CERN 68-5
Data Handling Division
23 February 1968

ORGANISATION EUROPÉENNE POUR LA RECHERCHE NUCLÉAIRE
CERN EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

PROGRAMMING TECHNIQUES

D. Ball, T. Bloch, H. von Eicken, G.A. Erskine,
J. Garratt, R. Keyser, A. Maver and G.C. Sheppey

Lectures given in the Academic
Training Programme of CERN

G E N E V A
1968

Genève - Service d'Information scientifique - H/306 - 2000 - avril 1968

CONTENTS

*) See number in square brackets at the foot of the page.

PART I

THE CDC 6600 COMPUTER AND ITS OPERATING SYSTEMS--

GENERAL FORTRAN PROGRAMMING TECHNIQUES

THE CDC 6600 COMPUTER

by

T. Bloch

This series of lectures is meant to give an idea of the operation of the CDC 6600 computer. The aim of the first lecture is to cover the organization of the machine, in order to give the background necessary to the understanding of the later lectures on the SIPROS and SCOPE operating systems.

The organization of the CDC 6600 computer is best shown on a drawing where the input/output devices (I/Ø), the peripheral processors (PP's), and the central processor (CP) are set apart (see attached figure). The present lecture is an attempt to describe the communication between these basic parts of the computer.

The central processor has access to central memory (CM) only. The processor executes instruction after instruction of the programs which have been loaded into the memory. In the figure we see three of a series of registers which, for a program in execution, contain the beginning address (RA), the program length (FL), and the address in execution (P). Jumps inside a program can be made, but jumps outside of the region of a certain program (given by RA and RA + FL) are impossible since there is a special hardware protection against these jumps. As we will see later, a peripheral processor can switch the CP between programs in central memory by changing the contents of the registers in the CP by a special instruction.

The programs must be entered into the central memory by a peripheral processor. Any PP can read from or write on any I/Ø device; it also has access (read or write) to the whole of the central memory. Thus, the hardware protection which makes sure that one CP program does not overwrite areas outside its allocated memory has no counterpart for PP programs.

Certain of the rules to be obeyed by every PP program ensure that a correct PP program never writes into wrong areas in the CM. Protection against faulty PP programs is not possible -- they are likely to hang up the system.

One peripheral processor is used as organizational head of the system (MANAGER). It keeps track and directs the work of the other peripheral processors, it supervises the central processor continuously, and it tries to use the CP efficiently, i.e. not letting it wait for I/Ø. MANAGER (normally situated in PPO) switches the central processor from one program to another in case the present program is finished or is waiting for an I/Ø task to be completed. This switch instruction is called an exchange jump and works as follows: the registers in the CP (including RA, FL, and P) are changed to values defining another program in CM which is then executed next.

MANAGER also assigns I/Ø tasks generated by CP programs to peripheral processors which are idle. When the I/Ø operation is completed, the PP assigned to it tells MANAGER, and MANAGER knows that the PP is again idle and that it can switch (exchange-jump) the central processor back to the CP program which put up the I/Ø request (and was perhaps exchange-jumped out of the central processor for precisely this reason).

Of course, a lot of information is channelled through to MANAGER from all the system components in order to make it possible for MANAGER to decide which CP programs to run and when to run them. Examples of this type of information are, as we have seen:

SIS/kw/gs

i) the CP works on a program that needs an I/Ø operation to be performed (CP);

ii) an I/Ø operation for a CP program has been completed (PP);

iii) a PP wants its central processor program to get control (PP). This happens, for instance, when a PP transmits data directly from Luciole to tape and that tape is full.

We have seen how MANAGER communicates to the CP -- namely, by the exchange-jump instruction. There is no similar hardware interrupt available to a CP program which wants to communicate something to MANAGER -- all such communication is built up around conventions.

Such conventions rely on the two ways in which a PP can investigate the situation in the CP:

i) the PP can read the value of P;

ii) the PP can read any central memory word.

A working scheme of communication (from SCOPE) is outlined in the following:

The program address register P is periodically read by MANAGER. Normally P contains a value above n, the first n locations -- let us call it the communication area -- not being used for program instructions (for instance, n = 64 in the SCOPE-system). When something goes wrong hardwarewise (e.g. illegal jump or illegal instruction appears), a jump is made to P = 0 and the central processor stops. MANAGER sees this next time it reads P, and it takes the necessary steps to terminate the program. MANAGER also reads periodically the contents of location 1 of the central processor program running. This location normally contains 0, but when the CP program wants an I/Ø operation done by a PP it places information about the operation in location 1. This is seen by MANAGER next time it reads location 1, and it will find an idle PP to do the I/Ø task and clear location 1, thus telling the CP program that its I/Ø request has commenced.
MANAGER might, of course, remove the CP from the program and give it to another program in CM (exchange-jump) but this is not unavoidable (BUFFER IN, BUFFER ØUT in FORTRAN for example).

Peripheral processors can hardwarewise exchange information via a channel or via central memory. In both SCOPE and SIPROS only central memory is used -- the channels on the CDC 6600 all being tied up with I/Ø devices.

Apart from this choice there is the question of whether PP's are allowed to communicate freely or whether it all has to be channelled through MANAGER. Again, both systems have communication only between a "normal" PP on one side and MANAGER on the other side. The complications are very big indeed in the decentralized scheme, and there is no reason why it should be more efficient.

A typical example of difficulties which can be encountered in PP operation and can be solved by MANAGER communication is the following:

A CP program asks for an output (by setting location 1). MANAGER notes this, assigns the job to PP2, and switches the CP over to another program in CM. This program also asks for an output and this job is assigned to PP6. Although the outputs are on different tapes, these may be on the same channel and as they are more or less at the same time the outputs might interfere. Therefore, PP2 must reserve the channel before using it, and PP6 has to wait until PP2 releases the channel again.

Thus the flow of action for one unbuffered I/Ø request is as follows:

The CP program puts up an I/Ø request in location 1. MANAGER assigns the output job to PP2 and takes the control away from the CP program. PP2 checks and reserves the channel, writes onto the tape, and releases the channel. PP2 informs MANAGER that the I/Ø operation is terminated by setting a central memory word. MANAGER gives back control to the CP program.

