AN INTRODUCTION TO COMPUTATIONAL COMPLEXITY

R. F. Churchhouse
University College, Cardiff, United Kingdom

ABSTRACT

This single lecture, given as a seminar, introduces the basic ideas which underlie the subject of computational complexity and illustrates some of the more commonly used techniques with a number of examples. Although the subject is of relatively recent origin there is a considerable research literature, but so far relatively few books treating the subject in depth have appeared.

1. INTRODUCTION

In the early days of computers when all programming was in machine code programmers would often challenge one another to find the most efficient way of performing some computation. "Most efficient" usually meant "least CPU time", but sometimes it might mean "least number of words of storage". Two particular examples that I recall, from about 1955, were

(i) to find the fastest combination of instructions for deciding if a given (36-bit) word contained exactly one '1' in its 36 bits (I remember the result : 3 instructions taking 103μ secs),

(ii) to find how many of the bits in a word were equal to 1 ("sideways add") - on the first machine I programmed (Ferranti Mark I) this was a machine code instruction.

The second problem is interesting in that one solution is to break the 36-bit word into 3 twelve-bit words and look up the number of bits in each twelve-bit segment in a prepared table 4096 words long using the 12-bit numbers in index registers. This is quite a good way if the operation is to be performed many times; whether it is efficient depends upon how we regard the loss of 4096 words of store compared to the gain in speed.

From these early beginnings people began to take an interest in the number of operations required to solve problems where one or more parameters were involved. Some people learned to take such an interest the hard way; I can recall an "open-shop" programmer around 1957 who was quite calmly writing a program involving one parameter, n, where the running time was proportional to n! and n was quite likely to be in the region 20-40! Had he not been stopped his program would still be running.

It is therefore clearly desirable that if an algorithm is proposed for the solution of a problem then not only should we know that the algorithm will work (possibly under certain conditions) but also, if the algorithm

involves parameters, how long the algorithm will take to complete its work as a function of these parameters. Since the actual time depends on the particular machine it is customary to measure the algorithm by the number of operations of various kinds (multiplications, comparisons, additions, etc.) required. For some algorithms the number of such operations can be specified precisely, for others the number is data-dependent and one can only give "expected values" or "worst-case values".

2. NOTATION : Suppose that we have an algorithm which involves a parameter n; if the algorithm requires f(n) operations of some kind, we say that the algorithm has complexity O(f(n)) in that operation. We shall see that algorithms may have different orders of complexity in different operations. Usually the number of multiplications is taken to be the most significant in numerical algorithms but this need not necessarily be so and some combination of arithmetic operations and/or storage may be more important in some cases.

3. PRODUCT OF TWO COMPLEX NUMBERS

As a simple example consider the problem of multiplying two complex numbers x=(a+ib) and y=(c+id). On a machine which possesses only 'real' multiplication the obvious way of finding the product is to note that

$$xy = (a+ib)(c+id) = (ac-bd)+i(bc+ad)$$

The expression on the right involves 4 multiplications and one addition and one subtraction - there appear to be 2 additions, but this is due solely to the notation for complex numbers; the + between the brackets is symbolic.

Consider however the identities:

$$(a+b)c + (d-c)a = ad+bc = \mathrm{Im}(xy)$$

$$(a+b)c - (d+c)b = ac-bd = \mathrm{Re}(xy)$$

which reveal that the real and imaginary parts of the product can be found with 3 multiplications - one of the products being used twice. We have paid a price of course since we have increased the number of additions/subtractions from 2 to 5 but if minimising the number of multiplications is our criterion of efficiency this second method is superior. The algorithm above can be neatly described by:

$$\begin{aligned} & f_1 = a+b \\ & f_2 = f_1*c \\ & f_3 = d-c \\ & f_4 = f_3*a \\ I_m(x,y) = \quad & f_5 = f_2+f_4 \\ & f_6 = d+c \\ & f_6 = f_6*b \\ R_e(x,y) \quad & f_8 = f_2-f_7 \end{aligned}$$

- the number of operations of various kinds is now easy to see.

4. PRODUCT OF 2 n-BIT NUMBERS

Suppose we wish to multiply two n-bit numbers x,y; we shall suppose for simplicity that n is a power of 2. The "obvious" way requires $O(n^2)$ bit operations. Let us break both of x,y into two halves

$$x = \boxed{a \mid b}$$

$$y = \boxed{c \mid d}$$

and treat each half as an $\frac{n}{2}$-bit number. Then the "obvious" multiplication of xy is

$$xy = (a2^{n/2}+b)(c2^{n/2}+d)$$
$$= ac2^{n}+(ad+bc)2^{n/2}+bd$$

- involving four multiplications of $^{n}/_{2}$ bit numbers plus some additions and shifts.

An an alternative method consider:

$$u = (a+b)*(c+d)$$
$$v = a*c$$
$$w = b*d$$
$$z = v*2^{n}+(u-v-w)*2^{n/2}+w = xy$$

- this scheme requires 3 multiplications, 2 shifts and some additions. If multiplications are expensive compared to shifts and additions then this algorithm is superior to the obvious one and furthermore if n is a power of 2 it can be applied recursively so that ultimately the number of multiplications is not $O(n^2)$ but $O(n^{\log_2 3}) \doteq (n^{1.59})$.

This 'fast' integer-multiplication algorithm can be applied to integers in bases other than binary, e.g. in decimal.

Example Multiplication of two 4-digit numbers on a machine which can only multiply two-digit numbers.

3741 x 2849

a = 37, b = 41; c = 28, d = 49

(a+b) = 78, (c+d) = 77

u = 78*77 = 6006

v = 37*28 = 1036

w = 41*49 = 2009

$z = 1036 \times 10^4 + (6006-1036-2009) \times 10^2 + 2009$

= 10360000

296100

2009

10,658,109

The technique adopted here is by no means optimal. Using a

generalisation of this technique followed by an ingenious application of the Fast Fourier Transform Schönhage and Strassen produced an algorithm to multiply two n-bit numbers in O(n logn log logn steps) (see [4]).

This last example is a particular case of the "divide and conquer" approach which is frequently used in problem solving viz: break the problem into smaller parts, solve each of the smaller problems and then combine the solutions to give the solution to the original problem. If the technique is applied recursively it may be possible to obtain a functional equation the solution of which gives a formula for the complexity of the algorithm.

Thus, suppose we have an algorithm which we propose to use to solve a certain problem which involves a parameter n. Let T(n) denote the (unknown) number of operations required by the algorithm to complete its work. If we break the problem into two so that n is replaced by $\frac{1}{2}n+\frac{1}{2}n$ we will have two problems which will require $T(\frac{1}{2}n)$ operations to complete and we may be able to relate T(n) to $T(\frac{1}{2}n)$ by a formula of the type

$$T(n) = F(T(\tfrac{1}{2})) \qquad (4.1)$$

where F(.) is some known function, in which case (4.1) provides a functional equation for T(n) and if we can solve this we will have a formula for T(n).

