSEH
    CLOSEH              \ flush to log-file, close
    CRT ;               \ close new printer handle
    BEHEAD" LOG-HANDLE P-H#   \ go back to CRT only
                                \ make VARs local

```

§§3 Redirection while DEBUGging

The only problem with the logging functions defined above is that they depend on an idiosyncracy of MS-DOS called the *environment*. That is, when we execute a program from within another using the HS/FORTH word **DOS"** we automatically create a new shell and run all DOS commands under a second copy of the command processor, **COMMAND.COM**⁷. This means the environment variables —specifically the handles of open files— will not be valid. Hence the output sent to the screen by a program being run via **DOS"** will not be echoed to the log file.

7. DOS" must be defined in machine language to invoke the appropriate DOS function through another INT 21H call

To capture screen output while DEBUGging, I resort to a utility published in PC Magazine called PRN2FILE⁸. This utility replaces the DOS printer-interrupt routine (INT 17h) by a modification that sends the output to a file instead of the printer. The assembly language listing can be downloaded from PCMag-Net, and is included on the program disk as a convenience to the reader.

To use this utility from within HS/FORTH, first say DOS* <cr> to get into the DOS shell. A DOS prompt such as C> should appear. Now give the DOS commands

```
C> prn2file log.fil [/b8]
C> ctrl-PrtSc      ( toggle echoing to printer)
C> debug          ( get into DEBUG)
..... debugging session .....
-q                ( quit DEBUG)
C> ctrl-PrtSc      ( stop echoing to printer)
C> prn2file        ( with no argument, closes log.fil)
                  ( and redirects back to printer)
C> type log.fil   ( check that it has worked)
C> exit            ( get back to HS/FORTH)
```

Although this is somewhat cumbersome, it does what I need, and I did not have to write my own.

With the illustration of how to use HS/FORTH's MS-DOS extensions to fill an array from a file, to send the contents of an array to a file of standard structure, and to capture programming sessions to a file, we have reached the end of the I/O trail.

Symbolic Programming

Contents

§1 Rules	260
§2 Tools	262
§§1 Pattern recognizers	262
§§2 Finite state machines	265
§§3 FSMs in FORTH	268
§§4 Automatic conversion tables	271
§3 Computer algebra	273
§§1 Stating the problem	273
§§2 The rules	275
§§3 The program	276
§4 FORmula TRANslator	284
§§1 Rules of FORTRAN	285
§§2 Details of the Problem	286
§§3 Parsing	289
§§4 Coding the FORmula TRANslator	296

All symbolic programming is based on **rules** – a set of generalized instructions that tells the computer how to transform one set of **tokens** into another. An assembler, e.g., inputs a series of machine instructions in mnemonic form and outputs a series of numbers that represent the actual machine instructions in executable form. A FORTH compiler translates a definition into a series of addresses of previously defined objects. Even higher on the scale of complexity, a FORTRAN compiler inputs high-level language constructs formed according to a certain **grammar** and outputs an executable program in another language such as assembler, machine code or C.

What do rules have to do with scientific problem-solving? The crucial element in the rule-based style of programming is the ability to specify general **patterns** or even classes of patterns so

the computer can recognize them in the input and take appropriate action.

For example, in a modern high-energy physics experiment the rate at which events (data) impinge on detectors might be 10⁷ discrete events per second. Since each event might be represented by 5-10 numbers, the storage requirements for recording the results of a search for some rare process, lasting 3-6 months of running time, might be 10¹⁶ bytes, or 10⁷ high-capacity disk drives! Clearly, so much storage is out of the question and most of the incoming data must be discarded. That is, such experiments demand extremely fast filtering methods that can determine – in 10-20 μ sec – whether a given event is interesting. The criteria for “interesting” may be quite general and may need to be changed during the running of the experiment. In a word, they must be specified by some form of pattern recognition program rather than hard-wired.

Another area where pattern recognition helps the scientist is computer algebra. Closely related is the ability to translate mathematical formulae into machine code. So far we have stressed a FORTH programming style natural to that language, namely postfix notation, augmenting it primarily for readability or abstracting power. It cannot be denied, however, that sometimes it is useful simply to be able to write down a mathematical formula and have it translated automatically into executable form. This chapter develops the tools for symbolic programming and illustrates their use with a typical algebra program and a simple FORmula TRANslator.

§1 Rules

Before we can specify rules we need a language to express them in. We need to be able to describe the grammar of the rules in some way. The standard notation states rules as **regular expressions**¹. The following rules describing some parts of FORTRAN illustrate how this works:

1. See A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: ...* (Addison-Wesley, Reading, 1988).

\ Rules for FORTRAN

\ NOTATION:

```
\ |      -> "or",  
 \ ^     -> "unlimited repetitions"  
 \ ^n    -> "0-n repetitions"  
 \ Q     -> "empty set"  
 \ &     -> + | -  
 \ %     -> * | /  
 \ <d>   -> "digit"
```

\ NUMBERS:

```
\ <int>  -> {- | Q} {<d> <d> ^ 8}  
 \ <exp't> -> {dDeE} {& | Q} {<d> <d> ^ 2} | Q  
 \ <fp#>  -> {-|Q} {<d> | Q}. <d> ^ <exp't>
```

\ FORMULAS:

```
\ <assign> -> <subj> = <expression>  
 \ <id>    -> <letter> {<letter> | <d>} ^ 6  
 \ <subject> -> <id> {<idlist> | Q}  
 \ <idlist> -> ( <id> {, <id>} ^ )  
 \ <arglist> -> ( <expr'n> {, <expr'n>} ^ )  
 \ <func>   -> <id> <arglist>  
 \ <expr'n> -> <term> | <term> & <expr'n>  
 \ <term>   -> <fctr> | <fctr> % <trm> | <fctr> ** <fctr>  
 \ <factor> -> <id> | <fp#> | ( <expr'n> ) | <func>
```

We use angular brackets “<”, “>” to set off “parts of speech” being defined, and arrows “->” to denote “is defined by”. Other notational conventions, such as “|” to stand for “or”, are listed in the “NOTATION” section of the rules list, mainly for mnemonic reasons. A statement such as

\ <int> -> {-|Q}<d><d>^8

therefore means “an integer is defined by an optional leading minus sign, followed by 1 digit which is in turn followed by as many as 8 more digits”. Similarly, the phrase

\ <assign> -> <subj> = <expression>

means “an assignment statement consists of a subject – a symbol that can be translated into an address in memory – followed by an equals sign, followed by an expression”. Literal symbols – parentheses, decimal points, commas – are shown in **bold** type.

Note that some of these definitions are recursive. A statement such as

$\backslash <\text{expr}'n> \rightarrow <\text{term}> | <\text{term}> \& <\text{expr}'n>$

seems to be defined in terms of itself. So it is a good bet the program that recognizes and translates a FORTRAN expression will be recursive, even if not explicitly so.

§2 Tools

In order to apply a rule stated as a regular expression, we need to be able to recognize a given pattern. That is, given a string, we need —say— to be able to state whether it is a floating point number or something else. We want to step through the string, one character at a time, following the rule

$\backslash <\text{fp}\#> \rightarrow \{-|Q\}\{\{d\} . | . d | d\} d^{\wedge} \text{exp't}$

This pattern begins with a minus or nothing, followed by a digit and a decimal point or a decimal point and a digit or a digit with no decimal point, followed by zero or more digits, then an exponent.

§§1 Pattern recognizers

One often sees pattern recognizers expressed as complex logic trees, *i.e.* as sequences of nested conditionals, as in Fig. 11-1 on page 263 below. As we see, the tree is already five levels deep, even though we have concealed the decisions pertaining to the exponent part of the number in a word **exponent?**. When programmed in the standard procedural fashion with **IF...ELSE...THEN** statements, the program becomes too long

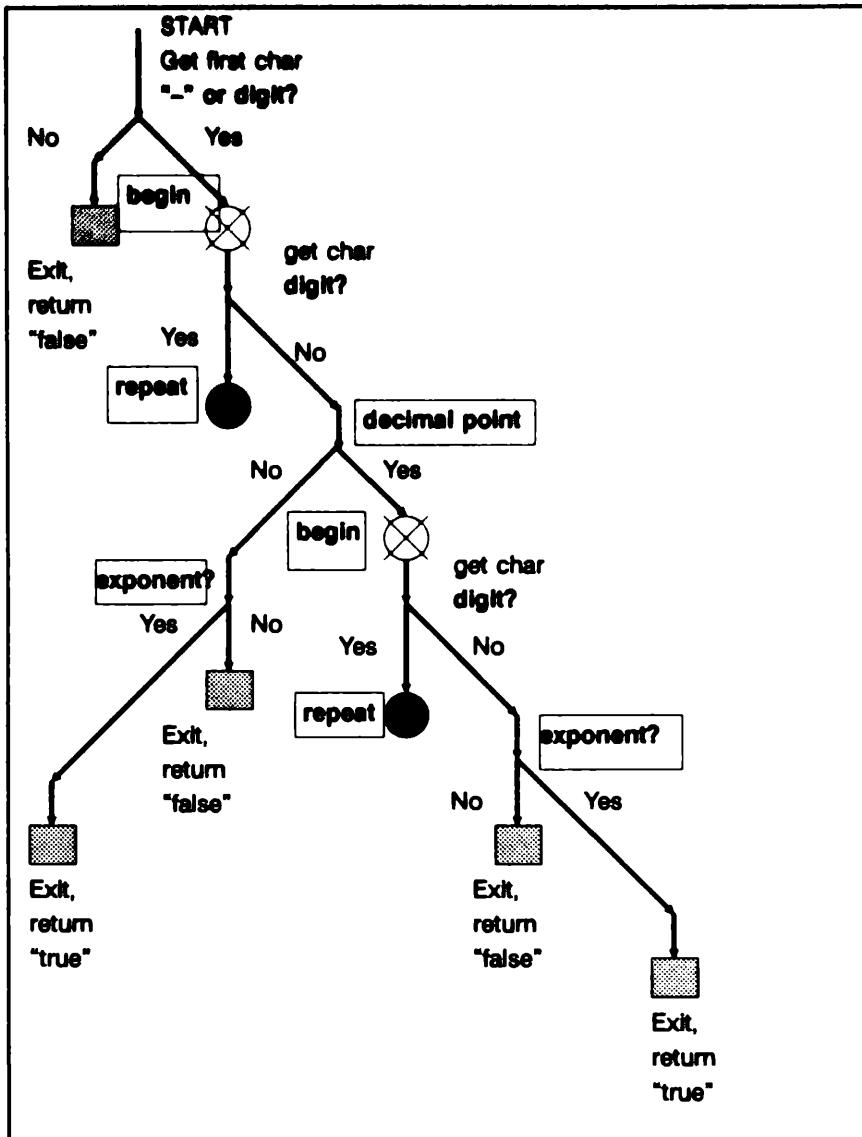


Fig. 11-1 Logic tree for <floating point #>

and too complex either for easy comprehension or for easy maintenance².

It has been known for many years that a better way to apply general rules – e.g., to determine whether a given string conforms to the rules for “floating point number” – uses finite state machines (FSMs – we define them in §2 §§2 below). Here is an example, written in standard FORTH:

```
\ determine whether the string at $adr is a fp#
: skip-  ( adr -- adr') DUP C@ ASCII - = - ;
: skip_dp ( adr -- adr') DUP C@ ASCII . = - ;
\ NOTE: these "hacks" assume "true" = -1.
: digit?  ( char -- f) ASCII 9 ASCII 0 WITHIN ;
: skip_dig ( adr2 adr1 -- adr2 adr1')
    BEGIN DDUP > OVER C@ digit? AND
    WHILE 1+ REPEAT ; \ ... cont'd below
: dDeE?  ( char -- f) 95 AND \ -> uppercase
    DUP ASCII D = SWAP ASCII E = OR ;

: skip_exponent ...; \ this definition shown below

: fp#?  ($adr -- f)
    DUP 0 OVER COUNT + CI \ add terminator
    DUP C@ 1+ OVER CI \ count = count + 1
    COUNT OVER + 1- SWAP (- $end $beg )
    skip- skip_dig skip_dp skip_dig
    skip_exponent
    UNDER = \ $beg' = $end?
    SWAP C@ 0= AND ; \ char[$beg'] = terminal?
```

The program works like this:

- Append a unique terminal character to the string.
- If the first character is “–” advance the pointer 1 byte, otherwise advance 0 bytes.

2. These defects of nested conditionals are generally recognized. Commercially available CASE tools such as Stirling Castle's *Logic Gem* (that translates logical rules to conditionals); and Matrix Software's *Matrix Layout* and AYECO, Inc.'s *COMPETITOR* (that translate tabular representations of FSMs to conditionals in any of several languages) were originally developed as in-house aids.

- Skip over any digits until a non-digit is found.
- If that character is a decimal point skip over it.
- Skip any digits following the decimal point.
- A floating point number terminates with an exponent formed according to the appropriate rule (p. 261). `skip_exponent` advances the pointer through this (sub)string, or else halts at the first character that fails to fit the rule.
- Does the initial pointer (`$beg'`) now point to the calculated end of the string (`$end`)? And is the last character (`char[$beg']`) the unique terminal? If so, report “true”, else report “false”.

We deferred the definition of `skip_exponent`. Using conditionals it could look like