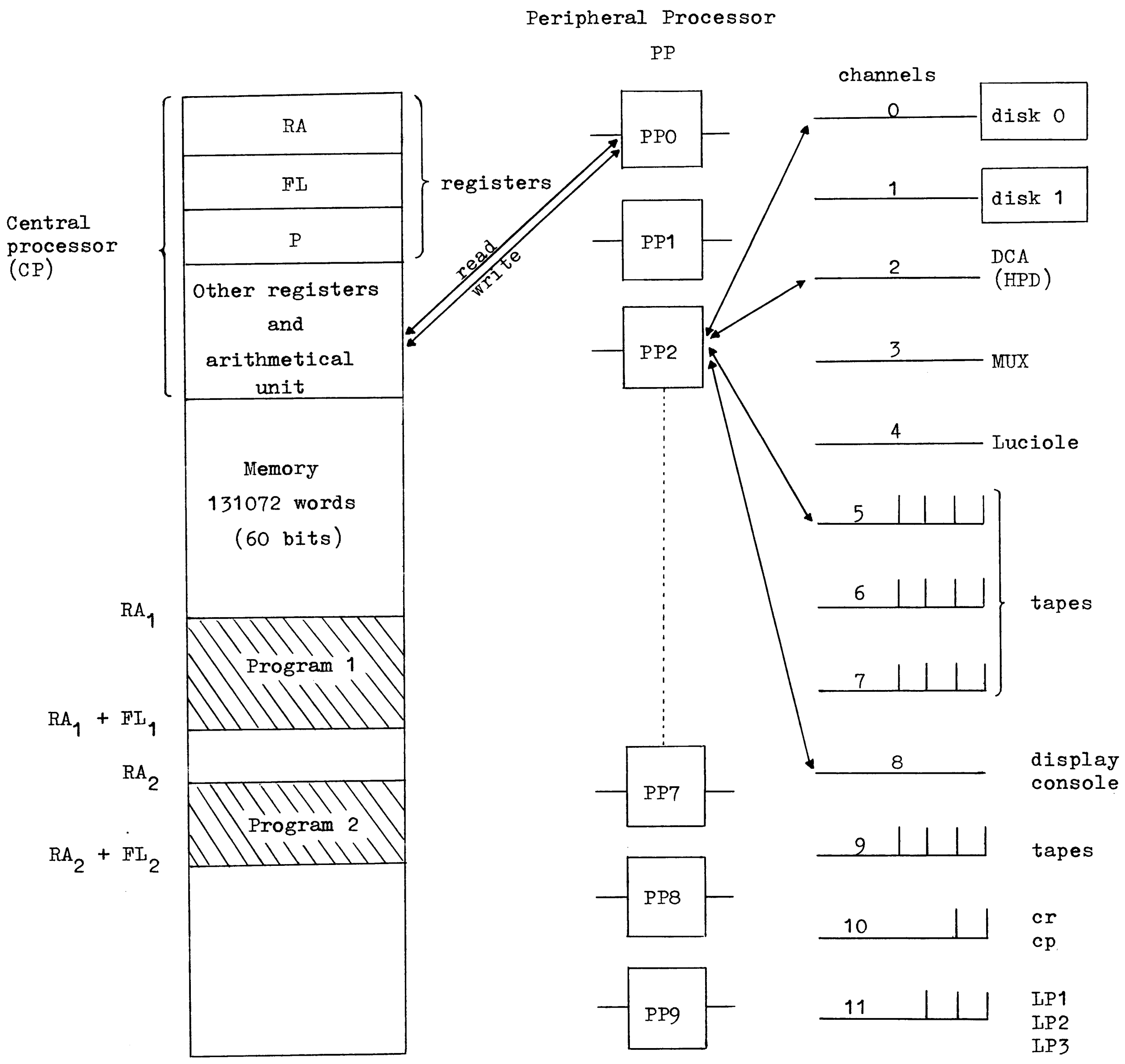
This is only one example from an area which is quite extensive. PP's must handle system tables to which no two PP's may have access simultaneously; they discover errors which require termination of the CP program; they arrive at situations where operator communication is indispensable; they provide accounting information, and so on. To cover all this would mean designing the operating system, and I am sure that the time will not allow this.

An operating system for the CDC 6600 is the set of programs needed to control the job flow through the computer, service the I/Ø requests, and provide the user CP programs with those facilities which are needed in order to achieve efficient execution both from a programming and a throughput point of view.

Such a system is constrained by the communication facilities available in the hardware, and I have tried to describe these as well as some of their effects on the operating systems in use at CERN.

It is hoped that the two lectures devoted to the SCOPE and the SIPROS systems will give a clearer picture of the situation than do these rather scattered examples of communication inside a CDC 6600 operating system.

RA :	reference address
FL :	field length
P :	program register
DCA :	data channel adapter
MUX :	multiplexor (tape lights, calcomp plotter, TTY)
cr :	card reader
cp :	card punch
LP :	line printer



SIPROS--SIMULTANEOUS PROCESSING AND OPERATING SYSTEM

by

D. Ball

This paper describes the CERN version of SIPROS, not the CDC one. However, the general philosophy of both is very similar.

To give an idea of the operation of the system, the passage through it of a typical job will be described. To do this, it is first necessary to explain the major constituents of the system and how the parts communicate with each other. The system is kept on tape and loaded from any one-inch tape drive onto one of the two disks, and from there into the central and peripheral processor memories. If it is necessary to reload the system later, it can be loaded directly from disk, which is considerably faster. After this loading process, the programs in the peripheral processors are:

PP0	EXECUTIVE
PP1	DISK EXECUTIVE plus a small display program
PP2	PRIMARY DISK SLAVE
PP3	SECONDARY DISK SLAVE
PP4	MULTIPLEXOR program
PP5-11_8	PP RESIDENT

The function of these programs will become clear later.

Into the low part of the central memory ("low" is used to denote that part containing addresses from 00000_8) are loaded tables and copies of some frequently used PP programs. Following these is a copy of the CERN FORTRAN compiler, starting usually around $33{,}000_8$. The central processor is "executing" a program with a completely zero exchange-jump package -- called the "idle program".

The EXECUTIVE program, which controls the whole system, is monitoring the card reader, waiting for it to be ready, signifying there are cards to be read. Associated with each PP is an area in CM consisting of eight CM words. These areas consist of:

INPUT REGISTER	(1 word)
ØUTPUT REGISTER	(1 word)
MESSAGE BUFFER	(6 words).

Since one CM word is 60 bits, and one PP word is 12 bits, it is customary to talk of a CM word consisting of 5 bytes, each of 12 bits. The bits of the CM word are numbered 0-59 from the right, and the bytes 1-5 from the right.

Each PP RESIDENT is monitoring the 5th byte of its input register, waiting for it to be set to 1 (they are set to zero initially). The PP Resident is a small program occupying the higher core locations of the PP in which it is stored.

The DISK EXECUTIVE is monitoring a table into which disk requests are placed, the two DISK SLAVES are monitoring a table by which the Disk Executive passes work on to them, and the MULTIPLEXOR is monitoring a table into which its requests are placed.