The method can be simply illustrated by considering an algorithm which sorts n items into order. Suppose the algorithm requires S(n) operations to complete its task; divide the n items into two sets each containing $\frac{1}{2}n$ items; sort each set, this takes $2S(\frac{1}{2}n)$ operations. We must now merge the two sorted sets; merging for $\frac{1}{2}n$ sorted items with another $\frac{1}{2}n$ sorted items takes about kn operations, where k is some constant. It follows that, approximately:

$$S(n) = 2S(\tfrac{1}{2}n) + kn \qquad (4.2)$$

and the exact solution of (2) is

$$S(n) = kn \log_2 n \qquad (4.3)$$

(provided $k \neq 0$)

In reality the situation is a little more complicated e.g. there may also be a constant term on the r.h.s. of (4.2) but the analysis indicates quite clearly that sorting based on recursive bisection ought to have complexity $O(n\log_2 n)$ - and this is, of course, the case for many of the better sorting methods. At the extreme ends of the spectrum of sorting methods, so to speak, we have "bucket-sort" which is suitable only if n is fairly small and "bubble sort", perhaps the simplest sort of all, which have complexity O(n) and $O(n^2)$ respectively. Bucket sort corresponds to the case k=0 in (4.2) when the solution degenerates to S(n) = An.

5. POLYNOMIAL EVALUATION

It has been known for a long time that if we wish to evaluate a polynomial of the n-th degree

$$p(x) = a_0+a_1x+a_2x^2+\ldots+a_nx^n \tag{5.1}$$

then Horner's method in which we perform the computation as

$$p(x)=(((a_nx+a_{n-1})x+a_{n-2})x+a_{n-3})x+\ldots.) \tag{5.2}$$

completes the task in n multiplications and n additions. What is not obvious, but can be proved is that Horner's rule is optimal (see [1], 438-439) on a uniprocessing machine.

6. MATRIX OPERATIONS

Direct methods for solving systems of n linear equations or for inverting nxn matrices can be analysed very easily and the number of operations of various kinds counted precisely. Some results are summarised below.

Method	x	÷	±
Gaussian elimination	$\frac{1}{3}n^3+n^2-\frac{1}{3}n$	n	$\frac{1}{3}n^3+\frac{1}{2}n^2-\frac{5}{6}n$
Jordan	$\frac{1}{2}n^3+n^2-\frac{1}{2}n$	n	$\frac{1}{2}n^3-\frac{1}{2}n$
Inverse(Gauss)	n^3	n	n^3-2n^2+n
Inverse(Jordan)			

We shall now indicate how algorithms of lower order can be achieved.

6.1 Matrix multiplication

Consider the problem of multiplying two 2x2 matrices A,B

$$C = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} \tag{6.1}$$

the elements of the resulting matrix C are given by four relations such as

$$c_{11} = a_{11}*b_{11} + a_{12}*b_{21} \tag{6.2}$$

-clearly we require 8 multiplications and 4 additions to find C by this method.

Generalising, we see that if we multiply two nxn matrices by this method we shall require n^3 multiplications and (n^3-n^2) additions.

It seems almost aniomatic that the number of multiplications needed when we multiply two nxn matrices must be $O(n^3)$ but this is not so. In 1969 Strassen [2] published a method which requires $O(n^{2.81})$ arithmetic operations; this is based upon the following identities, using the

notation of (6.2).

Let

$$\begin{aligned} m_1 &= (a_{12}-a_{22})(b_{21}+b_{22}) \\ m_2 &= (a_{11}+a_{22})(b_{11}+b_{22}) \\ m_3 &= (a_{11}-a_{21})(b_{11}+b_{12}) \\ m_4 &= (a_{11}+a_{12})b_{22} \\ m_5 &= a_{11}(b_{12}-b_{22}) \\ m_6 &= a_{22}(b_{21}-b_{11}) \\ m_7 &= (a_{21}+a_{22})b_{11} \end{aligned} \tag{6.3}$$

then

$$\begin{aligned} c_{11} &= m_1+m_2-m_4+m_6 \\ c_{12} &= m_4+m_5 \\ c_{21} &= m_6+m_7 \\ c_{22} &= m_2-m_3+m_5-m_7 \end{aligned} \tag{6.4}$$

This method involves 7 multiplications and 18 additions/subtractions and it follows that we can multiply two 2x2 matrices with 7 multiplications and 18 additions.

Suppose that now A and B are two nxn matrices (where we take n as a power of 2 for simplicity), then $a_{11}, a_{12}, b_{21}, b_{22}$ are all $\frac{n}{2}x\frac{n}{2}$ matrices and if we look at (6.3) and (6.4) we see that the 18 additions in reality involve us in

$$18(\tfrac{n}{2})^2 \tag{6.5}$$

additions of actual elements. If T(n) is the total number of arithmetic operations required to multiply two nxn matrices using (6.3) and (6.4) we see that T(n) satisfies

$$T(n) = 7T(\tfrac{n}{2}) + 18(\tfrac{n}{2})^2 \tag{6.6}$$

from which we deduce that

$$T(n) = O(n^{\log_2 7}) \doteq O(n^{2.81}) \tag{6.7}$$

i.e. we can multiply two nxn matrices in $O(n^{2.81})$ arithmetic operations.

A refinement of Strassen's method, due to Winograd, involves 7 multiplications and 15 additions (see [3], 181-196).

Based upon these ideas it can be proved both that a system of n linear equations can be solved and that any nxn matrix can be inverted in $O(n^{2.81})$

arithmetic operations.

7. PARALLELISM

Everything that I have mentioned so far has assumed implicity that the algorithms have been implemented on a serial machine. If however we have available k parallel processors how should the algorithms be redesigned (a) if the k processors are identical, (b) if the processors are not necessarily identical, (c) if k can become arbitrarily large? Some research has been done on such questions but there is plenty of scope for more; one interesting result, quoted by Dr. Zacharov in his lectures, will indicate the flavour : Maruyama's proof that polynomials of degree n might be evaluated in $\log n + \sqrt{2 \log n} + O(1)$ time steps if an unlimited number of processors are available (see [5]).

References

[1] Aho,A., Hopcroft,J.E., Ullman,J.D. "The Design and Analysis of Computer Algorithms," Addison-Wesley, 1974.

[2] Strassen,V. "Gaussian elimination is not optimal," Numerische Mathematik 13 (1969), 354-356.

[3] Traub, J.F., (ed), "Complexity of Parallel and Numerical Algorithms," Academic Press, New York (1973).

[4] Schöhage,A. and Strassen,V. "Schnelle Multiplikation grosser Zahlen," Computing, 7 (1971), 281-292.

[5] Maruyama, K. "On the parallel evaluation of polynomials" IEEE Trans Computers, Vol. C-22 No. 1 (Jan. 1973) pp.2-5.