```
: skip_exponent ( adr -- adr')
    DUP C@ dDeE? IF 1+ ELSE EXIT THEN
    skip- skip+
    DUP C@ digit? IF 1+ ELSE EXIT THEN
    DUP C@ digit? IF 1+ ELSE EXIT THEN
    DUP C@ digit? IF 1+ ELSE EXIT THEN ;
```

which, as we see in Fig. 11-2 below, has nearly as convoluted a logic tree as Fig. 11-1 on page 263 above.

§§2 Finite state machines

Just as we needed a FSM to achieve a graceful definition of `fp#?`, we might try to define `skip_exponent` as a state machine also. This means it is time to define what we mean by finite state machines. (We restrict attention to deterministic FSMs.) A **finite state machine**^{3,4} is a program (originally it was a hard-wired switching circuit⁵) that takes a set of discrete, mutually exclusive

3. R. Sedgewick, *Algorithms* (Addison-Wesley Publishing Co., Reading, MA 1983) p. 257ff.
4. A.V. Aho, R. Sethi and J.D. Ullman, *Compilers: Principles, Tools and Techniques* (Addison Wesley Publishing Company, Reading, MA, 1988).
5. Zvi Kohavi, *Switching and Finite Automata Theory*, 2nd ed. (McGraw-Hill Publishing Co., New York, 1978).

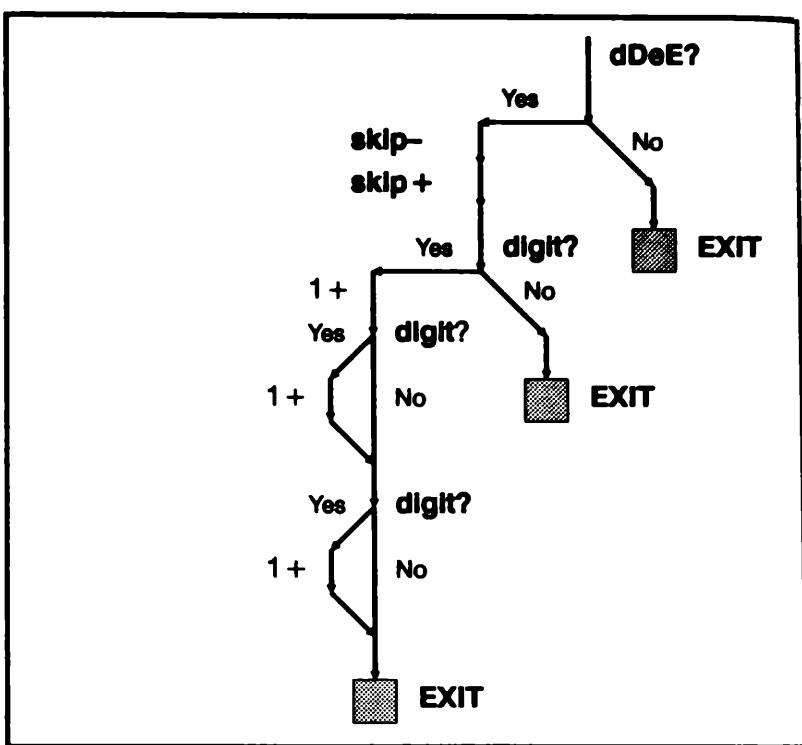


Fig. 11-2 Logic tree for <exponent>

inputs and also maintains a **state variable** that tracks the history of the machine's inputs. According to which state the machine is in, a given input will produce different results. The FSM program is most easily expressed in tabular form, as in Table 11-1, which we interpret as follows:

- each major column heading is an input.
- the inputs must be **mutually exclusive** and **exhaustive**; to exhaust all possibilities we include “other”.
- each row represents the current state of the machine.
- each cell contains an action, followed by a state-transition.

Table 11-1 Example of finite state machine arrow (→) means "next state"

<u>Input: other</u>	<u>dDeE</u>	<u>±/-</u>	<u>digit</u>
<u>State</u>	→	→	→
↓			
0	Next 5	1 + 1	Error 5 Next 5
1	Next 5	Next 5 1 + 2	1 + 3
2	Next 5	Next 5 Next 5	1 + 3
3	Next 5	Next 5 Next 5	1 + 4
4	Next 5	Next 5 Next 5	1 + 5

The tabular representation of a FSM is much clearer than the logic diagram, Fig. 11-2. Since the inputs must be **mutually exclusive**⁶ and **exhaustive**⁷, there are *never* conditions that cannot be fulfilled – that is, leading to “dead” code – as frequently happens with logic trees (owing to human frailty). This means the chance of introducing bugs is reduced by FSMs in tabular form.

FORTRAN, BASIC or Assembler can implement FSMs with computed GOTOS. In BASIC, e.g.,

```

DEF SUB FSM (c$, address%)
k% = 0      ' convert input to column #
C$ = UCASE (c$)
IF C$ = "D" OR C$ = "E" THEN k% = 1
IF C$ = "+" OR C$ = "-" THEN k% = 2
IF ASC(C$) >= 48 AND ASC(C$) <= 57 THEN k% = 3
' cont'd

```

6. “Mutually exclusive” means only one input at a time can be true.
7. “Exhaustive” means every possible input must be represented.

```

    ' begin FSM proper
ON state% *3 + k% GOTO
(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19)
0: state% = 5
1: adres% = adres% + 1 : state% = 1
2: CALL Error : state% = 5
3: state% = 5
4: state% = 5
5: state% = 5
6: adres% = adres% + 1 : state% = 2
7: adres% = adres% + 1 : state% = 3
..... etc. .....
16: state% = 5
..... etc. .....
19: adres% = adres% + 1 : state% = 5
END SUB
    ' row 0
    ' row 1
    ' row 4
}

```

The advantage of FSM construction using computed GOTOS is simplicity; its disadvantage is the linear format of the program that hides the structure represented by the state table, 11-1. CASE statements –as in C, Pascal or QuickBASIC– are no clearer. We can use a state table for documentation, but the subroutine takes more-or-less the above form.

§§3 FSMs in FORTH

In the preceding FORTH example we *synthesized* the FSM from **BEGIN...WHILE...REPEAT** loops. FORTH's lack of line-labels and GOTOS (jumps) imposed this method, producing code as untransparent as the BASIC version. The Eaker CASE statement can streamline the program,

```

CASE: ADVANCE? NEXT 1+ ;CASE
: 1digit DUP C@ digit? ABS ADVANCE? ;
: skip_exponent ( adr -- adr')
    DUP C@ dDeE? IF 1+ ELSE EXIT THEN
    skip- skip+
    1digit 1digit 1digit ;

```

(three of four **IF...ELSE...THENs** have been factored out and disguised as **CASE: ... ;CASE**), but this does not much improve clarity. We still need the state transition table to understand the program.

Various authors have tried to improve FSMs in FORTH using what amounts to line-labels and GOTOS^{8,9,10}. The resulting code is less elegant than the BASIC version shown above.

Whenever we reach a dead end, it is helpful to return to the starting point, restate the problem and re-examine our basic assumptions. One fact our preceding false starts make abundantly clear is that nowhere have we used the power of FORTH. Rather, our attempts merely imitated traditional languages in FORTH.

But FORTH is an endlessly protean language that lends itself to any programming style. Ideally FORTH relies on names so cunningly chosen that programs become self-documenting – readable at a glance.

Since state tables clearly document FSMs, it eventually occurred to me to let FORTH compile the state table – representing an FSM – directly to that FSM!

Compilation implies a compiling word; after some experimentation¹¹ I settled on the following usage¹²:

4 WIDE FSM: (exponent)								
\ Input:	other		dDeE		+/-		digit	
\ state:	-----		-----		-----		-----	
(0)	NEXT	>5	1+	>1	Error	>5	NEXT	>5
(1)	NEXT	>5	NEXT	>5	1+	>2	1+	>3
(2)	NEXT	>5	NEXT	>5	NEXT	>5	1+	>3
(3)	NEXT	>5	NEXT	>5	NEXT	>5	1+	>4
(4)	NEXT	>5	NEXT	>5	NEXT	>5	1+	>5 :

Fig. 11-3 Form of a FORTH finite state machine

8. J. Basile, *J. FORTH Appl. and Res.* 1,2 (1982) 76-78.
9. E. Rawson, *J. FORTH Appl. and Res.* 3,4 (1986) 45-64.
10. D.W. Berrian, *Proc. 1989 Rochester FORTH Conf.* (Inst. for Applied FORTH Res., Inc., Rochester, NY 1989) p. 1-5.
11. The development process is described in detail in my article "Avoid Decisions", *Computers in Physics* 5, #4 (1991) 386.
12. The FORTH word NEXT is the equivalent of NOP in assembler.

The new defining word **FSM:** has a colon ":" in its name to remind us of its function. Its children clearly must "know" how many columns they have. The word **WIDE** reminds us the newly created FSMs incorporate their own widths.

The column labels and table headers are merely comments following "\"; the state labels "(0)", "(1)", etc. are also comments, delineated with parentheses. Their only purpose is readability.

The actions -- **NEXT**, **Error**, **1+** -- in Fig. 11-3 are obvious: they are simply previously-defined words. But what about the state transitions "**>1**", "**>2**", ... ? The easiest, most mnemonic and natural way to handle state transitions defines them as **CONSTANTS**

```
0 CONSTANT >0
1 CONSTANT >1
2 CONSTANT >2
... etc. ...
```

which are also actions to be compiled into the FSM. This follows the general FORTH principle that words should execute themselves¹³.

In Chapter 5§2§§6 we used components of the compiler, particularly the **IMMEDIATE** word] ("switch to compile mode"), to create self-acting jump tables. We apply the same method here: The defining word **FSM:** will **CREATE** a new dictionary entry, build in its width (number of columns) using ",", and then compile in the actions and state transitions as cfa's of the appropriate words.

The runtime code installed by **DOES>** provides a mechanism for finding the addresses of action and state transition corresponding to the appropriate input and current state (that is, in the cell of interest). Then the runtime code updates the state.

13. That is, we should not continually reinvent interpreters equivalent to the FORTH interpreter.

To allow nesting of FSMs (i.e., compiling one into another), we incorporate the state variable for each child FSM within its data structure. This technique, using one extra memory cell per FSM, protects the state from accidental interactions, since if state has no name it cannot be invoked inadvertently.

The FORTH code that does all this is

```
: WIDE 0 ;
: FSM:           ( width 0 -- ) CREATE , , ]
  DOES>         ( col# -- )
    UNDER D@      ( -- adr col# width state )
    * + 1+ 4*      ( -- adr offset )
    OVER +
    DUP@ SWAP 2+  ( -- adr [adr'] adr' + 2 )
    @ EXECUTE      ( -- adr [adr'] state' )
    ROT ! EXECUTE ;
0 CONSTANT >0
1 CONSTANT >1
2 CONSTANT >2
... etc. ...
```

We are now in a position to use the code for (**exponent**) (defined in Fig. 11-3 on page 269 above) to define the key word **skip_exponent** appearing in **fp#?**. The result is

```
: skip exponent          ( adr -- adr' )
  ' (exponent) 0!        \ initialize state
  BEGIN DUP C@ DUP      ( -- adr char)
    dDeE? ABS OVER      \ input -> col#
    +/-? 2 AND + SWAP
    digit? 3 AND + ( -- adr col#)
    ' (exponent) @        \ get state
    5 <                  \ not done?
  WHILE (exponent) REPEAT ;
```

§§4 Automatic conversion tables

Our preceding example used logic to compute (*not decide!*) the conversion of input condition to a column number, via

```
( -- char) dDeE? ABS OVER
  +/-? 2 AND + SWAP
  digit? 3 AND + ( -- col#)
```

When the input condition is a character, it is usually both faster and clearer to translate to a column number using a lookup table rather than tests and logic. That is, we can trade increased memory usage for speed. If a program needs many different pattern recognizers, it is worth generating their lookup tables via a defining word rather than crafting each by hand.