When the card reader becomes ready, the Executive instructs one of PP's 5-11_8 (the so-called POOL PP's since each of them can perform one of several jobs, and is allocated to the task if it happens to be free) to load the BATCH LØADER and to read the cards for one job. A job is defined as the deck of cards starting with an *JØB card and terminating with an $^8_9$FINISH card (if an $^8_9$FINISH card is not found before the next *JØB card, the next

SIS/kw/eg

JØB card is routed to the secondary stack of the card reader and the reader stops waiting for the operator to insert an $^{8}_{9}$FINISH card). The instruction is carried out by setting the PP's input register to

$$1\ 0\ 0\ 0\ 14_8$$

The 1 signifies to the PP Resident that there is work to do, and 14_8 is the address of a CM word which points to a table containing details of the request. Each request is three CM (= 15 PP) words long. One item in it is the internal job number. This unique (at any one time) number is used throughout to identify the job, rather than the external programmers' name and account number which may not be unique.

The PP Resident examines the request, discovers it must be processed by the Batch Loader, and using tables in the lower part of CM finds out whether it is stored on the disk or in CM and loads it. In fact the Batch Loader is stored on the disk, so the loading process involves the Disk Executive and Slaves, as all disk transfers are handled by these packages to optimize disk transfers. This is necessary since the time taken to reposition the disk heads is very long compared to the time needed to actually transfer the data. By placing all requests for disk transfer under one program it is possible to optimize disk activity, in that before repositioning the heads all requests from various programs which need to read or write from the present track can be processed.

The PP Resident requests disk transfers by placing a three-CM word request in its message buffer and setting the top byte of its output register to 7 (a non-zero value in this byte informs the Executive that it must take some action; the particular one depending on its precise value). The request is passed to the Disk Executive by the Executive which places it in the table which the Disk Executive is periodically scanning. One other piece of information in the three-word request is the address of a CM status response word. This word is used for communication between the program initiating the request and the program processing the request. In this case, the PP Resident will be told by the Disk Executive when the transfer of information from the disk to CM (via the memories of the slaves) has been completed.

After the Batch Loader has been read into the PP memory, the Resident transfers control to it, by jumping to location 100_8 with the A register containing the address of the 15 PP word request, which the PP Resident had transferred from CM to its own area in the PP.

The Batch Loader needs the card reader and its associated channel to read cards. Other packages also require equipment and channels; in some cases two different packages, or two copies of the same package in different PP's, require the same equipment. There is no hardware protection to prevent two PP's attempting to use the same equipment simultaneously, so it must be done by software. With each piece of equipment is associated a CM location which indicates whether the equipment is down (not working), in use by a PP, or available, amongst other information. For each channel is similarly associated a word saying whether it is reserved for use by a particular PP or is free.

The Batch Loader checks the words for the card reader and channel, then changes them to indicate it is using them. It also reserves two buffers of 1000_8 CM words each, from a pool of such buffers available to all PP programs.

Assume the cards for a job are:

```
*JØB, SØMEBØDY, 000014/CØM
*LIMITS, 2
*TAPE 5    CØMMØN
*DUMP, 4000, 6000
*BINARY
*FULL
   FORTRAN source routine 1
*BINARY
   FORTRAN source routine 2
   FORTRAN source routine 3
   Binary cards
*DATA
   Data cards
8
 FINISH
9
```

The Batch Loader reads cards and transfers them to one of its two CM buffers (one card is stored as 16 CM words). From the system control cards it builds up the job table entry for this job, which contains such information as the equipment the job needs, what terminal dump, if any, it requires, etc. This prevents continual scanning of the control cards by many programs. (The control cards such as *BINARY, *OPTIMIZE, are compiler, as opposed to system, control cards, since they instruct the compiler how to process the next source deck; these cards are not processed by the Batch Loader, they are simply copied into the buffer.) When one of the CM buffers is full, the Batch Loader requests, via its PP Resident, that it be written onto disk. While this buffer is being transferred, the Batch Loader continues to read cards into the other buffer. During the reading of binary cards a serialization check is carried out, as this can be a useful aid in finding incorrectly made-up binary decks. When the *DATA control card is found (or the $^{8}_{9}$FINISH card, if the job has no data) an End-of-File indicator is placed in the buffer, and the buffer written onto disk. Any data is placed in a separate file on the disk. The starting addresses of the files for the job are placed in the job table entry, the status of the job is set to "waiting to be compiled", and the entry written into CM. The status of a job is determined by a word in its JTE which reflects which stage the job has reached, and is used by the Executive in its scheduling of work.

After the Batch Loader has read the cards for one job it returns control to the PP Resident which scans its input register to see if any further work has been scheduled for this PP. If the card reader is still ready, it is probable that another Batch Loader request is waiting since, in order to avoid unnecessary loading of programs, tasks of a similar nature are sent to the same PP as far as possible. However, switching becomes necessary at some stage, as there are more different tasks than there are PP's. (To avoid complicating the description, the fact that, for efficiency, the Batch Loader program is combined with the Card Punch program since the reader and punch are on the same channel, has been ignored.)

Since this is the first job in, the compiler is loaded and available, so the Executive will exchange-jump the central processor to FORTRAN 1 (it is possible to have two copies of the compiler loaded at the same time, so they are called FORTRAN 1 and FORTRAN 2, to distinguish them). With the compiler is a copy of a program called the CENTRAL RESIDENT which occupies 1000_8 CM words starting at relative location zero. Following this are 4 buffers, each of 1000_8 words. These are:

starting at 1000_8	Print buffer
2000_8	Punch buffer
3000_8	Job Stack buffer
4000_8	Card Read buffer

The compiler proper therefore starts at 5000_8 relative. The Executive plants information in cells in the Resident pertinent to the particular job being compiled. The Resident acts as a standard interface between the compiler and the Executive. Since central processor programs cannot perform I/Ø directly, there must be a software communication between the running CP program and the Executive. The latter is periodically looking at location 1 of the running program. When the CM Resident wishes to pass on I/Ø requests, it builds up a request in locations 2-8 and then sets in location 1 information concerning the type of request. Whenever location 1 becomes non-zero, the Executive extracts the information in locations 1-8 and builds a three-CM word request from it, of the same form as those built by PP programs. Control is then normally switched to another central program. In the present example the first I/Ø request from the compiler to the resident is to read a card. The resident does not wish to ask the Executive for every single card, instead it asks for its Card Read buffer to be filled. The Executive builds a three-word request for the Disk Executive to read the next 512 buffer from the source file of the job and place it into the Card Read buffer of the compiler. While this is in process, the compiler can do nothing, so control is passed to another program (probably the idle program in this case, until other jobs have been read in). The Executive sets the status of the compiler to "waiting on I/Ø" (the job itself has the status "being compiled" until compilation is complete). For all such jobs, the Executive scans the status word associated with the three-word request to know when the operation is complete. It then sets the status of that CM program to "waiting in CM". Periodically the Executive looks at jobs whose status is "waiting in CM", and if one of them has a priority higher than the currently running job, control is switched by exchange-jumping the two programs. (Priority of a job has not been discussed. It is sufficient to mention here that every job and every request has a priority level, and that resources, be they the central processor, a PP or whatever, are used to satisfy the highest priority request first.)