INTERNATIONAL ATOMIC ENERGY AGENCY
UNITED NATIONS EDUCATIONAL, SCIENTIFIC AND CULTURAL ORGANIZATION

INTERNATIONAL CENTRE FOR THEORETICAL PHYSICS
34100 TRIESTE (ITALY) - P.O.B. 586 - MIRAMARE - STRADA COSTIERA 11 - TELEPHONE: 2240-1
CABLE: CENTRATOM - TELEX 460392 - I

SECOND SCHOOL ON ADVANCED TECHNIQUES
IN COMPUTATIONAL PHYSICS
(18 January - 12 February 1988)

SMR.282/25

PROGRAM LIBRARIES

J. Du Croz
Numerical Algorithms Group Ltd., Oxford, U.K.

PROGRAM LIBRARIES

Jeremy Du Croz
Numerical Algorithms Group Ltd
256 Banbury Road
Oxford OX2 7DE
England

1. Survey of Libraries
2. Methods of Library Development

PROGRAM LIBRARIES 1

Survey of Subroutine Libraries

Commercial Libraries:
- NAG
- IMSL
- PORT
- BCSLIB
- MathAdvantage
- . . .

Manufacturers' Libraries (machine-specific)
- IBM (ESSL)
- Cray
- . . .

Libraries from large laboratories
- SLATEC
- Harwell
- CERN
- . . .

Public Domain Software
- Linpack
- Eispack
- ACM algorithms
- . . .

Comments on Libraries

The word 'library' may create the wrong impression.

Most 'subroutine libraries' are not like libraries of books; they are more like encyclopaedias.

They usually provide just one selected subroutine for each distinct type of problem. (A choice of subroutines may be provided if no one subroutine can be relied on to solve all problems of that type with the desired reliability and efficiency.)

All good libraries offer the following benefits:

* accurate, robust and reliable numerical methods

* tried and test software

* careful and consistent documentation

Even the commercial libraries are comparatively cheap, and can save a great deal of programming effort.
Do not re-invent the wheel!

Commercial Libraries

NAG and IMSL

the most widely distributed and most comprehensive libraries (see below).

PORT

the in-house library of Bell Labs.

BCSLIB

the in-house library of Boeing Computer Services (>150 routines); uses VectorPak for efficency on Cray-like machines.

MathAdvantage

routines for linear algebra and signal-processing, aimed at supercomputers. From Quantitative Technology Corporation.

NAG and IMSL

Common Features

* Comprehensive coverage of the main areas of numerical mathematics and statistics

* Comprehensive and consistent documentation (7 volumes)

* Library is normally provided as compiled and tested object code (with source-text available for reference)

* Available on a wide range of different computers

* Available at >>1,000 user sites all over the world

* Provide a library service, in return (usually) for an annual licence fee, including:
  - notification and correction of errors
  - periodic updates (new Marks or Editions)
  - consultation
  - newsletters

* Well-established organizations (since early 1970's)

* Sell or distribute other related software

NAG and IMSL

Common Features (continued)

* Library divided into chapters, each covering one area of numerical mathematics or statistics

* Each chapter has a corresponding Chapter Introduction document, giving background information and advice on choosing a routine

* Each routine is documented by a routine document, which includes a complete example program.

NAG

Size of the Library

Mark 12 of the NAG Fortran Library contains 688 documented routines.

Mark 13 (to be released in the second half of 1988) will contain about 750.

Sublibraries

NAG Workstation Library (172 routines)

NAG PC50 Library (50 routines)

Supplements

NAG Graphical Supplement (56 routines for graph-plotting)

NAG On-line Information Supplement (an interactive help system giving advice and machine-based documentation on the NAG Fortran Library)

NAG (continued)

Libraries in Other Languages

NAG Algol 68 Library (350 routines)

NAG Pascal Library (81 routines)

NAG Ada Library (to be released shortly)

Specialized topic libraries

NAG/SERC Finite Element Library: a collection of subroutines and template programs for solving finite element problems

SLICE (Subroutine Library in Control Engineering): a library of subroutines for control system design (developed at Kingston Polytechnic with support from SERC)

DASL (Data Approximation Subroutine Library): a library of subroutines for fitting or interpolating curves or surfaces (developed at the National Physical Laboratory)

NAG (continued)

Statistical Packages

GLIM, GENSTAT, TSA, MLP

Distribution Services

Toolpack/1, Eispack, Linpack, Minpack

For further information contact:

Numerical Algorithms Group Ltd
256 Banbury Road
Oxford OX2 7DE
England

Tel: International +44 865 511245
Telex: 83354 NAG UK G

IMSL

On 1 April 1987 IMSL released a complete revision of their library (Edition 10), which is incompatible with earlier editions (but an interface is provided for about 80% of the routines in edition 9).

The library is now in fact divided into three libraries:

MATH Library (426 routines)

SFUN Library (172 routines, for special functions)

STAT Library (351 routines)

(There is a small amount of overlap between the three libraries.)

IMSL also sell PROTRAN, a problem-solving environment which provides an alternative interface to IMSL library routines. It has the following modules:

MATH/PROTRAN
STAT/PROTRAN
LP/PROTRAN
PDE/PROTRAN

IMSL (continued)

IMSL provide a distribution service for the following public domain software: ACM algorithms, B-Spline, Eispack, EDA, Elefunt, Graphpak, Linpack, LLSQ, Minpack, Quadpack, Rosepack, Toeplitz)

For further information contact:

IMSL
2500 ParkWest Tower One
2500 CityWest Boulevard
Houston
Texas 77042-3020
USA

Tel: (713) 782-6060
Telex: 791923 IMSL INC HOU

Manufacturers' Libraries

Provided now mainly by manufacturers of modern high-performance computers.

The libraries tend to be small collections of routines for basic numerical computations (e.g. Linear Algebra, FFT's), optimized for a particular architecture. E.g.

SCILIB for Cray machines
ESSL for IBM 3090 VF
Paralin and Paraeig for Alliant FX/8

The code is not intended to be portable; often it is written in machine-language.

The user-interface may be non-portable too (i.e. you won't find a routine with the same name and argument-list available anywhere else). However some routines do conform to a widely accepted 'standard' specification (e.g. BLAS or Linpack) and this is a welcome development: it allows users' programs which call these routines to be moved from machine to machine and take advantage of the optimized implementations.

Libraries from the Laboratories

Harwell

A collection of over 300 subroutines, mostly written by people at Harwell (or visiting there). The contents cover most of the standard areas of numerical mathematics, but to some extent reflect the special interests and requirements of people at Harwell.

The library is especially strong in subroutines for sparse problems (real and complex linear equations, eigenvalue problems, non-linear equations, linear and quadratic programming, and non-linear optimization).

Available for a handling charge to academic sites (higher charge for commercial sites). Contact: S.Marlow, Building 8.9, AERE Harwell, Didcot, Oxfordshire OX11 0RA, England.

Libraries from the Laboratories

CERN

A collection of several hundred subroutines, covering the standard areas of numerical mathematics, together with a number that are of special interest to theoretical physicists.

Source-text is available (usually for a small charge) from: Program Library, Division DD, CERN, CH-1211 Geneve 23, Switzerland.

Libraries from the Laboratories

SLATEC

The SLATEC Common Mathematical Subroutine Library is an experiment in resource sharing by the computing departments of U.S. Department of Energy laboratories (Air Force Weapons Lab.; Lawrence Livermore; Los Alamos; National Bureau of Standards; Oak Ridge; Sandia, Albuquerque; Sandia, Livermore).