```
: TABLE:          ( -- #bytes )
    CREATE HERE      ( -- #bytes tab[0] )
    OVER ALLOT       \ allot #bytes in dictionary
    SWAP 0 FILL      \ initialize to all 0's
    DOES> + C@ ;     ( n tab[0] .. [tab[n]] )

: install ( col# adr char.n char.1 .. )      \ fast fill
    SWAP 1+ SWAP
    DO DDUP 1+ CI LOOP DDROP ;
```

Here is how we define a new lookup table:

```
128 TABLE: [exp]           \ define 128-byte table
\ modify certain chars
\ Note: all unmodified chars return col# 0

1 ASCII d '[exp] + CI      \ col# 1
1 ASCII D '[exp] + CI
1 ASCII e '[exp] + CI
1 ASCII E '[exp] + CI

2 ASCII + '[exp] + CI      \ col# 2
2 ASCII - '[exp] + CI

3 '[exp] ASCII 9 ASCII 0 install \ col# 3
```

With the lookup table **[exp]**, **skip_exponent** becomes faster and more graceful,

```
: skip_exponent ( adr .. adr' )
    '(exponent) 0!           \ state = 0
    BEGIN DUP C@            ( .. adr char)
    [exp]                   ( .. adr col#)
    '(exponent) @           ( .. adr col# state)
    5 <                     \ not done?
    WHILE (exponent) REPEAT ;
```

at a cost of 128 bytes of dictionary space. If dictionary space becomes tight, it would be perfectly simple to export the lookup

tables to their own segment, in the same way we did with generic arrays in Chapter 5.

§3 Computer algebra

One of the most revolutionary recent developments in scientific programming is the ability to do algebra on the computer. Programs like REDUCE, SCHOONSCIP, MACSYMA, DERIVE and MATHEMATICA can automate tedious algebraic manipulations that might take hours or years by hand, in the process reducing the likelihood of error¹⁴. The study of symbolic manipulation has led to rich new areas of pure mathematics¹⁵. Here we illustrate our new tool (for compiling finite state automata) with a rule-based recursive program to solve a problem that does not need much formal mathematical background. The resulting program executes far more rapidly on a PC than REDUCE, e.g., on a large mainframe.

§§1 Stating the problem

Dirac γ -matrices are 4×4 traceless, complex matrices defined by a set of (anti)commutation relations¹⁶. These are

$$\gamma^\mu \gamma^\nu + \gamma^\nu \gamma^\mu = 2 \eta^{\mu\nu}, \quad \mu, \nu = 0, \dots, 3 \quad (1)$$

where $\eta^{\mu\nu}$ is a matrix-valued tensor,

$$\eta^{\mu\nu} = \begin{matrix} \mu \backslash \nu & 0 & 1 & 2 & 3 \\ 0 & \begin{pmatrix} 1 & 0 & 0 & 0 \end{pmatrix} \\ 1 & \begin{pmatrix} 0 & -1 & 0 & 0 \end{pmatrix} \\ 2 & \begin{pmatrix} 0 & 0 & -1 & 0 \end{pmatrix} \\ 3 & \begin{pmatrix} 0 & 0 & 0 & -1 \end{pmatrix} \end{matrix} \quad (2)$$

14. R. Pavelle, M. Rothstein and J. Fitch, "Computer Algebra", *Scientific American* 245, #6 (Dec. 1981) 136.
15. See, e.g., M. Mignotte, *Mathematics for Computer Algebra* (Springer-Verlag, Berlin, 1992).
16. They are said to satisfy a Clifford algebra. See, e.g., J.D. Bjorken and S.D. Drell, *Relativistic Quantum Mechanics* (McGraw-Hill, Inc., New York, 1964); also V.B. Berestetskii, E.M. Lifshitz and L.P. Pitaevskii, *Relativistic Quantum Theory, Part 1* (Pergamon Press, Oxford, 1971) p. 68ff.

The trace of a matrix is defined to be the sum of its diagonal elements,

$$\text{Tr}(\mathbf{A}) \hat{=} \sum_{k=1}^N A_{kk} . \quad (3)$$

Clearly, traces obey the **distributive law**

$$\text{Tr}(\mathbf{A} + \mathbf{B}) \equiv \text{Tr}(\mathbf{A}) + \text{Tr}(\mathbf{B}) \quad (4a)$$

as well as a kind of **commutative law**,

$$\text{Tr}(\mathbf{AB}) \equiv \text{Tr}(\mathbf{BA}) . \quad (4b)$$

From Eq. 1, 2, and 4a,b we find

$$\begin{aligned} \text{Tr}(\mathbf{AB}) &\equiv \frac{1}{2} [\text{Tr}(\mathbf{AB}) + \text{Tr}(\mathbf{BA})] = \frac{1}{2} \text{Tr}(\mathbf{AB} + \mathbf{BA}) \\ &= \frac{1}{2} 2 \mathbf{A} \cdot \mathbf{B} \text{Tr}(\mathbf{I}) \equiv 4 \mathbf{A} \cdot \mathbf{B} \end{aligned} \quad (5)$$

where the ordinary vectors \mathbf{A}^μ and \mathbf{B}^ν form the (Lorentz-invariant) "dot product"

$$\mathbf{A} \cdot \mathbf{B} \equiv A^0 B^0 - A^1 B^1 - A^2 B^2 - A^3 B^3 \quad (6)$$

The trace of 4 gamma matrices can be obtained, analogous to Eq. 5, by repeated application of the algebraic laws 1-4:

$$\begin{aligned} \text{Tr}(\mathbf{ABCD}) &= 2 \mathbf{A} \cdot \mathbf{B} \text{Tr}(\mathbf{CD}) - \text{Tr}(\mathbf{BCAD}) \\ &\equiv 2 \mathbf{A} \cdot \mathbf{B} \text{Tr}(\mathbf{CD}) - \text{Tr}(\mathbf{ACBD}) \\ &\equiv 4 (A \cdot B C \cdot D - A \cdot C B \cdot D + A \cdot D B \cdot C) \end{aligned} \quad (7)$$

We include Eq. 7 for testing purposes. The factors of 4 in Eq. 5 and 7 do nothing useful, so we might as well suppress them in the interest of a simpler program.

§§2 The rules

We apply formal rules analogous to those used in parsing a language¹⁷. The rules for traces are:

\ Gamma Matrix Algebra Rules:

\ a	->	string of length < = 3
\ /	->	delineator for factors
\ <factor>	->	a/
\ product/	->	a/b/c/d/ ...
\ Tr(a/b/prod/)	->	a.b (tr(prod/)) - tr(a/prod/b' /)
\ b'	->	mark b as permuted
\ Tr(a/)	->	0 (a single factor is traceless)

Repeated (adjacent) factors can be combined to produce a multiplicative constant:

$$\mathbf{B} \mathbf{B} \equiv \mathbf{B} \cdot \mathbf{B}; \quad (8)$$

recognizing such factors can shorten the final expressions significantly, hence the rule

$$\backslash \text{Tr}(\mathbf{a}/\mathbf{a}/\text{prod}/) \quad -> \quad \mathbf{a} \cdot \mathbf{a} (\text{Tr}(\text{prod}/)).$$

In the same category, when two vectors are orthogonal, $\mathbf{A} \cdot \mathbf{B} = 0$, another simplification occurs,

$$\begin{aligned} \backslash \text{PERP } \mathbf{A} \mathbf{B} &\quad -> \quad \mathbf{A} \cdot \mathbf{B} = 0. \\ \backslash \text{Tr}(\mathbf{A}/\mathbf{B}/\text{prod}/) &\quad -> \quad - \text{Tr}(\mathbf{A}/\text{prod}/\mathbf{B}' /) \end{aligned}$$

The ability to recognize orthogonal vectors lets us (correctly) include the trace of a product times the special matrix¹⁸

17. It is not difficult to introduce one further level of indirection and produce Forth code for generating a "compiler". See T.A. Ivanko and G. Hunter, *J. Forth Appl. and Res.* 6 (1990) 15.
18. Note: Berestetskii, et al. (*op. cit.*) define γ^5 with an overall "-" sign relative to Eq. 9.

$$\gamma^5 = i \gamma^0 \gamma^1 \gamma^2 \gamma^3 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \quad (9)$$

that **anticommutes** with all four of the γ^μ :

$$\gamma^5 \gamma^\mu + \gamma^\mu \gamma^5 \equiv 0, \quad \mu = 0, \dots, 3.$$

The fully antisymmetric tensor $\epsilon_{\mu\nu\rho\lambda} \equiv -\epsilon_{\nu\rho\lambda\mu}$, etc., lets us write

$$\gamma^5 A^\mu B^\nu C^\rho \gamma^\lambda \equiv i \epsilon_{\mu\nu\rho\lambda} A^\mu B^\nu C^\rho \gamma^\lambda, \quad i = \sqrt{-1} \quad . \quad (10)$$

A complete gamma matrix package includes rules for traces containing γ^5 :

$$\begin{cases} \text{\textbackslash Trg5(a/b/c/d/x/) } & \rightarrow & i \text{Tr(} \wedge /d/x/) \\ \wedge .d & \rightarrow & [a,b,c,d] \text{ (antisymmetric product)} \\ \text{\textbackslash Note: } \wedge .a = \wedge .b = \wedge .c = 0 \end{cases}$$

Since γ -matrices often appear in expressions like $(A + m_A I)$, it will also be convenient to include the additional rules

$$\begin{cases} \text{\textbackslash *A/} & \rightarrow & [A/ + m_A] \\ \text{\textbackslash Tr(*A/ x/)} & \rightarrow & m[A] (\text{Tr(} x/)) + \text{Tr(} x/ A/) \end{cases}$$

§§3 The program

We program from the bottom up, testing, adding and modifying as we go. Begin with the user interface; we would like to say

`TR(A/B/C/D/)`

to obtain the output:

`= A.BC.D-A.CB.D+A.DB.C ok`

Evidently our program will input a string terminated by a right parenthesis ")", i.e., the right parenthesis tells it to stop inputting. This can be done with the word¹⁹

```
: get$ ASCII ) TEXT PAD XS $! ;
```

Since the rules are inherently recursive, we push the input string onto a stack before operations commence. What stack? Clearly we need a stack that can hold strings of "arbitrary" length. The strings cannot be too long because the number of terms of output, hence the operating time, grows with the number of factors N, in fact, like $\left(\frac{1}{2}N\right)!$.

The pseudocode for the last word of the definition is clearly something like

```
: TR(      )get$      \ get a $ – terminated with ")"
    setup      \ push $ on $stack
    parse   ;
```

The real work is done by **parse**, whose pseudocode is shown below in Fig. 11-4; note how recursion simplifies the problem of matching left and right parentheses in the output.

Next we define the underlying data structures. Recursion demands a stack to hold strings in various stages of decomposition and permutation. Since the number of terms grows very rapidly with the number of factors, it will turn out that taking the trace of as many as 20 distinct factors is a matter of some weeks on –say– a 25 Mhz 80386 PC; that is, 14 or 16 factors are the largest practicable number. So if we make provision for expressions 20 factors long, that should be large enough for practical purposes²⁰.

19. The word **TEXT** can be defined as : **TEXT WORD HERE PAD \$!** ;

20. Actually, 19 factors, since we want to **ALLOT** space in multiples of 4 to maintain even paragraph boundaries. That is, the strings will use 20 bytes including the count.

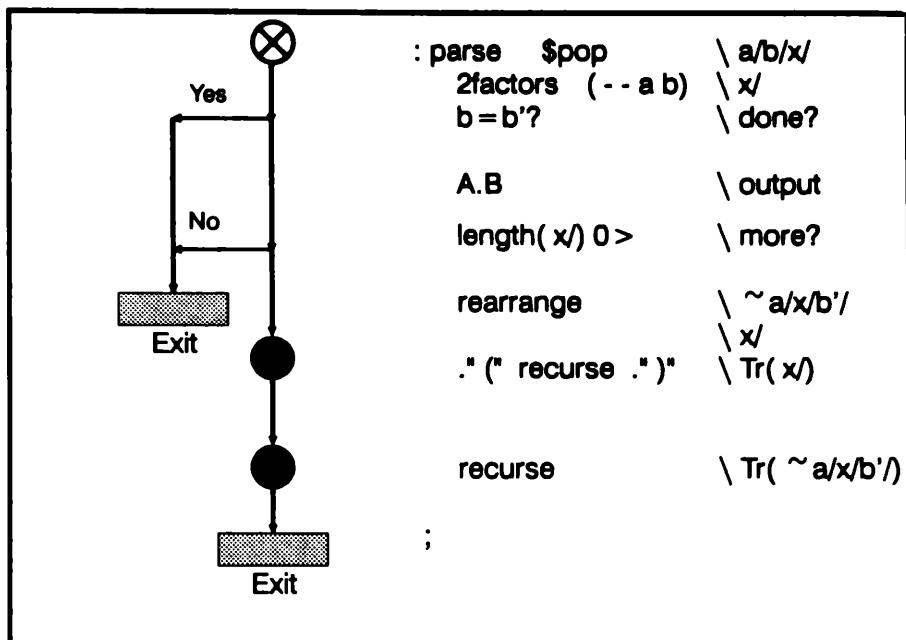


Fig. 11-4 Pseudocode/flow diagram for "parse"

How deep can the \$stack get? The algorithm expressed by the rule

$$\backslash \text{Tr}(a/b/\text{prod}/) \rightarrow a.b (\text{Tr}(\text{prod}/)) - \text{Tr}(a/\text{prod}/b'/)$$

suggests that for an expression of length $n = 2k$, the maximum depth will be $k + 1$. Thus we should plan for a stack depth of at least 11, perhaps 12 for safety. Assuming factor-names up to three characters long, and strings of up to 20 names, we need ≈ 60 characters per string. My first impulse was to create a dynamic \$stack that could accommodate strings of variable length, meaning that some 330 bytes of storage would be needed, at the cost of some complexity in keeping track of the addresses. This is a big improvement on 720 bytes needed for a 60-wide fixed-width stack, of course. However, the convenience of a fixed-width \$stack led me ultimately to set up a table (array) of 20 names, whose indices would act as tokens; that is, the strings that actually go on the \$stack would be tokenized. The memory cost of the table together with a 12-deep, fixed-width \$stack is thus only $80 + 242 = 322$ bytes.