The Central Resident is now in a position to satisfy up to 32 Card Read requests without further requests to the Executive. During the compilation there will be binary card images produced, and information to be printed. These are packed by the Resident into the appropriate buffer, and complete buffers written onto the disk as necessary. The punch buffer file contains the binary card images of any routines which were preceded by a compiler control card *BINARY. The job stack file contains binary card images for all routines in the job deck, whether they were binary or source. (The compiler streams binary cards from the deck, straight into the job stack; this mechanism allows source and binary routines to be mixed in any order.) When the compiler reaches the end-of-file of the source file, it sets a flag for the Executive to say whether any fatal compilation errors were encountered, and then executes a stop instruction. Another of the functions of the Executive is to check if the running job has stopped. To do this it has to read regularly the P counter and examine the instructions at that location, if it appears P is not changing. (One of the major design shortcomings of the CDC 6600 is that it is extremely cumbersome and time-consuming to detect if a program in central memory has stopped.) The Executive will exchange-jump the central processor to another job and set the status of the job to "waiting on disk". Jobs with this status are checked periodically and the equipment requirements for the one with highest priority examined. The only requirement for our job is one 1" tape which will be available. The Executive will allocate one particular tape drive to this job and build two three-word requests; one which will go to the Multiplexor package to flash the tape display lights, and the other for the MAGNETIC TAPE package to check the tape label. The latter request will go to a pool PP which will load the Tape package into its memory and then transfer control to it. When a common tape has been loaded and its label checked by the tape package, the Executive will build another three-word request for the multiplexor package to blank the display lights. The Executive will now check that there is sufficient CM space for the job to be loaded, reserve space, and build a three-word request for the JOB LOADER (the space allocated for a job is either derived from the *MEMORY control card, or a standard value assumed as in this example). The three-word request is passed to a pool PP which will load the Job Loader into its memory and transfer control to it. The Job Loader will use the Disk Executive to read the Job Stack file, a buffer at a time. The program will undoubtedly require routines from the library file on the disk, and the Job Loader uses a central library directory which is stored in CM to locate these. It is possible that during the loading phase it is discovered that the job requires more space than has been allocated to it. At this stage it is not possible to exceed the limit, as probably the extra space needed is occupied by another job. Instead, the Job Loader changes to a pseudo-load process. It continues to build up tables, etc., as before, but does not store anything else in CM. By this process it is possible to compute the exact amount of storage required for this job. At the completion of its task, the Job Loader informs the Executive whether it was a successful load or not, whether it was rescheduled for another time and, if it was loaded, how much space if any is left over. Any space not used is returned to the pool of available space. (A table is retained for each 512 word block of CM, the entry for each block signifying whether it is in use or not.) If the job was rescheduled, the Executive builds another Job Loader request when sufficient space becomes available.

For this job, the first 4000_8 locations comprise:

$0000\text{-}0777_8$	Control Resident
$1000\text{-}1777_8$	Print buffer
$2000\text{-}2777_8$	Punch buffer
$3000\text{-}3777_8$	Card Read buffer.

(Extra buffers would be allocated to a job with extra print or punch files.)

The Central Resident is identical to the one used with the compiler and serves the same purpose -- all I/Ø is carried out via this interface. The status of the job will be "waiting in CM". Control will be switched to this job when its priority determines. The job runs until the Resident informs the Executive that it requires some I/Ø carried out, a job of higher priority pre-emps, it exceeds its time limit of 2 minutes central processor time, or it stops.

When the central processor part of the job has finished, the Executive builds a three-word TERMINATION request. The termination package puts any dumps required in the print file of the job, and finishes the print file for the job with the accounting information which appears at the end of every job. It then writes any partial buffers in the appropriate file. It builds a printing and, if required, a punching request for the job, and places each in turn in the message buffer of the PP and signals to the Executive by setting the PP output register. These requests are sent by the Executive to pool PP's for printing and punching. Only when termination has signalled to the Executive that it has finished its task will the CM space occupied by the job be released and marked as not used.

After the printing and punching packages signal they have finished their tasks, the Executive builds an ACCOUNTING request for the job. The ACCOUNTING package writes a record of information containing pertinent statistics for the job onto the accounting tape, releases the job table entry for the job, and requests the Disk Executive to release all disk space used by the job. Now SIPROS is purged of that job and will reallocate the internal job number to another job entering the system.

MAGNETIC TAPE HANDLING

by

H. von Eicken

PREFACE

The purpose of this note is to give information about the handling of magnetic tapes from different aspects:

a) Physical properties of tapes and their transports.

b) Logical layout of information recorded on tapes.

c) CERN's tape labelling scheme.

d) A programming example for efficient tape use on the CDC 6600.

e) BUFFER IN and BUFFER OUT statements on the CDC 3800.

All information given is based on the hardware available and operating systems used at CERN's central computer installation at the end of 1966. Publication of this lecture note has been made possible through permission granted by Control Data Corporation to reproduce parts of its Magnetic Tape Transport Reference Manual; Chapter I and its figures are largely derived from this source; other chapters contain some CDC material.

SIS/kw/mn

CONTENTS

I. PHYSICAL PROPERTIES OF TAPES AND THEIR TRANSPORTS

1. CONSTRUCTION OF A TAPE

The tape has a mylar base and is coated on one side with minute particles of iron oxide mixed with a binding agent. It is upon this coating that information is recorded. To give an idea, each of these particles is less than one micron in length (1/1000 of a millimetre or 0.000039 inch) and in the form of a cigar. The thickness of the coating is 0.00045 inch and must be uniform and smooth along the whole tape.

2. TAPE PATH THROUGH TAPE UNIT

The information is read (detected) or written (stored) by passing the oxide side of the tape over read/write heads contained in the tape transport. Figure 1 illustrates the tape path through a tape unit (CDC 607 or 626 tape transport). During a read/write operation the tape is moved from the supply reel, pastthe read/write heads, to the take-up reel. The tape motion is provided by two fluted capstans, which rotate continuously in opposite directions. Tape is drawn against the drive capstan by vacuum and floated over the non-driving capstan by air pressure. If the tape is moving from the supply reel to the take-up reel (forward motion), the left capstan drives the tape and vice versa if it is moved in reverse direction. Tape motion is stopped by means of a pneumatic brake port. Tape is drawn to and firmly held against the brake port by means of vacuum. Because pressure is applied to both capstans during this period, neither capstan contacts (drives) the tape.