Based largely on public domain software (see below), but adapted to meet consistent standards of software and documentation. All documentation is machine-readable. Fairly comprehensive covereage of the standard areas of numerical mathematics.

Available to other sites from the National Energy Software Center (see below for address).

W.H.Vandevender and K.H.Haskell,
The SLATEC Mathematical Subroutine Library,
Signum Newsletter, 17, no. 2, 16-21, 1982.

Public Domain Software

* quality of software is often high, but can be variable

* software may need to be modified to run correctly on a particular machine (e.g. machine-dependent constants, precision conversion)

* test programs vary in their thoroughness

* documentation may be published in a book or technical report, or may be embedded in the source-text

* no support (except by goodwill of authors)

* free, except that there is usually a handling charge for distribution

Available from: IMSL or NAG distribution services; authors or their institutions; NESC (National Energy Software Center, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, Illinois 60439, USA). See below for details.

Individual routines may be obtained from NETLIB (see below)

Public Domain Software (continued)

Examples:

Eispack:

routines for matrix eigenvalue problems

Matrix Eigensystem Routines - EISPACK Guide,
B.T.Smith, J.M.Boyle, J.J.Dongarra, B.S.Garbow,
Y.Ikebe, V.C.Klema and C.B.Moler, Lecture Notes in
Computer Science 6, Springer Verlag, 2nd. ed, 1976.

Matrix Eigensystem Routines - EISPACK Guide Extension,
B.S.Garbow, J.M.Boyle, J.J.Dongarra and C.B.Moler,
Lecture Notes in Computer Science 51, Springer Verlag, 1977.

available from IMSL, NAG or NESC.

Public Domain Software (continued)

Linpack:

routines for systems of linear equations and linear least squares problems

LINPACK User's Guide,
J.J.Dongarra, C.B.Moler, J.R.Bunch and G.W.Stewart,
SIAM, 1979.

available from IMSL, NAG or NESC.

Minpack:

routines for non-linear equations and non-linear least squares problems

User Guide for MINPACK-1,
J.J.More, B.S.Garbow and K.E.Hillstrom, Argonne National
Laboratory technical report ANL-80-74, 1980.

available from IMSL, NAG or NESC.

Public Domain Software (continued)

Quadpack:

routines for one-dimensional integration

QUADPACK - A Subroutine Package for Automatic Integration,
R.Piessens, E.de Doncker-Kapenga, C.W.Uberhuber and
D.K.Kahaner, Springer Verlag, 1983.

available from IMSL.

PPPack:

routines for interpolation and approximation using splines

A Practical Guide to Splines,
C.de Boor, Springer Verlag, 1978.

available from IMSL

Public Domain Software (continued)

Fishpak:

routines for solving elliptic P.D.E.'s,
by P.M.Swarztrauber and R.A.Sweet (documentation
in source-text)

available from: NCAR/SCD, P.O.Box 3000, Boulder,
CO 80307, USA.

FFTpack:

routines for Fast Fourier Transforms,
by P.M.Swarztrauber (documentation in source-text)

available from: NCAR/SCD (see above)

Public Domain Software (continued)

Y12M:

routines for large sparse unsymmetric systems of equations,

Y12M,
Z.Zlatev, J.Wasniewski and K.Schaumburg, Lecture Notes in Computer Science 121, Springer Verlag, 1979.

available from: J.Wasniewski, UNI-C, DTH, Bygning 305, DK-2800 Lyngby, Denmark.

ACM Algorithms:

as published in ACM Transactions on Mathematical Software

available from IMSL.

NETLIB

Distribution of mathematical software via electronic mail

Send requests to

netlib@anl-mcs (on arpanet/csnet)

or

research!netlib (from unix)

Advantages:

* rapid response (if you have good connections to email network)

* up-to-date software

* free (but someone will be paying for the phone-calls)

NETLIB (continued)

Requests may take the following forms:

To obtain index to netlib and general advice:

send index

To obtain the index from an individual library (e.g. Linpack):

send index from linpack

To obtain one routine from Linpack, with all the routines that it calls:

send ssifa from linpack

To obtain one routine on its own:

send only ssifa from linpack

NETLIB (continued)

Points to beware of:

* not to be used to obtain complete libraries (only for selected routines or programs)

* no support (bug reports forwarded to authors)

* no claim for quality of software

Caveat Receptor!

Other information available:

Addresses of numerical analysts
Details of high performance computers
Linpack benchmark and programs
Errata to books on numerical analysis

NETLIB (continued)

Quick summary of contents

alliant - set of programs collected from Alliant users
apollo - set of programs collected from Apollo users
benchmark - various benchmark programs and a summary of timings
bihar - Bjorstad's biharmonic solver
bmp - Brent's multiple precision package
conformal - Schwarz-Christoffel codes by Trefethen; Bjorstad+Grosse
core - machine constants, blas
domino - communication and scheduling of multiple tasks; Univ. Maryland
eispack - matrix eigenvalues and vectors
elefunt - Cody and Waite's tests for elementary functions
errata - corrections to numerical books
fishpack - separable elliptic PDEs; Swarztrauber and Sweet
fitpack - Cline's splines under tension
fftpack - Swarztrauber's Fourier transforms
fmm - software from the book by Forsythe, Malcolm, and Moler

NETLIB (continued)

fn - Fullerton's special functions
go - "golden oldies" gaussq, zeroin, lowess, ...
harwell - MA28 sparse linear system
hompack - nonlinear equations by homotopy method
itpack - iterative linear system solution by Young and Kincaid
lanczos - Cullum and Willoughby's Lanczos programs
laso - Scott's Lanczos program for eigenvalues of sparse matrices
linpack - gaussian elimination, QR, SVD by Dongarra, Bunch, Moler, Stewart
lp - linear programming
machines - short descriptions of various computers
microscope - Alfeld and Harris' system for discontinuity checking
minpack - nonlinear equations and least squares by More, Garbow, Hillstrom
misc - everything else
ode - ordinary differential equations
odepack - ordinary differential equations from Hindmarsh
paranoia - Kahan's floating point test
pchip - hermite cubics Fritsch+Carlson
pltmg - Bank's multigrid code
port - the public subset of PORT library

NETLIB (continued)

pppack - subroutines from de Boor's Practical Guide to Splines

quadpack - univariate quadrature by Piessens, de Donker, Kahaner

siam - typesetting macros for SIAM journal format

slatec - machine constants and error handling package from the Slatec library

specfun - transportable special functions

toeplitz - linear systems in Toeplitz or circulant form by Garbow

toms - Collected Algorithms of the ACM

PROGRAM LIBRARIES 2

Methods of Library Development

The purpose of this lecture is to give some understanding of what is involved 'behind the scenes' in developing and maintaining a large subroutine library.

The ideas and methods should be of interest to anyone who is involved - either as an individual or as a member of a group - in developing scientific software of high quality which is intended to be used by many people on different types of computers.

Recommended reading:

Problems and Methodologies in Mathematical Software Production,
ed. P.C.Messina and A.Murli,
Lecture Notes in Computer Science 142,
Springer Verlag, 1982.