```

TASK GAMMA
\ FINITE STATE MACHINE
:WIDE 0;
:FSM: ( width 0 - ) CREATE ... ]
DOES> ( code - )
  UNDER D@ ( - adr code width state )
  * + 1+ 4* ( - adr offset )
  OVER +
  DUP@ SWAP 2+ ( - adr [adr'] adr'+2 )
  EXECUTE@ ( - adr [adr'] state )
  ROT ! EXECUTE ;
0 CONSTANT >0 3 CONSTANT >3
1 CONSTANT >1 4 CONSTANT >4
2 CONSTANT >2 5 CONSTANT >5
\END FINITE STATE MACHINE

\ Automatic conversion tables
:TAB: (#bytes - )
  CREATE HERE OVER ALLOT
    SWAP 0 FILL \ init table
  DOES> + C@;

:install ( code adr char.n char.1 - ) \ fast fill
  SWAP 1+ SWAP \ addresses
  DO DDUP ! + C! LOOP DDROP;
\ end automatic conversion tables

\ STRING HANDLING
HEX
\ Note: ?((( ..... ))) conditionally compiles "....."
FIND $! 0=
?(((:$! (addr adr - ) OVER C@ 1+ CMOVE ;)))
FIND $+ 0=
?(((:$+ (adr$1 adr$2 - pad)
  DUPC@ (- $adr$1)
  >R 1+ OVER C@ PAD + 1+
  R@ <CMOVE PAD $!
  R> PAD C@ + 0 MAX FF MIN PAD C! ;)))
DECIMAL
:1+C! (adr - ) DUPC@ 1+ SWAP C! ;
:BL (addr - ) \ delete all blanks from $
  DUP>R 0 PAD C! COUNT OVER + SWAP
  DO 1 C@ DUP 32 <>
    IF PAD COUNT + C! PAD 1+C!
    ELSE DROP THEN
  LOOP PAD R> $! ;
\END STRING HANDLING

\ PARSE WORDS
\ Note: The factor "A" in the input string
\ means (A + M sub A)
0 VAR star
:star! 128 IS star DROP; \ set 7th bit if 1st char = "
0 VAR #/
:#/ AT #/ 1+ | counts # of factors
  DROP; \ inc #/
:err OR -1 ABORT" Name -> letter (letter | number)" ;
:+PAD (char - ) star OR \ set bit 7
  PAD COUNT + C! PAD C@ 1+ PAD C! 0 IS star;

5 WIDE FSM: (factor) (char code - )
\ Input | other | letter | digit | / | * |
\state -----
(0) err >0 +PAD >2 err >0 +#/ >5 star! >1
(1) err >0 +PAD >2 err >1 +#/ >5 err >1
(2) err >0 +PAD >3 +PAD >3 +#/ >5 err >2
(3) err >0 +PAD >4 +PAD >4 +#/ >5 err >3
(4) err >0 DROP >4 DROP >4 +#/ >5 err >4
: \ terminate definition

128 TAB: (factor) \ convert char to code
1 '[factor] ASCII Z ASCII A install
1 '[factor] ASCII z ASCII a install
2 '[factor] ASCII 9 ASCII 0 install
3 '[factor] ASCII / + C!
4 '[factor] ASCII * + C!

:<factor> (adr - adr')
  PAD 0! 'factor 0! 0 IS star \ initialize
  BEGIN DUPC@ DUP [factor] (factor) 1+
    'factor) @ 5 =
  UNTIL;

CREATE factor{ 20* ALLOT OKLW \ up to 20 factors
:} (adr n - adr + 4*n) 4* +; \ compute address

0 VAR N 0 VAR N1
CREATE BUFS 20 ALLOT OKLW \ data structures

:unstar 127 AND; \ token["A"] = token[A] 128 OR
:$= ($adr1 $adr2 - f)
  -1 -ROT COUNT
  ROT COUNT (- $adr2 + 1 n2 $adr1 + 1 n1)
  ROT DDUP =
  IF DROP
    0 DO DUPC@ unstar >R 1+
    OVER C@ unstar R> <>
    IF ROT NOT -ROT LEAVE THEN
      SWAP 1+
    LOOP DDROP
  ELSE DDROP DDROP NOT THEN;
:check.table (--) \ prevent duplicate tokens
  -1 IS N1
  N0 DO PAD factor{1} $=
    IF 1 IS N1 LEAVE THEN
  LOOP;

:+bufs NN1 -1 = AND N1 -1 > N1 AND +
  \ taken N or N1
  PAD 1+ C@ 128 AND OR \ If star, set bit 7
  BUFS DUPC@ + 1+ C! BUFS 1+C! ;
  \ append token

```

```

: tokenize ($adr -)           \ decompose into factors
  factor( $0 0 FILL          \ initialize table
  COUNT OVER + (- $adr + 1 $adr') \ addresses
  > R                         \ $adr' = $adr + LEN($) + 1
  0 BUF$ CI                   \ init. buffer
  0 IS N                      \ init. N
  BEGIN <factor>             \ begin loop; get factor
    check.table                \ prevent multiple entries
    +buf$                      \ add factor to tokenized $
    N1-1 =                     \ not in table?
    IF PAD factor{ N } $!     \ put in table
      AT N 1+!                 \ inc. N
    THEN
    DUP R@ = UNTIL            \ end loop
    DROP RDROP ;              \ clean up
  \ END PARSE WORDS

\ $stack
CREATE $stack 20 20 * 2 + ALLOT OKLW \ 2 for ptr
:$push ($adr -)
  DUPC@ 19 > ABORT" STRING TOO LONG!"
  $stack DUP@ DUP 19 > ABORT" $stack too deep!"
  20 * 2 + + $! $stack 1+!
:$pop ($adr -) $stack DUP@ 1- 0 MAX
  DDUP SWAP !
  20 * 2 + + SWAP $! ;
\ end $stack

CREATE XS 80 ALLOT OKLW \ buffer for input $, tail$
:$get$ ASCII) TEXT          \ input a $ terminated by )
  PAD -BL                   \ delete blanks
  PAD XS $! ;

: 1factor ( -- a)           \ get 1 factor, tail -> x$
  BUF$ 1+ C@ ( -- a)
  BUF$ COUNT 1- > R
  1+ XS 1+ R@ <CMOVE      \ tail -> xs
  R > XS CI ;              \ adjust count
: 2factors ( -- ab)         \ get 1st 2 factors
  1factor XS BUF$ $! 1factor ;

0 VAR sign                  \ emit correct sign
2 TAB: [sign]                \ make table
ASCII + '[sign] CI           \ fill table
ASCII - '[sign] 1+ CI
:$sign ( a - ) 64 AND 0 > ABS [sign] sign AND EMIT ;

\ Note: we mark prime (') by 64 OR (set bit 6 in token)
\ since 1st factor is never ', we set bit 6 for leading "-" sign
: prime! 64 OR ;
: unprime 63 AND ;
: A.B ( a b -)               \ emit "dot product"
  unprime                   \ b' -> b
  OVER .sign                 \ emit sign
  SWAP unprime               \ drop leading "-"
  DDUP MAX-ROT MIN          \ sort factors
  factor{ SWAP } $.

```

```

ASCII .EMIT factor{ SWAP } $. -1 IS sign ;

: clean ($adr --)           \ unprime all factors
  COUNT OVER + SWAP
  DO 1 C@ unprime 1 CI LOOP ;

: rearrange ( a b -)
  1 BUF$ ! \ init buf$
  BUF$ XS $+ PAD BUF$ $! \ buf$ -> _/x/
  prime! ( -- a b )
  BUF$ COUNT + CI BUF$ 1+CI \ buf$ -> _/x/b'
  64 XOR ( -- a xor 64) \ toggle sign of a
  BUF$ 1+ CI ; ( -- ) \ buf$ -> ^a/x/b'