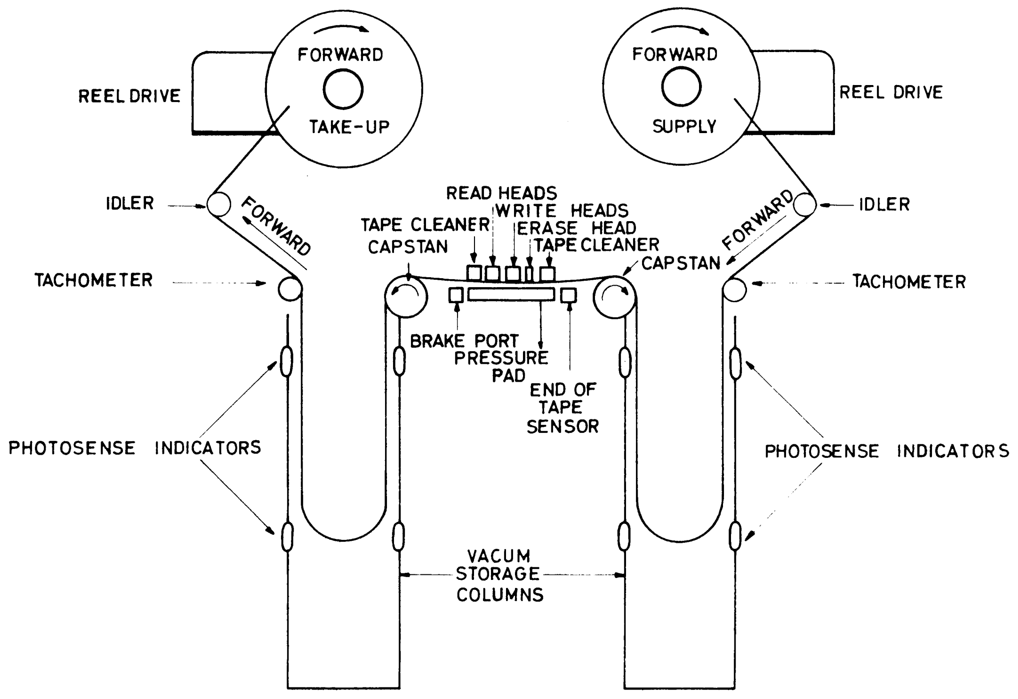


Fig. 1 Tape path through a CDC 607 or 626 tape transport

3. HEAD ASSEMBLY

The head assembly consists of individual read and write heads, an erase head, tape cleaners, and pneumatic pad. Each of the read/write heads has two magnetic gaps. One gap is used for writing; the other for reading. The gaps are arranged so that during a write operation the tape first passes under the write gap to record the data and then under the read gap to check the writing. This allows each frame of information to be examined and verified immediately after it is written on the tape. Thus if any discrepancy occurs during the write operation, it is detected at the read head. The broad band erase head removes any information recorded on the tape before new information is recorded by the write heads. The two tape cleaners, located on either side of the heads, pneumatically remove foreign particles on the tape surface during a read or write operation. The pneumatic pad maintains a precise contact pressure between the tape and the head gaps. This is provided by means of air pressure which minimizes read and tape wear by blowing the tape against the heads. The tape transport can accelerate tape to high speed within 2.5 (± 0.5) msec. Conversely, tape motion is completely stopped within 2.25 (± 0.5) msec. Vacuum loops in the storage columns minimize the tape mass that must be accelerated or stopped within this period by separating the heavy tape reels from the portion of tape under the heads.

4. NON-RETURN-TO-ZERO RECORDING

During a read or write operation, several recording heads, one head per recorded track, are placed vertically across the tape. As many bits as heads may, therefore, be simultaneously recorded, one bit per track. A non-return-to-zero scheme is used for recording (change on "1's"). In this system, magnetic particles on the tape are aligned in either the positive or negative direction. A binary "1" bit is recorded by reversing the alignment (polarity); no polarity reversed results in a "0" bit (Fig. 2). Thus each track of the tape is fully magnetized and the polarity is reversed as each "1" bit is recorded.

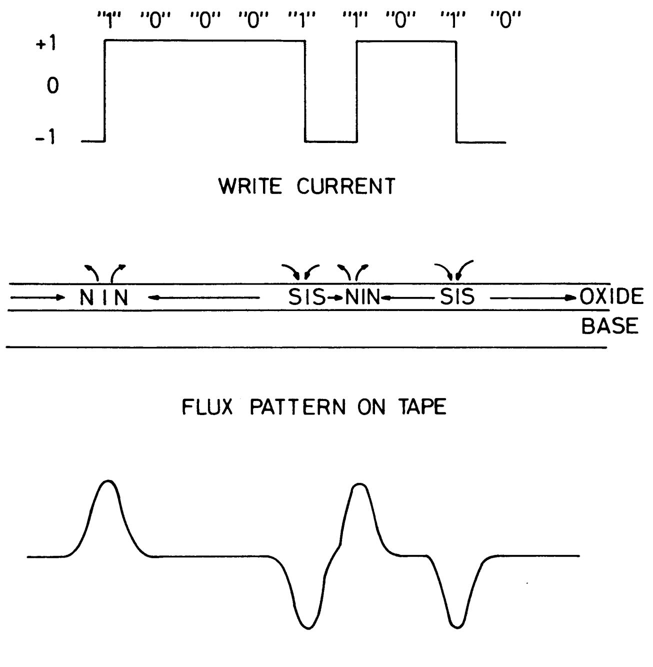


FIG. 2

5. 7 AND 14 TRACK TAPES

On our CDC 6600 we have two types of tapes which differ by width. The 626 tape transport records on 1-inch tape, the 607 tape transport on ½-inch tapes.

On 1-inch tapes the information is recorded with 14 tracks -- 12 data and 2 parity tracks (Fig. 3). A frame of data thus consists of two 6-bit binary data characters and one parity (check) bit for each character. Tracks 0 through 5 and 7 through 12 specify the characters, while tracks 6 and 13 hold the parity bits. The upper and lower characters in each frame correspond to one data word from the Peripheral Processor Unit (PPU). The parity bits are added by the tape controller. The data is recorded in binary format (just as they are represented in memory). The parity bit is chosen, so that the total number of "1" bits for each character in a frame is odd (odd or binary parity). The total number of "1" bits per frame is therefore even.

The information on tape is recorded in groups called records and files. A minimum of one frame of information constitutes a record. Adjacent records are separated by a 1-inch unrecorded area (inter-record gap). A longitudinal parity frame is recorded at the end of each record so that the total number of "1's" in each record track is made even. (This frame is not parity checked for vertical parity). During input/output operations the tape controller checks each character for correct vertical parity and each record for proper longitudinal parity. If a parity error occurs, a parity error status bit is set, which can be tested by the PPU.