The Size of the Task

Mark 12 of the NAG Fortran Library contains:

688 documented user-callable routines
1420 routines altogether, including auxiliaries
(200,000 lines of source-text)

544 example programs (50,000 lines)
468 certification programs (100,000 lines)

Development of this software has involved the collaboration of more than 100 contributors, interacting with the full-time staff of NAG.
This implies a need for central co-ordination, and agreed standards to work to.

Preparation, revision and maintenance of such a large body of software requires extensive use of Software Tools.

The Nature of the Task

* To select numerical algorithms which are
  - useful
  - robust
  - numerically stable
  - accurate
  - efficient

* To design user-friendly Fortran 77 subroutines

* To develop software which can easily be transported to many different computers without losing the desired accuracy and efficiency

* To develop suitable test software for certifying the subroutines on each computer

* To prepare suitable documentation and means for disseminating it

Selection of Numerical Algorithms

Usefulness

Depends on the application.

Robustness

The algorithm must either return acceptable results (for the intended class of problems), or must fail gracefully with a clear indication of the cause of failure. Beware of:

singular or ill-conditioned problems
problems with no solution
overflow or underflow

For some types of problems (e.g. numerical quadrature which relies on function values only), no algorithm can be 100% reliable: warn the users!

Numerical Stability

The algorithm must not be excessively sensitive to small changes or uncertainties in the data, or to rounding errors caused by computing in finite precision.

Selection of Numerical Algorithms (continued)

Accuracy

How much accuracy is required?
Can you state how much accuracy will be achieved?
Many algorithms can achieve an accuracy which is as good as one can reasonably expect, given the precision of the computer and the conditioning of the problem.
Some algorithms can return quite cheaply

an error-bound,
an error-estimate, or
a measure of the conditioning.

Occasionally special action must be taken to achieve greater accuracy (e.g. iterative refinement, working in higher precision).

Often only modest accuracy is required, and a cheaper algorithm may be acceptable, provided that the actual accuracy can be stated (e.g. approximations of special functions).

Selection of Numerical Algorithms (continued)

Efficiency

How important is efficiency?
Beware of achieving efficiency at the expense of accuracy, numerical stability or robustness!

Don't fuss about details of the algorithm: look at the overall cost (e.g. number of floating-point operations, number of iterations, number of function evaluations).

If you measure the speed of an algorithm, or compare the speeds of competing algorithms, remember that the results may be heavily influenced by the computer, the compiler, the compilation options.

Vector-processors have forced us to pay closer attention to efficiency: details of the code can have a dramatic impact. Even so, concentrate on the most heavily used parts of the algorithm.

External Design of Subroutines

An art!

Argument Lists

Choose a consistent scheme for ordering argument lists.

Much good numerical software uses the following:

1. Input arguments
2. Input-output arguments
3. Output arguments
4. Workspace arguments
5. Diagnostic argument (e.g. INFO or IERR)

External Design of Subroutines (continued)

NAG prefers a more refined scheme:

1. Option arguments (to define which task or tasks are to be performed)
2. Arguments which define the primary data for the problem (some may be overwritten by results)
3. Output arguments which return the rest of the principal results
4. Arguments which control the computation (e.g. required accuracy, maximum number of iterations)
5. Output arguments which return information about the computation (e.g. actual number of iterations)
6. Workspace arguments
7. Diagnostic argument

Example: for computing eigenvalues and optionally eigenvectors of a matrix A:

(WANTV, N, A, LDA, RLAMDA, V, LDV, NIT, WORK, IFAIL)

External Design of Subroutines (continued)

How to shorten argument lists

Users - reasonably - complain about long argument lists.
We would like to be able to use:

optional arguments
default values
dynamic allocation of work arrays (inside the routines)

but Fortran 77 does not provide these features
(Fortran 8x will, we hope!)

Various devices have been used in large libraries to provide crudely equivalent features in Fortran 77: e.g.

use COMMON for 'optional arguments' with default values defined by BLOCK DATA within the library (Harwell)
(risk of user error, portability problems with BLOCK DATA)

use COMMON for workspace (PORT, IMSL) (requires elaborate machinery within the infrastructure of the library; not standard Fortran 77, though it nearly always works)

External Design of Subroutines (continued)

combine 'less-interesting' arguments into arrays (IMSL, others)

option-setting routines (NAG) (again require elaborate machinery)

more than one level of interface, i.e. an 'easy-to-use' routine with a short argument list (for 'naive' users), calling a 'comprehensive' routine with a long argument list (for 'expert' users) (never satisfies everybody!)

Array Arguments

For 2-dimensional (and higher-dimensional) arrays, the leading dimension(s), as declared in the calling (sub)program, should be passed as separate arguments, e.g.

```
SUBROUTINE XXXX ( N, A, LDA, . . . )
REAL A(LDA,N)
```

NAG Option Setting Routines

Example: E04DGF

finds an unconstrained minimum of a function of N variables.

Calling sequence:

```
CALL E04DGF (N, OBJFUN, X, ITER, OBJF, OBJGRD,
             IWORK, WORK, IUSER, USER, IFAIL )
```

Options:

Estimated Optimal Function Value
Function Precision
Iteration Limit
Linesearch Tolerance
Maximum Step Length
Optimality Tolerance
Print Level
Start Objective Check at Variable
Stop Objective Check at Variable
Verify Level

NAG Option Setting Routines (continued)

To set options:

either

```
CALL E04DKF ('Verify Gradients')
CALL E04DKF ('Print Level = 1')
```

or

```
CALL E04DJF (7, INFORM)
```

with an options file on unit 7:

```
Begin                    * Example options file
  Verify Level
  Print Level = 1
End
```

Transportable Numerical Software

We would call software 'portable' if it could be moved from one computer to another and continue to perform correctly to the desired standards of accuracy and efficiency. Usually this cannot be achieved.

We call software 'transportable' if the changes required to satisfy these goals are predictable and:

either few in number and localised
(e.g. a routine which returns the machine-precision)

or systematic and capable of being made automatically by software tools (e.g. conversion from single to double precision)

To achieve transportability we have to consider:

the programming language
the numerical computing environment
(arithmetic, elementary functions)
performance

Transportable Numerical Software (continued)

Programming Language

Use standard Fortran 77 with no extensions!

Except perhaps COMPLEX*16 and associated intrinsic functions (but it doesn't work on all machines where it is needed).

Even standard Fortran 77 does not guarantee portability: notorious 'dodgy' features are:

- input/output
- length of character variables
- ordering of variables in COMMON blocks

How can you check for non-standard Fortran 77?
Many compilers have an option to do this, but they may not be totally reliable.
Use a reliable independent tool (e.g. PFORT 77 in Toolpack: this also checks for dodgy standard features (as above)).

Transportable Numerical Software (continued)

Numerical Computing Environment

Use adaptable algorithms, i.e. algorithms which adapt to the numerical characteristics of the computer on which they are being executed. Relevant characteristics are:

- the machine precision
- the underflow threshold **
- the overflow threshold **

** Beware of a lack of symmetry between these thresholds, and of unexpected failures in the elementary functions.
If U is the underflow threshold, then we would like to be able to compute:

```
-U
1/U
SQRT(U)
LOG(U)
EXP(LOG(U))
```

and so on

but this may cause underflow, overflow or failure in the elementary functions.