: setup XS tokenize          \ form tokens
  $stack 0!                   \ init $stack
  0 IS sign                  \ init sign flag
  BUF$ $push ;               \ input tokens on $stack
\ debugging code
:$$. ($adr --) COUNT OVER + SWAP \ translate tokens
  DUPC@ .sign factor{ OVER C@ unprime } $. ." "
  1+ DO factor{ 1 C@ unprime } $. .
  1 C@ 64 AND 0 > ( -- ) \ primed?
  IF ." ." THEN ." "
  LOOP ;
: dump$stack? DEBUG NOT IF EXIT THEN
  $stack 2+ ( -- $adr[0])
  $stack @ DUP 0 = \ $stack empty?
  IF DDROP CR ." $stack empty" EXIT THEN
  0 DO 1 20 * OVER + CR $$. LOOP DROP ;
0 VAR DEBUG
: DEBUG-ON -1 IS DEBUG ;
: DEBUG-OFF 0 IS DEBUG ;
\ end debugging code
: ( ." ( 0 IS sign ;
: ) ." ) -1 IS sign ;
: parse dump$stack?
  BUF$ $pop 2factors ( -- ab)
  DUP 64 AND 0 > ( -- ) \ b = b'?
  IF DDROP EXIT THEN
  DDUP AB ( -- ab) \ output
  XS C@ 0 > ( -- ) \ more?
  IF rearrange \ buf$ = ^a/x/b'
    BUF$ $push \ ^a/x/b'
    XS clean XS $push \ x/
    ( RECURSE ) .
    RECURSE
  ELSE DOROP THEN ;
: TR($get$ \ input $
  setup
  ." = CR \ for beauty
  parse ;

```

Since the tokens are 1-byte integers smaller than 32, their 5th, 6th and 7th bits can serve as flags to indicate their properties. For example, we need to indicate whether a factor was "starred", i.e. whether it represents $(A + m_A)$ or A , according to the rule

$$\backslash *A \rightarrow (A/ + m[A]).$$

Again, we need to be able to indicate a "prime", showing that a factor has been permuted following the rule

$$\backslash \text{Tr}(a/b/\text{prod}/) \rightarrow a.b (\text{tr}(\text{prod}/)) - \text{tr}(a/\text{prod}/b/)$$

Thus we set bit 7 (**128 OR**) to indicate "star", and bit 6 (**64 OR**) to indicate "prime".

We still need to indicate the leading sign. My first impulse was to use bit 5, but I realized the first factor is never permuted, hence its 6th bit is available to signify the sign. It is toggled by the phrase **64 XOR**. (In the \$stack pictures appearing in the Figures we indicate toggling by a leading " \sim ".)

Programming these aspects is fairly trivial so we need not dwell on it. The entire program appears on pages 279 and 280 above.

Now we test the program:

$$\begin{aligned}\text{TR}(A/B/C/D/E/F) = \\ A.B(C.D(E.F)-C.E(D.F) + C.F(D.E)) \\ -A.C(D.E(B.F)-D.F(B.E) + B.D(E.F)) \\ +A.D(E.F(B.C)-B.E(C.F) + C.E(B.F)) \\ -A.E(B.F(C.D)-C.F(B.D) + D.F(B.C)) \\ +A.F(B.C(D.E)-B.D(C.E) + B.E(C.D)) \text{ ok}\end{aligned}$$

Clearly the concept works. Our next task is to incorporate branches to take care of "starred", as well as identical and/or orthogonal adjacent factors. The possible responses to the different cases are presented in decision-table form in Table 11-2:

To avoid excessively convoluted logic we eschew nested branching constructs. A finite state machine would be ideal for clarity; however, as Table 11-2 makes clear, the logic is not really that of a FSM, besides which, the FSM compiler described above would

Input:	a/b/x/	*a/b/x/	a/*b/x/	a/a/x/	a/b/x/, a.b=0
Resulting stack:	$\sim a/x/b'/$	a/b/x/	a/b/x/
Action(s) [†] :	x/	b/x/	a/x/	x/	$\sim a/x/b'/$
	a.b	m[a]	m[b]	a.a	RECURSE
	(RECURSE)	(RECURSE)	(RECURSE)	(RECURSE)	
	RECURSE	RECURSE	RECURSE		

Note: characters shown in light typeface are EMITted.

have to be modified to keep its state variable on the stack, since otherwise it could not support recursion. The resulting pseudo-code program is shown in Fig. 11-5. Implementing the code is now straightforward, so we omit the details, such as how to define **PERP** to appropriately mark the symbols. The simplest method is a linked list or table of some sort, that is filled by **PERP** and consulted by the test word **perps?**.

How might we implement a leading factor of γ^5 ? While there is no difficulty in taking traces of the form

$$\text{Tr}(\gamma^5 A B C D \dots) \equiv i \text{Tr}(\epsilon_{\mu\nu\rho\lambda} A^\mu B^\nu C^\rho D^\lambda \dots), \quad (11)$$

expressions with γ^5 between “starred” factors are more difficult. However, the permutation properties of traces let us write, e.g.,

$$\begin{aligned} & \text{Tr}((A + m_A)(B + m_B)\gamma^5 C(D + m_D)\dots) \\ & \equiv \text{Tr}(\gamma^5 C(D + m_D)\dots(A + m_A)(B + m_B)), \end{aligned} \quad (12)$$

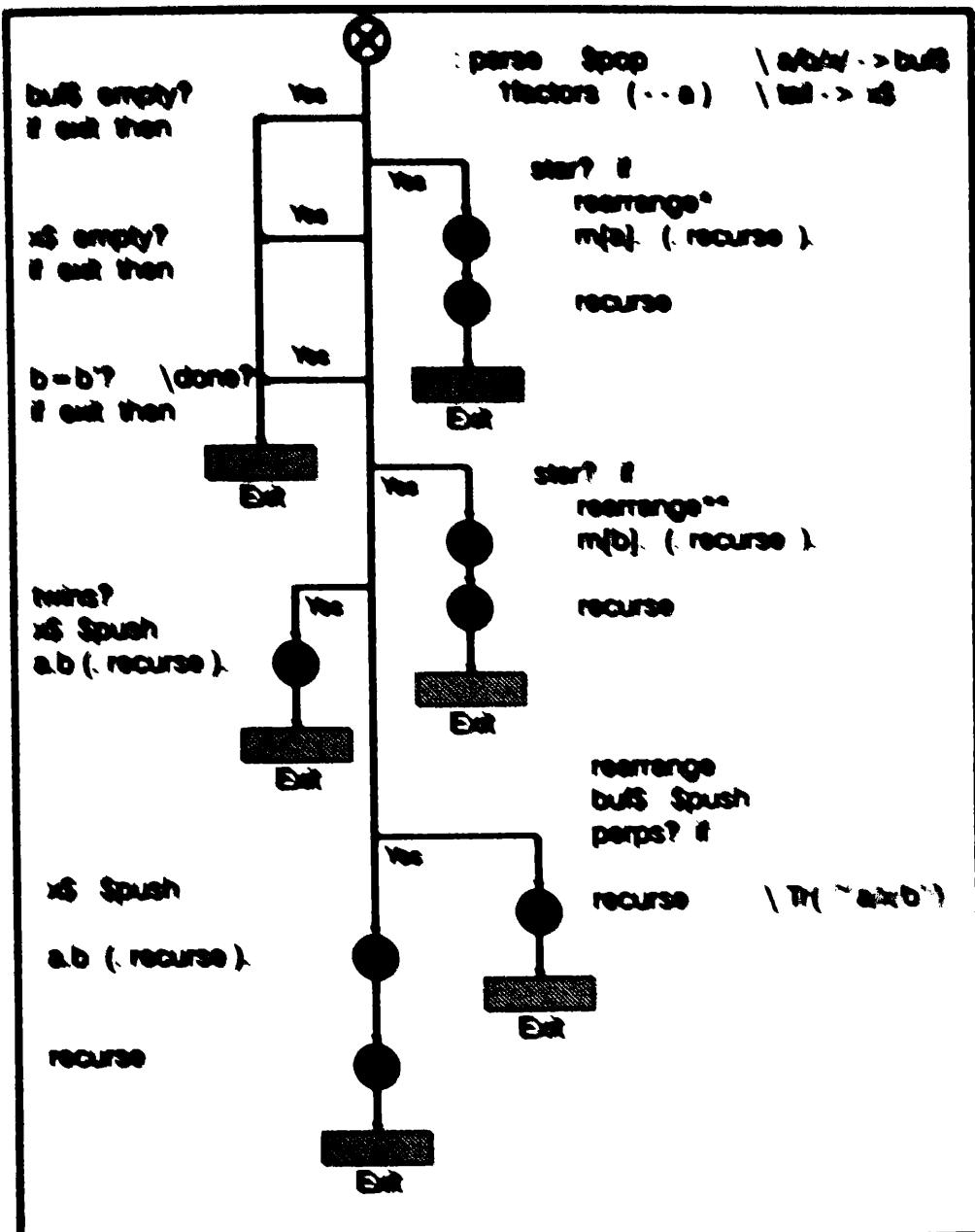


Fig. 11.4 Pressure-flow diagram for eastern 'Dense'

so the key issue becomes decomposing leading started factors, moving the pieces on the Stock, until at least three distinct unstarted factors are adjacent on the left. Replace these by a special

token which stands for “ \wedge ” as shown on page 276. This token is marked orthogonal to all three of the vectors it represents, at the time it is inserted.

To avoid further extending **parse**, probably the best scheme is to define a distinct word, **Trg5(**, that uses the components of **parse** to perform the above preliminary steps. Then **Trg5(** will invoke **parse** to do the rest of its work.

The only other significant task is to extend the output routine to

- a) recognize the special “ \wedge ” token; and
- b) replace dot products like “ $\wedge .d$ ” by **[a,b,c,d]**.

A final remark: one or another form of vectoring can simplify **parse** (relative to Fig. 11-5) by hiding the recursion within words that execute the branches. We have avoided this method here because it conceals the algorithm, a distinct pedagogical disadvantage.

FORmula TRANslator

That prehistoric language FORTRAN – despite its manifold deficiencies relative to FORTH – contains a useful and widely imitated invention that helps maintain its popularity despite competition from more modern languages: This is the FORmula TRANslator from which the name FORTRAN derives.

FORTH’s lack of FORmula TRANslator is keenly felt. Years of scientific FORTH programming have not entirely eliminated my habit of first writing a pseudo-FORTRAN version of a new algorithm before reexpressing it in FORTH. Sometimes I will even write a test program in QuickBASIC® before re-coding it in FORTH for speed and power, just to avoid worry about getting the arithmetic expressions correct.

§§1 Rules of FORTRAN

A FORmula TRANslator provides a nice illustration of rule-based programming. To maintain portability, we employ the standard FORTH kernel, omitting special HS/FORTH words as well as CODE words.

In principle we could provide a true compiler that translates formulae to machine code (or anyway, to assembler). But unless we use p-code or some such artifice²¹ we would lose all hope of portability. Thus, we take instead the simpler course of translating FORTRAN formulae to FORTH according to the rules

\ NUMBERS:

```

\ <int>      -> {-| Q} {digit digit^8}
\ <exp't>    -> {dDeE}{& | Q} {digit digit^2} | Q}
\ <fp#>      -> {-| Q} {dig | Q} . dig^ <exp't>

```

\ FORMULAS:

```

\ <assignment> -> <subject> = <expression>
\ <id>          -> letter {letter|digit}^6
\ <subject>     -> <id> {<idlist> | Q}
\ <idlist>       -> ( <id> { , <id>} ^ )
\ <arglist>      -> ( <expr'n> { , <expr'n>} ^ )
\ <function>    -> <id> <arglist>
\ <expression>   -> <term> | <term> & <expr'n>
\ <term>         -> <factor> | <factor> % <term>
\ <factor>       -> <id> | <fp#> | ( <expr'n> ) | 
                    -> <factor> ** <factor>

```

Clearly, the FORTH FORmula TRANslator could become the kernel of a more complete FORTRAN->FORTH filter by adding to the above rules for formulae the following rules for loops and conditionals:

\ DO LOOPS:

```

\ <label>      -> <integer>
\ <lim>        -> {<integer> | <id> }
\ <step>        -> { , <lim> | Q}
\ <do>          -> DO <label> <id> = <lim> , <lim> <step>

```

21. That is, code for an artificial, generic machine whose code can be mapped easily onto the instruction set of a real computer.

```

\ BRANCHING STRUCTURES:
<logical expr> -> <factor> .op. <factor>
<if0>           -> IF(<logical expr>) <assignment>
<if1>           -> IF(<logical expr>) THEN
                     { <statement> } ^
                     END IF

<if2>           -> IF(<logical expr>) THEN
                     { <statement> } ^
                     ELSE
                     { <statement> } ^
                     END IF

<if3>           -> IF(<logical expr>) THEN
                     { <statement> } ^
                     ELSEIF(<logical expr>)
                     { <statement> } ^
                     END IF

```

§§2 Details of the Problem

The general principles of compiler writing are of course well understood and have been described extensively elsewhere. Several computer science texts expound programs for formula evaluators²². Once we have our translator, we can easily make it an evaluator by compiling the FORTH as a single word, then invoking it.

Let us proceed by translating a FORTRAN formula into FORTH code by hand. For simplicity, ignore integer arithmetic and assume all literals will be placed on the intelligent floating point stack (ifstack). Similarly assume all variable names in the program refer to SCALARS (see Ch. 5). A word that has become fairly standard is %, which interprets a following number as floating point, and places it on the fstack. With these conventions, we see that we shall want to translate an expression like

22. See, e.g., R.L. Kruse, *Data Structures and Program Design*, 2nd Ed. (Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987).

$A = -15.3E7 * EXP(7/X) + Z/(W * SIN(THETA * PI/180)/4)$

into (generic) FORTH something like this:

```
% 4 REAL*8 >FS  
% 180 REAL*8 >FS  
PI G\  
THETA >FS G*  
GSIN G\  
W >FS GR-  
Z >FS G\  
X >FS  
% 7 REAL*8 G\  
GEXP  
% -15.3E7 REAL*8 >FS  
G*  
G+  
A FS>
```

Begin with the user interface. We will define a word, **F***, that will accept a terminated string and attempt to translate it to FORTH. That is, we might say

F* A = -15.3E7 * EXP(7/X) + Z/(W * SIN(THETA * PI/180)/4)"

and obtain the output (actual output from the working program!)

```
% -15.3E7 REAL*4 F>FS % 7 REAL*8 F>FS  
X >FS G/ GEXP G* Z >FS W >FS  
THETA >FS PI >FS G* % 180 REAL*8 F>FS  
G/ GSIN % 4 REAL*8 F>FS G/ GNEGATE  
G+ G/ G+ A FS> ok
```

Although the second version differs somewhat from the hand translation, the two are functionally equivalent.

We would also like to have the possibility of compiling the emitted FORTH words, if **F*** appears within a colon definition, as in

```
: do.B    F* B = 39.37/ATAN(X**W) + 7*Z/X" ;
```

A FORTRAN expression obeys the rules of algebra in a generally obvious fashion. Parentheses can be used to eliminate all ambiguity and force a definite order on the evaluation of terms

and factors. However, to reduce the number of parentheses, FORTRAN adopted a hierarchy of operators that has been followed by all other languages that incorporate semi-algebraic replacement statements like the above. The hierarchy is

0. FUNCTION
1. EXPONENTIATION (\wedge or $**$)
2. $*$ or $/$
3. $+$ or $-$
4. “ , ” (argument separator in lists)

The translator must both enforce these rules and resolve ambiguities involving operators at the same hierarchical level. Thus, e.g., does the fragment

A/B*C

mean A/(B*C) or (A/B)*C ? Many FORTRAN compilers follow the latter convention, so we will maintain this tradition.

A second issue is the function library. The FORmula TRANslator must recognize functions, and be able to determine whether a given function is in the standard library. In the example above, F^t recognized EXP and SIN as standard library functions and emitted the FORTH code to invoke them. A beauty of FORTH is that there are several easy ways to accomplish this, using components of the FORTH kernel.

A third issue is the ability of a true FORTRAN compiler to perform mixed-mode arithmetic, combining INTEGER*2, INTEGER*4, INTEGER*8, REAL*4, REAL*8, COMPLEX*8 and COMPLEX*16 types *ad libitem*. FORTRAN does this using the information contained in the type declarations at the beginning of a routine. A pure FORmula TRANslator has no such noncontextual information available to it, hence has no way to decide how to insert the proper FORTH words during compilation. To get around this we employ the generic data and operator conventions developed in Chapter 5 §1.

11.3 Parsing

Let us hand-parse the example, reproduced below:

A = -15.3E7*EXP(7/X) + Z/(W-SIN(THETA*PI/180)/4)

Clearly, we must apply the first rule

\ <assignment> -> <subject> = <expression>

embodied in the word <**assignment**>. We split at the “=” sign, and interpret the text to its left as a **SCALAR**. Since we want to emit the phrase **A FS>** last, yet have parsed it first, we have to hold it somewhere. Clearly the buffer where we store it will be a first-in last-out type; and by induction, last-in, first-out also. But a **LIFO** buffer is a stack. Hence the fundamental data structure needed in our parsing algorithm is a string stack. So we might imagine that after the first parsing step the string stack contains two strings

\$STACK

Notes

A FS>

-15.3E7*EXP(7/X) + Z/(W-SIN(THETA*PI/180)/4)

\ <subject>
\ <expression>

Next we apply the rule

\ <expr'n> -> <term> | <term> & <expr'n>

This breaks the top expression at the + sign between “)” and **Z**. We should think of the two terms

-15.3E7*EXP(7/X)

and

Z/(W-SIN(THETA*PI/180)/4)

as numbers on the ifstack; hence the code to evaluate each should be emitted before the addition operator (that is, these expressions are higher on the string stack than the addition operator **G+**). We adopt a rule that the right term is pushed before the left, so the \$stack now looks like

<u>\$STACK</u>	<u>Notes</u>
A FS > Z/(W-SIN(THETA*PI/180)/4) G + -15.3E7*EXP(7/X)	\ <subject> \ <term> \ <term>

We now anticipate a new problem: suppose we have somehow — no need to worry about details yet — emitted the code for the **<term>** **-15.3E7*EXP(7/X)** on top of the \$stack. Then we would have to parse the line **Z/(W-SIN(THETA*PI/180)/4) G +**. Assuming the program knows how to handle the first part, **Z/(W-SIN(THETA*PI/180)/4)**, how will it deal with the **G +**? We do not want to use the space as a delimiter (an obvious out) because this will cause trouble with **A FS >**.

The difficulty came from placing **G +** on the same line as **-15.3E7*EXP(7/X)**. What if we had placed the operator on the line above, as in

<u>\$STACK</u>	<u>Notes</u>
A FS > G + Z/(W-SIN(THETA*PI/180)/4) -15.3E7*EXP(7/X)	\ <subject> \ operator \ <term> \ <term>

Eventually we see this merely exchanges one problem for another of equal difficulty: How do we distinguish a **<factor>** or **<term>** that contains no more operators or functions — and is therefore ready to be emitted as code — from the operator **G +**, which contains a “+” sign? Now we need complex expression recognition, which will lead to a slow, complicated program.

When this sort of impasse arises (and I am pretending it had been realized early in the design process, although the difficulty did not register until somewhat later) it signals that a key issue has been overlooked. Here, we failed to distinguish FORTH words, **FS >** and **G +**, from FORTRAN expressions. We have, in effect, mixed disparate data types (like trying to add scalars and vectors). Worse, we discarded too soon information that might

have been useful at a later stage. This leads to a programming tip, *a la Brodie*²³:

| **TIP:** Never discard information. You might need it later.

Phrased this way, the solution becomes obvious: keep the operators on a separate stack, whose level parallels the expressions. So we now envision an expression stack and an operator stack, which we call E/S and O/S for short. On two stacks,

E/S	O/S	Notes
A	FS >	\ <subject>
Z/(W-SIN(THETA*PI/180)/4)	G +	\ <term>
-15.3E7*EXP(7/X)	NOP	\ <term>

Why the NOPs ("no operation") on the O/S? We want to keep the stack levels the same (so we do not have to check when POPping off code strings); we thus have to put NOP on the O/S to balance a string on the E/S.

The TOS now contains a <term>, so we apply the rules