A file consists of a group of records. Adjacent files are separated by recording an end-of-file mark at least 6 inches from the last record in the file. The file mark consists of an octal 17 and its longitudinal check character (also an octal 17) in the four lower order bit positions. The upper bits are all zero (including the parity bits). Thus reading/ writing an EØF mark always causes a parity error for vertical parity.

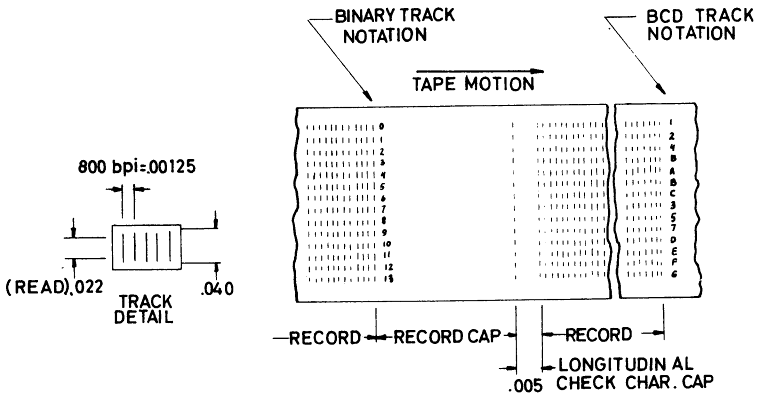


FIG. 3

Half-inch tapes are recorded with seven tracks (Fig. 4). Tracks 0 through 5 specify the characters, while track 6 holds the parity bits. Two frames, 6 bits each, correspond to one data word from the PPU. Data is recorded in either binary or BCD format (as represented in memory). There is no conversion carried out by the controller for BCD recording. The only difference between these two formats is: binary is recorded in odd, BCD in even vertical parity.

The inter-record gap for ½-inch tape is ¾ inch, and the file mark gap is at least 6 inches for ½-inch tapes. But there is a major difference between the two tape transports, the 607 for ½-inch and the 626 for 1-inch tapes. The 626 tape transport always records information with a density of 800 bits per inch (bpi) on each track, and it can only write tapes in odd parity. The 607 tape transport can record with 200 bpi, 556 bpi, and 800 bpi, and will write tapes in either BCD or binary mode.

Some other details about transfer time, start/stop time, etc., are given in Table 1.

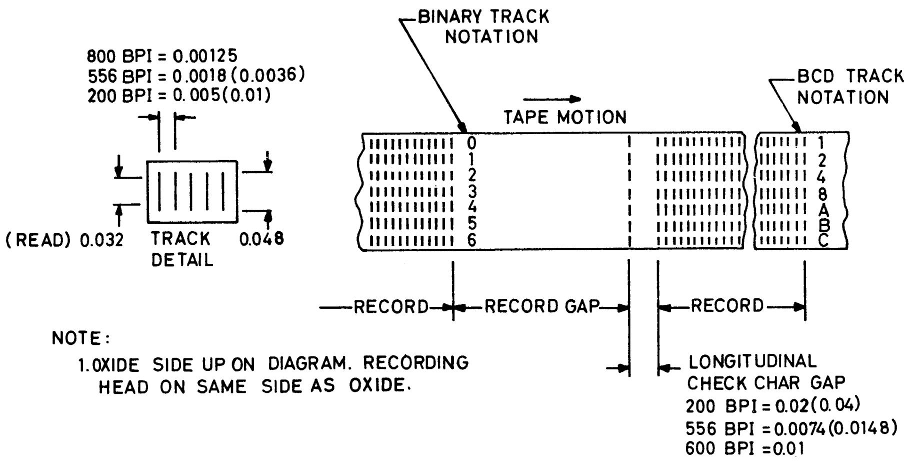


FIG.4

6. TAPE MARKERS

Reflective spots are placed on the tape to enable the tape unit to sense the beginning and the end of the usable portion of the magnetic tape. The reflective spots are plastic, 1 inch long by ³⁄₁₆ inch wide, coated on one side with adhesive strips and on the other with vaporized aluminium. They are placed on the base or uncoated side of the tape where they can be detected by photo-sensing circuits. Tapes at CERN have two loadpoints; in between them, a label to identify the tape has been written, and one end of tape marker. The reflective marker placement can be deduced from Fig. 5, which contains only one loadpoint. Figures 6 and 7 show the different layout for the two kinds of tapes used at CERN. The distance between first and second loadpoints is 4.5 to 5.0 m for 1-inch and 0.3 m for ½-inch tapes.