NAG uses a 'safe range' value slightly larger than the underflow threshold, which avoids these difficulties. However determining the correct value is not easy.

The simplest way to make the values of machine characteristics available is via function subprograms, containing a simple assignment statement which can easily be changed (NAG, PORT, IMSL).

Beware of anomalous behaviour of floating-point arithmetic (e.g. on CDC machines) and of loss of accuracy in the elemantary functions. You can test the arithmetic using the NAG package FPV or the public domain program Paranoia (by Kahan); you can test the elementary functions using the public domain package ELEFUNT (by Cody and Waite).

Transportable Numerical Software (continued)

Performance

To achieve high performance across a range of different vector-processors as well as on scalar machines, NAG's strategy is to make as much use as possible of a small set of kernel routines for matrix-vector operations, for which well-tuned implementations can be developed for different machines.

NAG has collaborated in specifying a 'standard' set of such kernel routines:

the Level 2 BLAS

(Basic Linear Algebra Subprograms).

Transportable Numerical Software (continued)

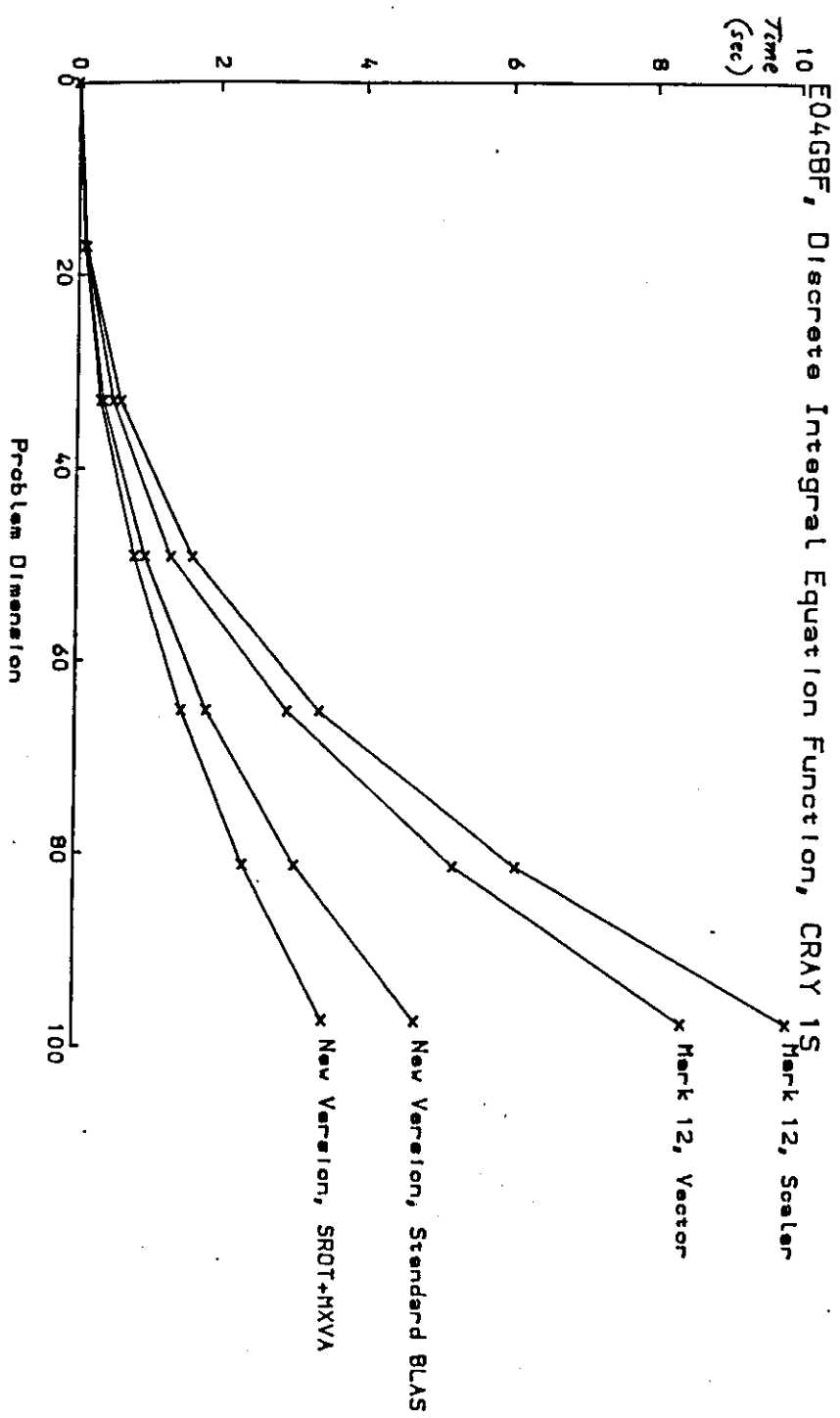
NAG Library routines call Level 2 BLAS routines by their BLAS names:

```
      DO 60 K = 1, IR
C        Solution of LY = B
         CALL STRSV('L','N','N',N,A,IA,B(1,K),1
C        Solution of UX = Y
         CALL STRSV('U','N','U',N,A,IA,B(1,K),1
   60 CONTINUE
```

Hence NAG routines can be linked to optimized machine-specific implementations of the BLAS, if available.

Speed in megaflops:

F03AEF (Cholesky Decomposition)					
CRAY-1 (University of London Computer Centre)					
N	AXP	AXP2	AXP3	AXP4 (M12)	MXVA
50	7.47	9.11	9.66	10.22	11.37
100	15.55	20.10	22.70	24.49	33.46
150	21.08	27.81	32.34	35.32	53.94
200	24.94	32.97	39.01	42.87	69.53
250	27.67	36.54	43.71	48.25	80.79
300	29.85	39.16	47.11	52.20	88.63
350	31.46	41.15	49.68	55.15	94.78
400	32.76	42.74	51.71	57.53	99.37
450	33.81	44.10	53.38	59.42	102.71
500	34.66	45.01	54.65	60.87	105.37



Library Infrastructure

Most large libraries include a certain amount of infrastructure to implement basic design features e.g.

- access to machine characteristics
- provision of workspace
- option-setting
- optimised basic modules
- communication with external files
- error-handling

Most of these have been described in previous sections.

Library Infrastructure (continued)

Error-handling

The simplest design is for routines to indicate their success or failure by the value of an integer argument (e.g. INFO). Usually

INFO = 0 indicates successful termination

INFO ≠ 0 indicates failure (different values of INFO indicate different reasons)

Some libraries have a more elaborate mechanism

to give users control over the action to be taken in the event of failure

to give users more information about the failure

Library Infrastructure (continued)

The NAG error-handling mechanism: each routine has an input-output argument IFAIL. Its value on entry specifies the action to be taken:

IFAIL = 1

soft failure, no error messages

IFAIL = 0

hard failure, with error messages

IFAIL = -1

soft failure, with error messages

On exit, IFAIL = 0 indicates succesful termination; other values indicate an error.