```
\ <function>      -> <id> <arglist>
\ <term>          -> <factor> | <factor> % <term>
\ <factor>-> <id> | <fp#> | (<expr'n>) | <func>
```

We note there is an operator at the " %" priority level (the "*" in the TOS). We split the top <term> at this point, issuing a G*.

E/S	O/S	Notes
A	FS >	\ <subject>
Z/(W-SIN(THETA*PI/180)/4)	G +	\ <term>
EXP(7/X)	G*	\ <term>
-15.3E7	NOP	

23. Leo Brodie, *Thinking FORTH* (Prentice-Hall, Inc., NJ, 1984).

The parsing has now reached a turning point: the top string on the E/S can be reduced no further. The program must recognize this and emit the corresponding line of code (see Ch. 5):

% -15.3E7 REAL*4 F>FS
leaving

E/S	O/S	Notes
A	FS >	\ <subject>
Z/(W-SIN(THETA*PI/180)/4)	G +	\ <term>
NULL	G *	
EXP(7/X)	NOP	\ <function>

What is **NULL** and why have we pushed it onto the E/S? Simply, it is not yet time to emit the **G*** so we have to save it; however, we have another operator, **G +**, to associate with $Z/(W-\text{SIN}(\text{THETA}*\text{PI}/180)/4)$. Thus we have no choice but to define a placeholder for the E/S, analogous to **NOP** on the O/S.

TOS now contains a function. Assuming we can recognize it as such, we want to check that it is in the library and put the correct operator on the E/S. Thus we want to decompose to

E/S	O/S	Notes
A	FS >	\ <subject>
Z/(W-SIN(THETA*PI/180)/4)	G +	\ <term>
NULL	G *	
NULL	GEXP	\ <function>
(7/X)	NOP	\ <arglist>

The parentheses around the **<arglist>** on TOS serve no purpose, so drop them.

We see, once again, an operator of the priority-level **%** (the “/” between **7** and **X**), so we again apply the rule

\ <term> -> <factor> | <factor> % <term>

to obtain

E/S	O/S	Notes
A	FS >	\ <subject>
Z/(W-SIN(THETA*PI/180)/4)	G +	\ <term>
NULL	G*	
NULL	GEXP	
X	G/	\ <id>
7	NOP	\ <fp#>

Once again we can emit a number, so we do it:

% 7 REAL*8 F > FS

Wait! Why did we say REAL*4 with -15.3E7, but REAL*8 with 7 just now? Can't we make up our minds? The answer is that we want to respect precision over-rides via FORTRAN's E (single precision, so we say REAL*4) or D (double precision – REAL*8) exponent prefixes. However, where we are free to choose, it makes sense to keep maximum precision.

We continue, emitting the next simple items on the \$stack:

X G/ GEXP G*

leaving

E/S	O/S	Notes
A	FS >	\ <subject>
Z/(W-SIN(THETA*PI/180)/4)	G +	\ <term>

Once again we find the most exposed operator to be “ / ”, which we split with the rule

\ <term> -> <factor> | <factor> % <term>

E/S	O/S	Notes
A	FS >	\ <subject>
NULL	G +	\ <term>
(W-SIN(THETA*PI/180)/4)	G/	\ (<expr>)
Z	NOP	

Emit the TOS:

Z > FS

and apply the rule (first drop the parentheses)

\ <expr'n> -> <term> | <term> & <expr'n>

E/S	O/S	Notes
A	FS>	\ <subject>
NULL	G+	\ <term>
NULL	G/	
-SIN(THETA*PI/180)/4	G+	
W	NOP	

Why did we issue **G +** and keep the leading “-” sign with **SIN**? Simple: any 9th grader can tell the difference between a “-” binary operator (**binop**) and a “-” unary operator (**unop**) in an expression. But, while not impossible, it is unnecessarily difficult to program this distinction. The FORTH philosophy is “Keep it simple!” Simplicity dictates that we embrace every opportunity to avoid a decision, such as that between “-” binop and “-” unop. The algebraic identity

$$X - Y \equiv X + (-Y)$$

lets us issue only **G +**, as long as we agree always to attach “-” signs as unops to the expressions that follow them. Eventually, of course, we shall have to deal with the distinction between negative literals (**-15.3E7**, e.g.) and negation of variables. The first we can leave alone, since the literal-handling word **%** (“treat the following number as floating point and put it on the 87stack”) surely knows how to handle a unary “-” sign; whereas the second case will require us to issue a strategic **GNEGATE**.

A consequence of this method for handling “-” signs is that the compiler will resolve the ambiguous expression

$$-X^Y \stackrel{?}{=} -(X^Y) \text{ or } (-X)^Y$$

in favor of the former alternative. If the latter is intended, it must be specified with explicit parentheses.

After sending forth the phrase

W >FS

the leading “-” preceding **SIN(...)/4** must be dealt with. To preserve the proper ordering on emission we will want a word **LEADING-** that puts the token for **GNEGATE** on the O/S and moves the string **SIN(THETA*PI/180)/4** to the TOS, issuing a **NOP** on the E/S, obtaining

E/S	O/S	Notes
A	FS >	\ <subject>
NULL	G +	\ <term>
NULL	G /	
NULL	G +	
NULL	GNEGATE	
SIN(THETA*PI/180)/4	NOP	

The next exposed operator is at “ % ” -level. We apply **<term>** once more, to get:

E/S	O/S	Notes
...	...	\ ...
NULL	GNEGATE	
4	G /	
SIN(THETA*PI/180)	NOP	\ (<expr'n>)

After handling the function as before we find the successive stacks and FORTH code emissions

E/S	O/S	Notes
A	FS >	\ <subject>
NULL	G +	\ <term>
NULL	G /	
NULL	G +	
NULL	GNEGATE	
4	G /	
NULL	GSIN	
(THETA*PI/180)	NOP	

E/S	O/S	Notes
A	FS >	\ <subject>
NULL	G +	\ <term>
NULL	G /	
NULL	G +	
NULL	GNEGATE	
4	G /	
NULL	GSIN	
180	G /	
PI	G *	
THETA	NOP	

THETA >FS PI >FS G* % 180 REAL*8 F>FS
 G/ GSIN % 4 REAL*8 F>FS G/ GNEGATE G+
 G/ G+ A FS > ok

§§4 Coding the FORmula TRANslator

We proceed in the usual bottom-up manner. The first question is how to define the \$stack. In the interest of brevity, I chose not to push the actual strings on the E/S, but rather pointers to their beginnings and ends. By using a token to represent the operator, we can define a 6-byte wide stack which will point to the text of interest (which itself resides in a buffer), and will hold the token for the operator at the current level. This way only one stack is pushed or popped and the levels never get out of synchronization.

Again at the lowest level, we can develop the components that recognize patterns, e.g., whether a piece of text is a floating point number. The word that does the latter is **fp#?**, already described in §2§§1.

A function is defined by the rule

\ <arglist>	-> (<expr'n> { , <expr'n> } ^)
\ <function>	-> <id> <arglist>

We may therefore identify a function by splitting at the first left parenthesis,

```
: > ( ($end $beg -- $end $beg') \ find first "("
1- BEGIN 1+ DUPC@ ASCII( = > R
DDUP = R> OR UNTIL ;
```

and then applying appropriately defined FSMs to determine whether the pieces are as they should be.

```
: <function> ($end $beg -- f)
DUP>R > ( -- ($end $beg')
UNDER 1- R> <id>
-ROT SWAP <arglist> AND ;
```

The FSM **<arglist>** must be smart enough to exclude cases such as

$\text{SIN}(A + B)/(C - D)$

and

$\text{SIN}(A + B)/\text{EXP}(C - D)$

that is, compound expressions that might contain functions; it must also correctly recognize, e.g.,

$\text{SIN}((A + B)/\text{EXP}(C - D))$

as a function.

In FORTH there is no distinction between library functions and functions we define ourselves. In either case, the protocol defined in Chapter 8 will work fine. Thus the code generator for **<function>** emits the code

USE(fn.name arg1 arg2 ... argN F(x)

What, however, do we do about translating standard FORTRAN names such as SIN, COS and EXP to their generic FORTH equivalents? The simplest method defines words with the names of the FORTRAN library functions. The FORTH-83 word FIND locates the code-field address of a name residing in a string. Thus we could have (**note**: .GSIN is a CONSTANT containing a token)

.GSIN CONSTANT SIN	\ etc.
--------------------	--------

```
: LIBRARY? ( $end $beg -- cfa | 0 )
-ROT UNDER - DUP >R
PAD 1+ SWAP CMOVE R> PAD CI \ make $
PAD FIND ( -- cfa n) 0= NOT AND ;
```

now we can define **function!** which, assuming pointers to the <id> and <arglist> are on the stack, rearranges the \$stack like this:

E/S	O/S	Notes
... (arg1, arg2, ..., argN) NULL	.. .F(X) .lib_name	\ an op. \ if library function
or, if it is a user-defined function, like this:		

E/S	O/S	Notes
... (arg1, arg2, ..., argN) name	.. .F(X) NOP	\ an op. \ user function

The code that does all this is

```
: function! ( $end2 $beg2 $end1 $beg1 -- )
.F(X) $push \ push arglist
DDUP LIBRARY? DUP 0=
IF DROP .NOP $push \ user fn
ELSE EXECUTE \ in lib.
NULL ROT $PUSH DDROP
THEN ." USE( " ;
```

For the program itself²⁴ we work from the last word, <**assignment**>, to the first (which we do not know the name of yet). We shall describe the program in pseudocode only, in the interest of saving space. Clearly,

```
: < assignment > \ input $ assumed in buffer
< subj > = < expr > \ split at "=" ( -- f )
IF subj! THEN \ put subj and its code on $stack
.NOP $push ; \ then put expr on $stack
```

Certain decisions need to be made here: for example, do we want F# to be able to parse an expression that is *not* an assignment (that is, generating code which evaluates the expression and leaves the result on the ifstack)? We allowed this with the IF...THEN.

Next we pseudocode <expression>:

```
: <expression> empty? IF EXIT THEN
  $pop
  <trm> & <expr>      ( - - ! ) \ split at &
  IF trm&expr! RECURSE
  ELSE NULL ROT $push
    .NOP $push <term>
  THEN ;
```

Defined recursively in this way, <expression> will keep working on the TOS until it has broken it up into term s.

We similarly define <term> recursively, so it will break up any term s into all their factors. It should also recognize <arglist>s. Thus:

```
: <term> empty? IF EXIT THEN
  $pop <arglist>
  IF arglist! <expression> EXIT THEN
    <factor>%<term>      ( - - ! ) \ split at % = "*/"
    IF fct%trm! RECURSE
    ELSE .NOP $push <factor> \ term = factor
  THEN ;
```

And finally, we define <factor>, again recursively,