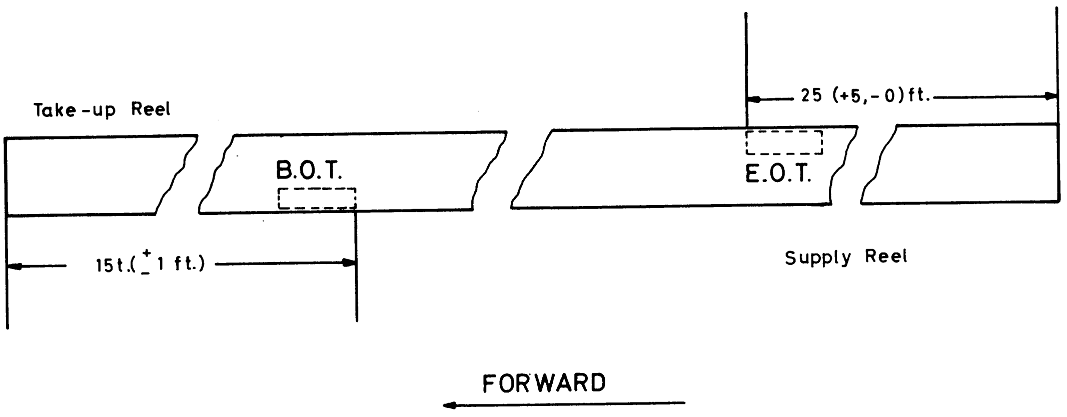


FIG. 5

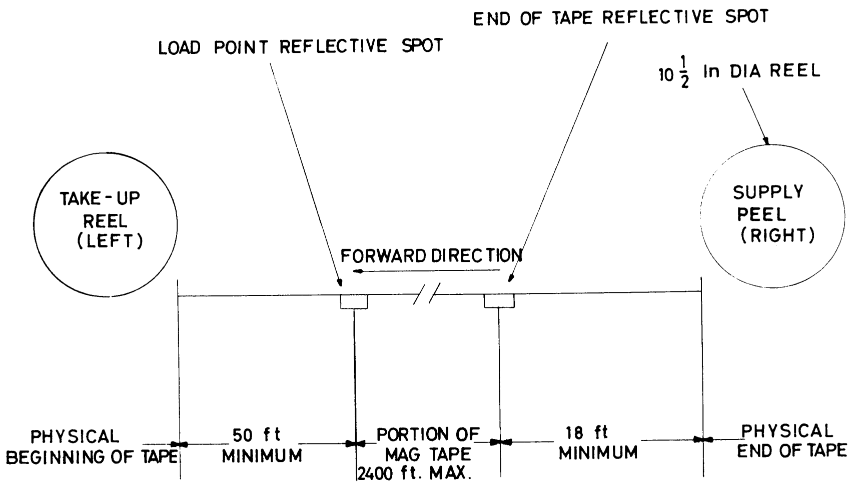


FIG. 6

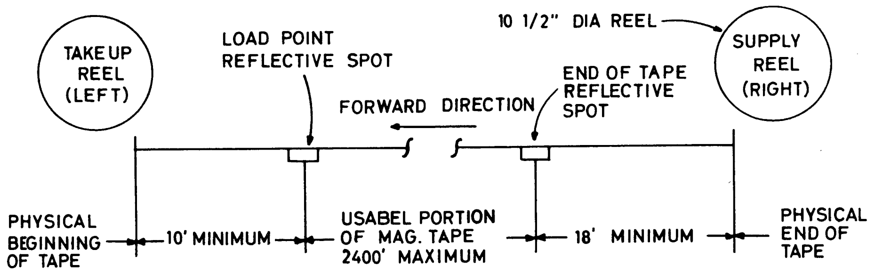

10 1/2" DIA REEL
TAKE UP REEL (LEFT)
LOAD POINT REFLECTIVE SPOT
END OF TAPE REFLECTIVE SPOT
SUPPLY REEL (RIGHT)
FORWARD DIRECTION
PHYSICAL BEGINNING OF TAPE
10' MINIMUM
USABEL PORTION OF MAG. TAPE 2400' MAXIMUM
18' MINIMUM
PHYSICAL END OF TAPE

FIG.7

7. FILE PROTECTION

The back of the tape has a slot near the hub which accepts a plastic file protection ring. Writing on tape is possible only when the reel contains the file protection ring. If an attempt is made by the program to write on a tape not containing the file protection ring, an error status is returned and the program normally is terminated. The tape may be read with or without ring. Presence of a write ring on a reel is signalled by the overhead lights in the tape transport, which come on immediately after the tape load process is executed. The ring should be removed from the tape reel after the run is completed to avoid loss of valuable records through accidental rewriting.

8. SUMMARY

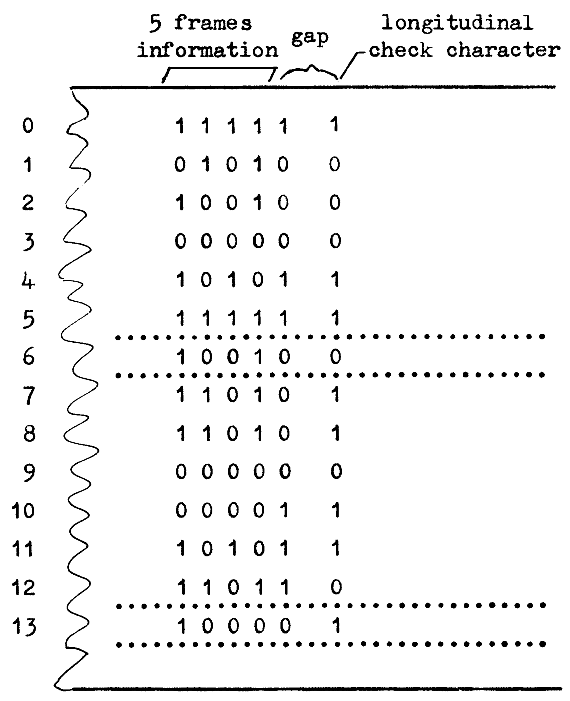
The following example illustrates the information given so far.

Memory cell ALPHA is assumed to contain:

63654343206143477061
C E L L   A L P H A

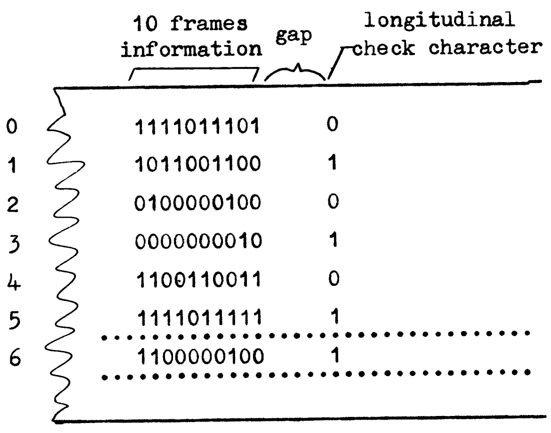
How would this information look on 1-inch and ½-inch tape? What would be the difference between binary and BCD write request?

a)



The dotted lines show the parity tracks. Information is stored beginning with track 12 through 0. So, beginning with track 12, the first frame contains 6365.

b)



```
0   1111011101   0
1   1011001100   1
2   0100000100   0
3   0000000010   1
4   1100110011   0
5   1111011111   1
6   1100000100   1
```

The dotted lines show the parity track. Information is stored beginning with track 5 through 0. (First frame: 63).

Both figures show how the information would look after execution of the binary write request. If the same memory cell had been written with a format specification of A10, the 1-inch tape would look the same, but the parity bits in track 6 of the $\frac{1}{2}$-inch tape would be complemented. Figure 8 shows how the information is actually recorded on tape; it shows a developed tape.