Library Infrastructure (continued)

Examples of error messages

```
** On entry, M.lt.1: M =            -255
** ABNORMAL EXIT from NAG Library routine C06FPF: IFAIL =     1
** NAG hard failure - execution terminated
```

```
** Bad integrand behaviour occurs around the subinterval
   ( 7.3193359E-01 , 7.3242188E-01 )
** ABNORMAL EXIT from NAG Library routine D01AKF: IFAIL =     3
** NAG soft failure - control returned
```

Most users seem to prefer a fairly simple, but informative error-handling mechanism.

Test Programs

Different stages of testing:

* Testing an algorithm for correctness and evaluating its performance

* Testing the correctness of the software as a realization of the algorithm ('certification')

We concentrate here on the latter.

Certification programs should be designed to detect:

corruption of source-text
errors in transformations performed by software tools
errors in compilation
errors in manual amendments to the code
failure of the algorithm to adapt satisfactorily to different machine-characteristics

Test Programs (continued)

Design guidelines:

exercise all the code (use a dynamic analysis tool to check for this)

test for 'edge-effects' (i.e. when some elements in the data take extreme values or satisfy some exceptional condition); in particular, test all error-exits, and special cases like n = 0 or n = 1, or f(x) = 0 for all x.

make the program check the results automatically

choose test problems which are no larger nor more complex than necessary

Note: there are extra difficulties in trying to achieve these aims on several different computers. E.g. a problem that is too ill-conditioned to be solved on one computer may be soluble on another: therefore use a sequence of problems of increasing ill-condition.

Test Programs (continued)

Try to take advantage of compilers which provide run-time checks for:

array-subscripts out of bounds
use of unassigned variables

To write suitable certification programs is fairly easy in some areas of numerical computing (e.g. linear algebra, FFT's); it is much, much harder in others (e.g. O.D.E.'s, non-linear optimization).

Documentation

For small software projects, the simplest approach is to keep all documentation in machine-readable form, in ordinary text-files (either embedded in the source-text or in separate files).

Text-processing utilities such as troff allow more elaborate documentation to be printed off as users require, but they are by no means universally available.

Printed documentation is likely to be worthwhile only for commercial libraries or for the most heavily used software (e.g. Linpack).

NAG's printed documentation is prepared using a typesetting program TSSD (developed at Harwell).

Other tools are used to derive, from the same textual data base, an abbreviated form in ordinary text-files for use in the On-line Information System.

TOOLPACK/1

A SUITE OF FORTRAN

SUPPORT TOOLS

Contents

TOOLPACK/1

EVOLUTION

Project Started in 1979

Funded by -

USA - National Science Foundation
Department of Energy
UK - SERC

Organisations Involved

University of Colorado
(Prof. Osterweil and colleagues)

Others

Argonne National Laboratory
Bell Communications Research
Jet Propulsion Laboratory
University of Arizona
Purdue University
NAG Ltd.

TOOLPACK PROJECT AIMS

1. To provide a suite of tools to assist the production, testing, maintenance and transportation of medium sized mathematical software projects written in standard conforming FORTRAN 77.

2. To investigate the development of extensible programming support environments built around integrated tool suites.

Notes:

A Standard conforming FORTRAN 77.

B Interactive Use (but with batch capability)

C Facilities for documentation etc.

D Portable across a wide spectrum of host systems

TOOLPACK/1 - Public Releases

NAG Ltd. took over responsibility for the public distribution of the results of the Toolpack project.

Some considerable effort was required to integrate the various parts of the research project into a form suitable for general use.

Early in 1985 the first release of Toolpack became available. By the end of last year about 400 copies had been distributed worldwide.

In the meantime further research funded by the SERC led to new tools. Some improvements were also made to the existing tools. This resulted in Release 2 of Toolpack/1 in January 1987.

There are now over 60 tools in the tool suite.

TOOLPACK/1 is:

Public domain software

Normally provided in source form

Also available are easy to install versions for:

VAX/VMS (Executables)
Unix (Source and makefile)
Apollo Domain (Executables)

TOOLPACK/1 TARGET FORTRAN

ANSI STANDARD FORTRAN77

with the following extensions:-

* Upper, lower and mixed case, $ and underscore in names
* Extra data types:-

INTEGER*2	DOUBLE COMPLEX
LOGICAL*2	LOGICAL*1
REAL*16	Holleriths

  REAL*8 etc. are also recognised
* No limit on symbolic name length
* Tab in initial label field
* Non numeric comment indicator

Use of non-standard features is still detected by the analysers.

TOOLPACK/1 TOOL SUITE

ANALYSERS

- Lexer
- Parser
- Semantic Analyser
- Portability Verifier
- Fortran Differencers (2)

TRANSFORMERS

- Name Changers (3)
- Declaration Standardiser
- Polisher
- Precision Transformer
- Structurer
- Real Number Standardiser
- Generic to Intrinsic Converter
- Format statement string standardiser

TOOLPACK/1 TOOL SUITE

GENERAL TOOLS

- Command Executor
- Data Comparison
- Fortran Aware Editor
- Merge Include Files
- Pattern Finder
- Macro processor
- File Splitter
- PFS File and Directory Save/Restore
- Text Differencer
- Version Controller

DOCUMENTATION

- Text Formatter
- Fortran Analyser and Report Generator

TOOLS - BASIC ANALYSERS

* ISTLX - Lexer or Scanner
* ISTYP - Parser
* ISTSA - Semantic Analyser

These tools are the first steps in making available the more complex functions of the other tools