```
: <factor> empty? IF EXIT THEN      \ done
  $POP <fp#>      \ fp#?
  IF fp#! RECURSE EXIT THEN
  leading -?
    IF leading-! <expression> \ forward ref.
    EXIT THEN
  <id> IF id! <expression> \ forward ref.
    EXIT THEN
  <f> ^ <f>      \ exponent?
    IF f^f! RECURSE RECURSE
    EXIT THEN
\ cont'd ...
```

```
<function>
    IF function! <expression>      \ forward ref.
    EXIT THEN
    (<expression>)          \ expression inside ( )
        IF exposel <expression>      \ forward ref.
        ELSE ." Not a factor!" ABORT THEN ;
```

Note the forward references found in **<factor>**; since **<expression>** is defined later, we must use vectored execution or some similar method to permit this recursive call.

With this we conclude our discussion of rule-based programming. The complete code for the FORmula TRANslator is too lengthy to print, hence it will be found on the included diskette.

		assembler ... X*	150
		avoiding decisions	97, 271
		B	
		BEHEAD" See beheading, headerless words	
		beheading	
		BEHEAD' and BEHEAD" 98 See also BEHEAD', BEHEAD" (words)	
		headerless words 41	
		binary search See table look-up	
		BIOS 250	
		bit-reversal See FFT	
		Brodie, L. 12	
		C	
		CALL 8, 16, 41, 43	
		Cauchy's theorem calculus of residues 180	
		CFA	
		vectored execution 35	
		chi-squared/degree of freedom 200	
		CODE See assembler	
		commenting See (, \ (words)	
		fstack comment 43	
		stack comments 42	
		comparing numbers See = (words)	
		relational operators 24	
		compile-time actions See : (colon) (words)	
		compiler compiled languages 2	
		See CREATE,], [(words)	
		compiling word See defining word, CREATE ... DOES >	
		complex conjugate See complex numbers, Hermitian matrix	
		complex functions exponential 154	
		logarithm 155	
		trigonometric 154	
80486DX cpu/fpu	66, 72		
80x87 chips			
See also 87stack			
80x87 co-processor family	65		
87stack	48, 67, 80		
>R			
See also R >			
A			
Abramowitz, M. and Stegun, I.A.	182		
absolute address	107		
action table	34 - 35		
See also execution array			
adaptive integration			
recursive version	165		
algebra			
See computer algebra			
ANSII FORTH	12, 24		
Argand			
-plot, -diagram, -plane	146		
arithmetic operators	24		
array notation	105		
arrays			
bounds checking	44		
FORTRAN notation	5		
assembler	52, 67		
<% ... %>	69		
CODE ... END-CODE	67		
DWORD-PTR	71		
FCOM.	74		
FCOMP.	74		
FCOMPP.	74 - 75		
FDECSTP.	80		
FILD.	70 - 71		
FINCSTP	80		
FISTP.	70 - 71		
FLD.	71		
FSTP.	71, 75		
FTST.	74		
FWAIT.	68		
FXCH.	68		
QWORD-PTR	71		
TBYTE-PTR	71		
WORD-PTR	70		

complex lexicon		
1/X	150	
ARG	150	
CMPLX	149	
CONJG	149	
DX@, DX!	148	
F*X	149	
FATAN2	150	
FSINCOS	150, 155	
IMAG	149	
to POLAR	150	
REAL	149	
X*	149	
X* (CODE version)	150	
X*F	149	
X*I	149	
X +	149	
X/	150	
X/F	149	
X2/	155	
X@, X!	148	
X-	149	
XABS	150	
XCOS	155	
XDROP	149	
XDUP	53, 78, 149	
XEXP	155	
XLOG	155	
XMODSQ	150	
XOVER	149	
XSIN	155	
XSQRT	154	
XSWAP	149	
complex numbers	144	
2×2 matrix representation	145	
–n roots of	151	
algebraic rules	145	
Cartesian representation	144, 147	
complex conjugate	146	
division	146	
multiplication	145	
multiplicative inverse	146	
polar representation	147	
real and imaginary parts	144	
roots of polynomials	144	
square roots	152 - 153	
computer algebra		
\$stack	278	
PERP	282	
TR(277	
Trg5(284	
contour integral		
See adaptive numerical quadrature		
CPDUP		
See XDUP		
Cramer's rule		
See determinants		
D		
danger		
See rstack		
Data structures		91 - 114
DEBUG		66, 69
logging output to file		257 - 258
defining words		28
See CREATE, DOES		
definite Integral		158
as limiting process		159
DERIVE		
See computer algebra		
determinants		
Cramer's rule		218
dictionary		13, 15, 28, 35, 39, 41
differential equation		
numerical quadrature as		160
differential equations		
See ordinary differential equations		
dimensioned data		
intrinsic units		32
division		
See multiplicative inverse		
documenting code		
See commenting; (, \ (words)		
self-documenting style		269
DOER ... MAKE		
See also vectored execution		
DOS functions		255
dummy words		
See CREATE, vectored execution		
E		
eigenvalue problems		
solution of		217
ESC (D8hex)		
See escape		
"escape" instruction		66
Euclid's algorithm		
GCD		166 - 167

Euler's theorem	151, 154	$F >, F =, F <$	53, 75
execution array	97	$F0 <, F0 =, F0 >$	53, 75
F		$F = 0$	48
		$F = 1$	48
		$F = 2$	76
F\		$F = L10(2) (0.3010300...)$	48
See FR/		$F = L2(10) (3.321928...)$	48
Fast Fourier Transform (FFT)	184	$F = L2(E) (1.442685...)$	48
fetch		$F = LN(2) (0.693147...)$	48
See @, D@, R32@, @L, etc. (words)		$F = PI (3.14159...)$	48
FFT			
bit-reversal	187, 189	$F@, F!$	47
FFT algorithm	186	FDUP	47, 67, 70
Fibonacci sequence		FSWAP	47, 67, 70
See recursion		FDROP	47, 67, 70
FIG		FROT	47, 67-68
Forth Interest Group	12	F-ROT	70
file-oriented FORTHs	250	FOVER	47, 67, 70
finite state machine (FSM)		S->F	47, 71
FORTH version	269	D->F	47
See rule-based programs		F->S	47, 71
tabular representation	267	F->D	48
fitting functions		%	48
determining parameters	183	FUNDER	48
See "linear, nonlinear least-squares"		FPLUCK	48
representation of data	181	FnX	48
FnR		FnR	48
floating point arithmetic	45		
floating point lexicon			
DEG->RAD	54, 77	FACOS	54, 80
F*	72	FASIN	54, 80
F**	54, 77	FATAN	54, 80
F+	49, 72	FATAN2	150
F-	49, 72		
FR-	49, 72	SINH	55
F*	49	COSH	55
F/	49, 72	TANH	55
FR/ (synonym F\)	49, 72	ARCSINH	55
FNEGATE	49, 72	ARCCOSH	55
FABS	49, 72		
1/F (synonym FINV)	49	FICONSTANT	56
F*NP	49	FINIT	57
F**2	52		
F2**	54, 77	FCOMPP	75
		FCONSTANT	77
F,	77	FCOS	79
F2/	55	FEXP	54, 77
F32,	63	FLG	76
		FLN	54, 77
F2XM1	76	FLOG	54, 77
F4x4 (IIT chip)	85	FMAX	53

floating point lexicon ...		FORTRAN, rules of
FMIN	53	See rule-based programs
FPATAN	77, 79	fstack
FPOP	82	software 87stack extension
FPREM	78	80
FPTAN	77	function library
FPUSH	82	6, 53
FTRUNC	57	function protocol
FRNDINT	57	USE(
FSQRT	54, 76	162
		fundamental theorem of calculus
		159
		G
FSBP0, FSBP1, FSBP2		gamma-5 (γ^5)
IIT chip instructions	85	special gamma matrix
FSCALE	77	276
FSIN	54, 79	Gaussian quadrature
FSGN	54, 72	See numerical integration
FCOS	54	generic lexicon
FTAN	54, 79)MONTE
RAD-> DEG	54	1162
		1ARRAY
FSTSW	74	111
FTST	75	2ARRAY
FTSTP	75	113
		?TYPE
FY*LG2X	76	113
FY*LG2XP1	76	}
		}}
I16@, I16!	70	binary operators
I32@, I32!	71	104
I64@, I64!	71	G: ... ;
		97
		See also CASE: ... ;CASE (words)
		G@, G!
		97
		GB:
		104
		GU:
		101
		operators
		100
		unary operators
		101
		generic memory access
		See G@, G!
R32@, R32!	71	GEnie network
R64@, R64!	71	12
R80@, R80!	71	
		globally optimizing compiler
		See optimization
floating point numbers		
testing	53	GOSUB
FORmula TRANslator	296	8, 16, 43
< arglist >	297	GOTO
< assignment >	298	computed, in FORTRAN etc.
< expression >	299	267
< factor >	299	See spaghetti code
< function >	297	
< id >	298	Gram polynomials
< term >	299	184, 193
		Gram-Schmidt orthogonalization
		196
FORTH words		
See words		

H

half-angle formulas	
See trigonometric functions	
Hammersley, J.M. and Handcomb, D.C.	
Monte Carlo Methods	165
hashing	
See table look-up	
Hermitian matrix	
See linear equations	
Hewlett-Packard calculator	17
Hilbert matrix	
See ill-conditioned matrix	
HMF	
See Abramowitz, M. and Stegun, I.A.	

I

FORTH I/O words	
\$"	251
\$->F	225, 251
<FILE	251
>FILE	253
CLOSE-INPUT	251
CLOSE-OUTPUT	253
CLOSEH	257
CRT	257
DUPH	256
file handle number	256
FILL <FILE	253
G#	251
G\$	251
GET-F#	225
IO-STAY	257
LOG-OFF	257
LOG-ON	257
MAKE-OUTPUT	253
MAT >FILE	253
MKFILE	256
NR	251
OPEN-INPUT	251
PCRT	257
SPLICEH	256
splicing file handles	256
VEC >FILE	253 - 254
ifstack	93
typed data stack manager	99 - 100
ill-conditioned matrix	195
Infinite series	116
divergent series	116
exponential	123
harmonic series	117
power series	116, 126
Weierstrass' theorem	121
Zeno's paradox	117
Infix	17
See also postfix	
Information hiding	40
and safety	41
Integral of a function	158
Integrated Information Technology	
802c87 and 80c387 chips	85
Integration	
See also adaptive integration	
Monte-Carlo method	160
numerical	157
"Intelligent" fstack	
See ifstack	
Interpolation	
See fitting functions	
"Interrupt"	
See DOS functions	
Inverse trigonometric functions	77
J	
Jump table	
See also execution array, action table	
in rule-based ...	270
K	
Kelly, M. and Spies, N.	12
L	
large arrays	106
linear equations	89
determinantal equation	216
determinants	214
eigenvalue problems	215
Hermitian matrices	216
linear algebraic equations	214

pivotal elimination	221	N
linear equations ...		
Strassen's algorithm	90	naming conventions
linear least-squares		self-documenting code
Gram polynomials	207	43
literal	3	nonlinear least-squares
local variables		201
rstack used for	22	numerical integration
Loeliger, R.)INTEGRAL
<i>Threaded Interpretive Languages</i>	1	165
logging screen output	254	Gaussian quadrature
logic trees		204
"dead" code	267	See also Integration, numerical
loops	25	Richardson extrapolation
BEGIN ... UNTIL	25	206
BEGIN ... WHILE ... REPEAT	25	trapezoidal rule
endless	26	205
Indexed	26	O
Lukasewcieicz		object oriented
See RPN	17	64, 96
M		offset
MACSYMA		See absolute address
See computer algebra		optimization
MATHEMATICA		51
See computer algebra		peephole
mathematics co-processor	65	53
Mathews, J. and Walker, R.L.		ordinary differential equations
<i>Mathematical Methods of Physics</i>	122	131
matrix, inverse		Implicit Runge-Kutta method
See linear equations		136
mean	59	Runge-Kutta method
minimization		132
)MINIMIZE (simplex algorithm)	210	orthogonal polynomials
– functions of many variables	201	196, 198
simplex algorithm	202	overloading
steepest descents	201	49
monomials	184	See "smart" operators
Monte-Carlo method		P
)MONTE	162	parameter stack
convergence	162	47
See also numerical integration		pattern recognition
multidimensional integrals	162	262
stratified sampling	165	polynomial
uncertainty in	161	Legendre
		7 - 8
		Gram
		184, 193
		polynomial evaluation
		50
		fast algorithm ($\log(n)$ time)
		186
		postfix notation
		17
		Pountain, Dick
		96
		Press, W.H., et al.
		<i>Numerical Recipes ...</i>
		160
		PRN2FILE
		258
		PRNGs
		16-bit Integer PRNG
		62
		random walk
		60
		Wichman and Hill
		60
		FORTH programs
)MINIMIZE
		210
)FFT
		191

FORTH programs			
)FIT	207	reverse Polish notation (RPN)	17
)POLY	51	roots of polynomials	
(pseudo) random numbers	56	See also fundamental theorem of algebra	
chi-squared (χ^2) test	58	roundoff error	195
GGUBS (PRNG)	57	rstack	
Monte-Carlo Integration	162	See return stack	
random data structures	61	rstack, caution in using	
		See also DO ... LOOP (words)	
Q		rule-based programs	
quadrature		\$stack	298
See Integration		skip_exponent	271 - 272