The simpler tools require only ISTLX

More complex tools require ISTLX and ISTYP

The Portability Verifier requires ISTLX, ISTYP and ISTSA

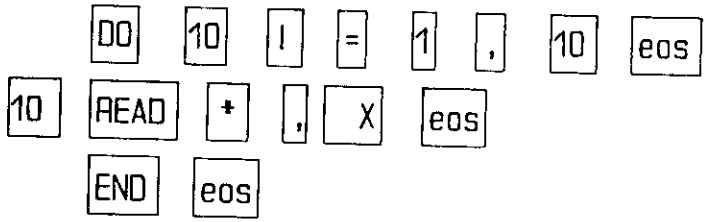
ISTLX
Lexer

Input: Source
Output: Token Stream

Fortran Input Program

```
      DO10I=1,10
10    READ*,X
      END
```

Token Stream

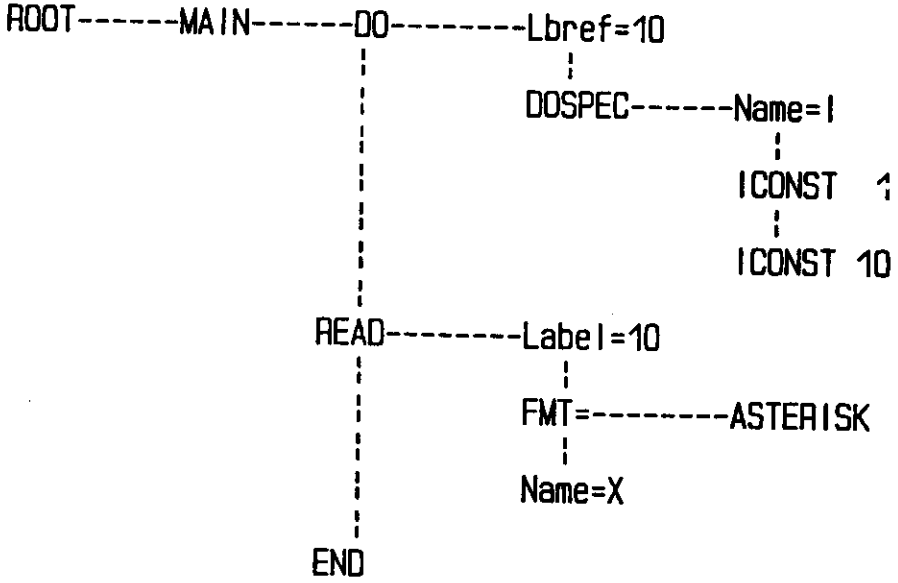


Integer values represent the various token types.
Some tokens have a value associated
e.g. TNAME "I"
TOCNST "10"

ISTYP
Parser

Input: Token Stream
Output: Parse Tree
Symbol Table

PARSE TREE



SYMBOL TABLE

1. String Table

   Text of the symbols

2. Symbol Table

   Symbol type
   Symbol name pointer to string table
   Further attributes depending on symbol type

The parser also creates a Comment Index, this relates comments to parse tree statements.

The vast majority of toolpack tools work at the parse tree / symbol table level.

ISTSA
Semantic Analyser

Input: Parse Tree
Symbol Table
Output: Extended Parse Tree
Extended Symbol Table

* Does checking for conformance to the Toolpack/1 target Fortran standard

* Adds further information to the parse tree and symbol table

  e.g. Recognises
  1. Constant sub-expressions
  2. Unsubscripted arrays

* Creates an attribute file. Pointers are added to the parse tree to access attribute information

  e.g. Program unit interface information

ISTPL
Polisher

Input: Token Stream
Output: Source

Input program

```
      PROGRAM                    TEST
            DO 10  I=1,   2
           IF (I.EQ.1) THEN
      X=2.0
                                      ELSE
          X=15.0
                ENDIF
10                    CONTINUE
      END
```

ISTPL

Output program

```
      PROGRAM TEST

      DO 10 I = 1,2
          IF (I.EQ.1) THEN
              X = 2.0

          ELSE
              X = 15.0
          END IF

   10 CONTINUE
      END
```

OPTIONS

Spacing	Case
Re-labelling	Continuation symbol
Line breaking	etc. etc.

Options can be set by an interactive program.
The default values will be acceptable to most users

Source to Source Monolith: ISTLP

ISTST
Structurer

Input: Parse Tree
Output: Source Code

Input program

```
      SUBROUTINE XXX
      A=1
      IF (A.EQ.B .OR. C.EQ.D) GOTO 200
      DO 10 C=1,10
      D=1
      IF (E) GOTO 220
10     CONTINUE
200    DO 210 F=1,10
      G=1
210    CONTINUE
220    CONTINUE
      END
```

ISTST

Output Program

```
      SUBROUTINE XXX

      A = 1
      IF (A.NE.B .AND. C.NE.D) THEN
          DO 10 C = 1,10
              D = 1
              IF (E) RETURN
 10       CONTINUE
      END IF

      DO 20 F = 1,10
          G = 1
 20   CONTINUE
      END
```

ISTST is a monolithic tool that rebuilds the flow of control of the program, converting it internally to token stream level. Finally the polish routines are called to output the source level structured and polished program.

Recommended tool sequence: ISTLY - ISTST

ISTPT
Precision Transformer

Input: Parse Tree
Output: Token Stream

Transforms single to double precision and vice-versa

Input Program

```
      REAL X
      X=0.5
      PRINT 9000,X
      Y=AMAX1(X,0.01)
9000  FORMAT(E10.6)
      END
```

Output Program

```
      DOUBLE PRECISION Y
      DOUBLE PRECISION X
      X = 0.500
      PRINT 9000,X
      Y = DMAX1(X,0.0100)
9000  FORMAT (D10.6)
      END
```

Source to Source Monolith: ISTQP

ISTDS
Declaration Standardiser

Input: Parse Tree
Output: Token Stream

1. Rebuilds declarative statements, declaring all names in a standard form.

or

2. Declares all undeclared variables.

Input Program

```
      SUBROUTINE SUB(A,B,C,I1)
      DIMENSION X(100)
      COMMON /T1/N2,A3,X
      W=A+B+C
      I=I1*10
      V=X(I)
      Y=FUNC(W,I,V)
      PRINT Y
      END
```

Source to Source DECS: ISTQD
Source to Source Precision Transformation and DECS: ISTQT - ISTDT

ISTDS

Output Program

```
      SUBROUTINE SUB(A,B,C,I1)
C      .. Scalar Arguments ..
      REAL A,B,C
      INTEGER I1
C      .. Scalars in Common ..
      REAL A3
      INTEGER N2
C      .. Arrays in Common ..
      REAL X(100)
C      .. Local Scalars ..
      REAL V,W,Y
      INTEGER I
C      .. External Functions ..
      REAL FUNC
      EXTERNAL FUNC
C      .. Common blocks ..
      COMMON /T1/N2,A3,X
C      ..
      W = A + B + C
      I = I1*10
      V = X(I)
      Y = FUNC(W,I,V)
      PRINT Y
      END
```

ISTPF
Portability Verifier

Input: Extended Parse Tree
Output: Reports errors

ISTPF checks that the program conforms to a subset of the ANSI standard that is known to be portable.

Input Program

```
      PROGRAM TEST
      DO 10 X=1,20
      READ *,Y
 10   CONTINUE
      END
```

Report

```
*  Error(s) have been detected by PFORT-77  *

Error: DO-loop index X not INTEGER in TEST
Warning: Variable set but not referenced - Y in TEST
```


ISTPF

Checks made by ISTPF
* That the program conforms to the ANSI standard

Plus
* A COMMON block must not appear in more than one BLOCK DATA subprogram
* In a COMMON block COMPLEX and DOUBLE PRECISION should come first
* A DO loop must have an integer control variable
* Saving of COMMON blocks is checked
* etc.

Inter Program Unit Communication

Unsafe references

* Constant or expression associated with a dummy argument which can be changed.
* Actual argument associated with 2 dummy arguments.
* Actual argument is in a COMMON block accessed by the called subprogram, one of which is modified.
* An actual argument is an active DO loop index and the associated dummy argument may be changed.

TOOLPACK/1 - Conclusions

Toolpack/1 provides a suite of software tools to aid the Fortran 77 programmer in:

- Fortran Analysis and Standard Checking
- Program standardisation
- Automated transformations

Toolpack/1 is Public Domain Software that can potentially be installed on almost any machine.

Easy to install versions are available on some machine ranges.

THE FUTURE OF TOOLPACK/1

Nag Ltd is actively working towards software tool support for Fortran 8X.
A new definition for the portability base will ease the installation task.