QuickBasic® (Microsoft)	169	automatic translation tables	272
R		deterministic FSMs	265
RAD->DEG		finite state machine	264
See DEG->RAD (floating point ...)		formula evaluator	266
Ralston, A.	136	FORmula TRANslator	284
random numbers		FORTRAN rules	281
See (pseudo) random numbers		FSM:	270
testing PRNGs	58	function library	288
recursion		grammar	260
See also adaptive integration		hierarchy of operators	288
adaptive numerical integration	171	regular expressions	261
disadvantages	171	state transitions	270
elimination of	171	state variable	266, 271
estimating running time	168 - 169	run-time actions	
See also Euclid's algorithm		See DOES > (words)	
Fibonacci numbers	168	Runge-Kutta	
In rule-based programs	262	See ordinary differential equations	
potential problems	167		
RECURSE	170	S	
recursive algorithms	165	say	14
sorting using (mergesort)	169	SCHOONSCIP	
speed of execution	167	See computer algebra	
when not to use	168	segment descriptor	
recursive-descent optimizer	52	See absolute address	
redirection		self-modifying code	28
-of MS-DOS output	254	Shaw, Gordon	114
REDUCE		signed integer	24
See computer algebra		See also >, <, =, + LOOP (words)	
regula falsi		simplex algorithm	184, 201
See transcendental equations		SMALLTALK	
return stack	21	See object oriented	
See also stack		"smart operators"	
		FORTRAN	3
		smart operators	5
		smoothing	
		See fitting functions	
		solving equations	127
		sorting	

		U
spaghetti code	39	unsigned integers
special constants		See also U<, /LOOP (words) 24
math coprocessor chips	48	
stack	14, 17 - 18, 20	
See also 87stack		
See also fstack, Ifstack, parameter stack,		
return stack		
state variable		
See rule-based programs		
store		
See !, DI, R3!, !L, etc. (words)		
Strassen's algorithm	87, 107	
strings	36	
See also FORTH I/O words		
structured programming	25, 38	
style		
FORTH conventions	38	
sub-programs	2	
subroutine		
word as	16, 44	
FORTH subroutines		
}POLY	51	
T		
table look-up	182	
threaded language	1	
TOS		
See stack		
trace		
gamma matrix products with γ^5	276	
of matrix	274	
product of gamma matrices	275	
transcendental equations	127	
binary search	127	
regula falsi	128	
trapezoidal rule		
See numerical integration		
trigonometric functions	77, 152, 184	
- identities	78	
typed data		
arrays	104	
COMPLEX (COMPLEX*8)	94	
DCOMPLEX (COMPLEX*16)	94	
multiple scalars	95	
REAL*4, REAL*8	94	
scalars	94	
variable	3	
		V
		variables
		See also words (VARIABLE), local variables
		variance
		59
		vectorized execution
		28
		analogy with EXTERNAL
		9
		See also DOER ... MAKE,
		CA' x DEFINES y, USE(
		forward referencing
		39, 300
		vocabulary
		67
		W
		Williams, Al
		109
		word
		subroutine
		8, 13, 16
		FORTH words
		49
		! (store)
		23, 29
		<
		15 - 16
		* +
		16
		: (colon)
		15-16
		\$. (string-emit)
		37
		\$"
		37
		(
		42
		*
		24
		*/
		24, 32
		+
		24
		+LOOP
		26
		. (emit)
		14
		/
		24
		/LOOP
		26
		/MOD
		24
		0<, 0=, 0>
		24
		AS
		33
		, (comma)
		29
		<, =, >
		24
		>R
		21
		@ (fetch)
		23, 29
		[(STATE = Interpret)
		34
		\ (comment)
		38, 43
] (STATE = compile)
		34
		- "minus"
		24
		-ROT
		20

ALLOT	28	RDROP	23
BEGIN	28	RECURSE	168, 170, 299
BEHEAD'	34	REPEAT	25
BEHEAD"	57	ROLL	20
C! "byte-store"	23	ROT	19 - 20
C@ "byte-fetch"	23	SWAP	19 - 20
CA' "code-address of"	27	THEN (compile only)	25
CASE: ... ;CASE	268	TRACE, STRACE	168
CONSTANT	28, 30	TYPE	37
COUNT	37	U <	24
CREATE	28, 28, 30	UNDER	20
CREATE ... DOES >	26	UNITS	33
D!	23	UNTIL	25
D > R	21	VARIABLE	22, 28
D@	23	WHILE	25
DDUP	22		
DECIMAL	13	Z	
DEFINES	27		
DO ... LOOP	21, 26	zero-based numbering	18
manipulating rstack within	23		

DOER ... MAKE	44
DOES >	26, 30
DOS"	257
DR >	21
DROP	19
DUP	19 - 20
DUP > R	21
DVARIABLE	29
ELSE (compile only)	25
ENDIF (THEN)	25
EXECUTE	21
EXPECT	37

See also "floating point lexicon"

HEX	14
IF (compile only)	25
IMMEDIATE	270
KEY	37
MAX	53
MIN	53
MOD	24
NUMBER	14
ok	14
OVER	20
PAD	37
PICK	20
+	14
R >	21
R@	

Bibliography

- Aho, A.V., R. Sethi and J.D. Ullman, *Compilers: ...* (Addison-Wesley, Reading, 1988).
- Barnhart, Joe "FORTH and the Fast Fourier Transform" *Dr. Dobb's Journal*, September, 1984, p. 34.
- Basile, F.J. *J. FORTH Appl. and Res.* 1,2 (1982) 76-78.
- Berestetskii, V.B., E.M. Lifshitz and L.P. Pitaevskii, *Relativistic Quantum Theory, Part 1* (Pergamon Press, Oxford, 1971) p. 68ff.
- Berrian, D.W. Proc. 1989 Rochester FORTH Conf. (Inst. for Applied FORTH Res., Inc., Rochester, NY 1989) p. 1-5.
- Bjorken, J.D. and S.D. Drell, *Relativistic Quantum Mechanics* (McGraw-Hill, Inc., New York, 1964)
- Bratley, P., B.L. Fox and L.E. Schrage, *A Guide to Simulation* (Springer-Verlag, Berlin, 1983).
- Brodie, L. *Starting FORTH*, 2nd ed. (Prentice-Hall, NJ, 1986)
- Brodie, L. *Thinking Forth* (Prentice-Hall, NJ, 1984).
- Callahan, J.S. Proc. 1988 Rochester FORTH Conference (Inst. for Applied FORTH Research, Inc., 1988), p. 39.
- Clark, D.D. "Simple Calculations with Complex Numbers", *Dr. Dobb's Journal*, October 1984, p. 30.
- Crawford, John H. and Patrick P. Gelsinger, *Programming the 80386* (Sybex, San Francisco, 1987).
- Duncan, Ray "FORTH support for Intel/Lotus expanded memory", *Dr. Dobb's Journal*, August 1986.
- Duncan, Ray and Martin Tracy, *Dr. Dobb's Journal*, September 1984, p. 110.
- Gedeon, D. "Complex Math in Pascal", *Byte Magazine*, July 1987, p.121.
- Ham, M. "Structured Programming", *Dr. Dobb's Journal*, July 1986.
- Ham, M. "Structured Programming", *Dr. Dobb's Journal*, October, 1986.
- Hammersley, J.M. and D.C. Hanscomb, *Monte Carlo Methods* (Methuen, London, 1964).
- Handbook of Mathematical Functions* ed. Milton Abramowitz and Irene Stegun (Dover Publications, Inc., New York, 1965).
- Harvard Softworks, P.O. Box 69, Springboro, Ohio 45066 Tel: (513) 748-0390.
- Ivancic, T.A. and G. Hunter, *J. Forth Appl. and Res.* 6 (1990) 15.
- Jones, Do-While *Dr. Dobb's Journal*, March 1987.
- Jump, D.N. *Programmer's guide to MS-DOS*, rev. ed. (Brady Books, New York, 1987).

- Kelly, M. and N. Spies, *FORTH: a Text and Reference* (Prentice-Hall, NJ, 1985)
- Knuth, D.E. *The Art of Computer Programming, v. 2: Seminumerical Algorithms*, 2nd ed. (Addison-Wesley Publishing Company, Reading, MA, 1981)
- Kohavi, Z. *Switching and Finite Automata Theory*, 2nd ed. (McGraw-Hill Publishing Co., New York, 1978).
- Kruse, R.L. *Data Structures and Program Design*, 2nd Ed. (Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987).
- Lafosse, R. *Assembly Language Primer for the IBM PC & XT* (Plume/Watson -New American Library, New York, 1984).
- Lefor, John A. and Karen Lund, "Reaching into expanded memory", *PC Tech Journal*, May 1987.
- Loesiger, R.G. *Threaded Interpretive Languages* (Byte Publications, Inc., Peterborough, NH, 1981).
- Mathews, J. and R.L. Walker, *Mathematical Methods of Physics*, 2nd ed. (W.A. Benjamin, Inc., New Jersey, 1970).
- Mignotte, M. *Mathematics for Computer Algebra* (Springer-Verlag, Berlin, 1992).
- Morgan, C. and M. Wolfe, *8086/8088 16-bit microprocessor primer* (Byte/McGraw-Hill, Peterborough, 1982)
- Noble, J.V. "Avoid Decisions", *Computers in Physics* 5, 4 (1991) 386.
- Noble, J.V. "Scientific Computation in FORTH", *Computers in Physics* 3 (1989) 31.
- Noble, J.V. J. *FORTH Appl. and Res.* 6 (1990) 47.
- Noble, J.V. J. *FORTH Appl. and Res.* 6 (1990) 131.
- Palmer, J.F. and S.P. Morse, *The 8087 Primer* (John Wiley and Sons, Inc., New York, 1984).
- Pan, V. *SIAM Review* 26 (1984) 393.
- Pavelle, R., M. Rothstein and J. Fitch, "Computer Algebra", *Scientific American* 245, #6 (Dec. 1981) 136.
- Pountain, Dick "Object-oriented FORTH", *Byte Magazine*, 8/86; *Object-oriented FORTH* (Academic Press, Inc., Orlando, 1987).
- Press, W.H., B.P. Flannery, S.A. Teukolsky and W.T. Vetterling, *Numerical Recipes* (Cambridge University Press, Cambridge, 1986)
- Ralston, A. *A First Course in Numerical Analysis* (McGraw-Hill Book Company, New York, 1965).
- Rawson, E. J. *FORTH Appl. and Res.* 3,4 (1986) 45-64.
- Scanlon, L.J. *IBM PC & XT assembly language: a guide for programmers* (Brady/Prentice-Hall, Bowie, Md., 1983).
- Sedgewick, R. *Algorithms* (Addison-Wesley Publishing Company, Reading, MA, 1983).
- Shaw, G. "Forth Shifts Gears, I", *Computer Language* (May 1988) p. 67
- Shaw, G. "Forth Shifts Gears, II", *Computer Language* (June 1988) p. 61
- Shaw, G. Proc. 9th Asilomar FORML Conference (JFAR 5 (1988) 347.)
- Straassen, V. *Numer. Math.* 13 (1969) 184.
- Williams, Al "DOS + 386 = 4 Gigabytes", *Dr. Dobb's Journal*, July 1990, p. 62.

Scientific FORTH: a modern language for scientific computing

FORTH has been called "...one of the best-kept secrets in the computing world." Combining the execution speed of a compiled language with the immediacy and convenience of an interpreted language, FORTH is nevertheless so simple its kernel can be compressed into a few kilobytes of machine code. Many scientists, engineers and programmers have recognized that FORTH provides the "most direct, revealing and flexible way for controlling computer hardware yet invented," applying FORTH to industrial control, robotics and laboratory instrumentation.

FORTH is the only completely extensible modern computer language. User-defined operators, data structures, commands, functions and subprograms act precisely like the core operators, data structures and commands — they are true extensions to FORTH. Moreover, the FORTH compiler is part of the language, available to the user. These features give FORTH enormous abstractive power and elegance of expression. Thus, a FORTH program to solve linear equations can look as simple as

```
: }}SOLVE ( adr[M] adr[y] -- ) SETUP TRIANGULARIZE BACKSOLVE ;
```

Scientific FORTH extends the FORTH kernel in the direction of scientific problem-solving. It is the first book to illustrate advanced FORTH programming techniques with non-trivial applications:

- high-speed real and complex floating point arithmetic
- numerical integration/Monte-Carlo methods
- linear equations and matrices
- functional representation of data (FFT, polynomials)
- function minimization
- differential equations
- roots of equations
- computer algebra

Julian V. Noble (B.S. Caltech '62; M.A. Princeton '63; Ph.D. Princeton '66) is Professor of Physics at the University of Virginia. His professional interests and publications include nuclear and particle physics, astrophysics, theoretical biology, and numerical methods. He has been programming digital computers since 1961, but became fascinated with personal computers since acquiring his first in 1979. He now uses FORTH almost exclusively for his scientific work.

ISBN 0-9632775-0-2



9 780963 277503



54995>