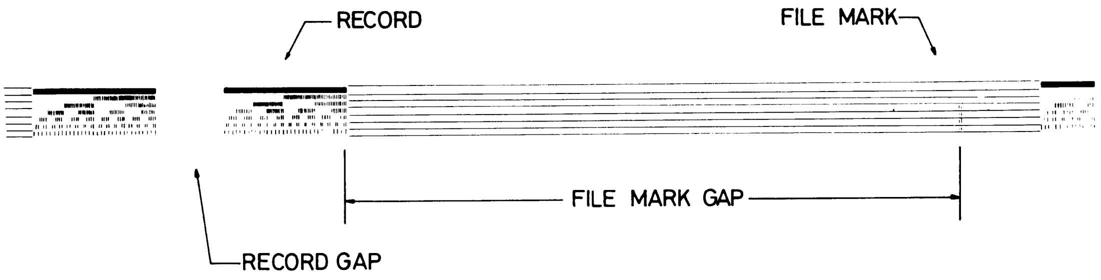


FIG.8

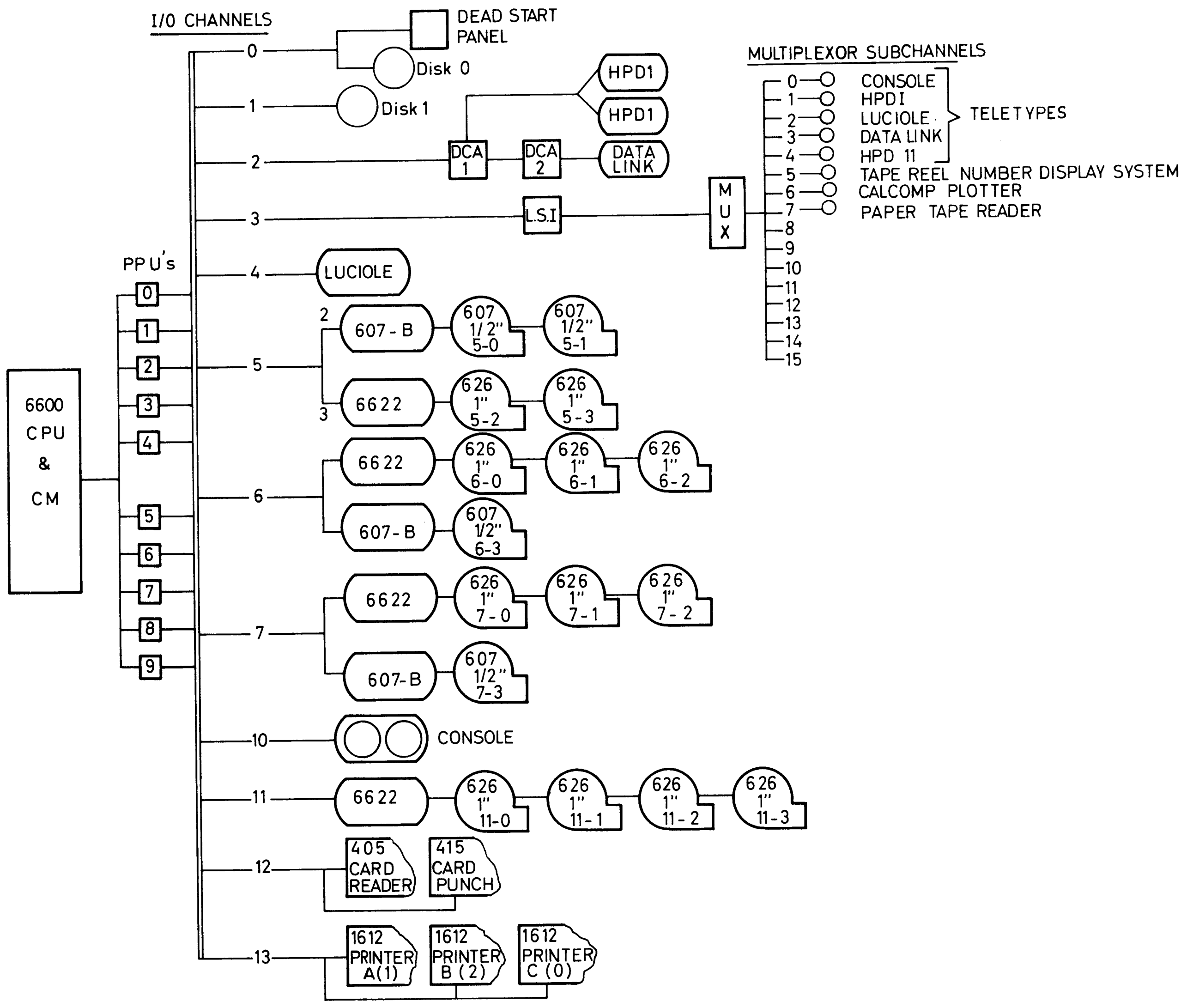

I/O CHANNELS
PPU's
6600 CPU & CM
0
1
2
3
4
5
6
7
8
9
DEAD START PANEL
Disk 0
Disk 1
HPD1
HPD1
DCA 1
DCA 2
DATA LINK
L.S.I
MUX
LUCIOLE
607 - B
607 1/2" 5-0
607 1/2" 5-1
6622
626 1" 5-2
626 1" 5-3
6622
626 1" 6-0
626 1" 6-1
626 1" 6-2
607 - B
607 1/2" 6-3
6622
626 1" 7-0
626 1" 7-1
626 1" 7-2
607-B
607 1/2" 7-3
10
CONSOLE
11
6622
626 1" 11-0
626 1" 11-1
626 1" 11-2
626 1" 11-3
12
405 CARD READER
415 CARD PUNCH
13
1612 PRINTER A(1)
1612 PRINTER B (2)
1612 PRINTER C (0)
MULTIPLEXOR SUBCHANNELS
0 CONSOLE
1 HPDI
2 LUCIOLE
3 DATA LINK
4 HPD 11
TELETYPES
5 TAPE REEL NUMBER DISPLAY SYSTEM
6 CALCOMP PLOTTER
7 PAPER TAPE READER
8
9
10
11
12
13
14
15

FIG.9

9. TAPE CONTROLLERS OR SYNCHRONIZERS

As pointed out by Mr. T. Bloch in his notes, all input/output handling on the 6600 is carried out by ten Peripheral Processor Units (PPU's) using the twelve I/Ø data channels available. Each data channel is bidirectional, not buffered, and can transmit a 12-bit data word every μsec, or major cycle, in either of both directions. Several peripheral devices may be physically attached to each data channel. Figure 9 illustrates the layout for CERN's 6600. Although there are several peripheral devices attached to one data channel, only one device may communicate with the PPU at one time. To accommodate this feature, each device has a unique function select code to connect it to the data channel. The code is contained in the upper three bits of the 12-bit function code sent out on the specific data channel by the function instruction of the PPU. Only the selected device responds to this code.

As Fig. 9 shows, the twelve 1-inch and four ½-inch tape transports available are attached to four different channels. They are not directly connected to the data channel but controlled by two different types of tape controllers or synchronizers: the 6622 synchronizer, which has one read/write control to control from one to four 626 1-inch tape transports, and the 607-B synchronizer, which also has one read/write control to control from one to four 607 ½-inch tape transports. Each tape unit belonging to a synchronizer is uniquely defined by a unit number between 0 and 3, which can be dialled on the tape transport.

Table 2 contains the different function and status codes for both types of controllers. As already stated, the 6622 synchronizer allows binary write and read only. Compared with other types of controllers, used for instance on the CDC 3800 at CERN, these two types of controllers are pretty poor. They do not have functions for

Search Forward EØF
Search Backward EØF
Skip bad spot
Select 200, 556 or 800 bpi

to mention some of the major deficiencies. But they have undoubtedly a big advantage -- they are much cheaper. It might be worth while to note that all adding and checking of parity bits is carried out by the controller and not by the tape transport.

II. LOGICAL LAYOUT OF INFORMATION RECORDED ON TAPES

The logical layout of information recorded on tapes is determined by the operating system used. This chapter will explain the actions taken by the SIPROS operating system for the various FORTRAN tape handling statements.

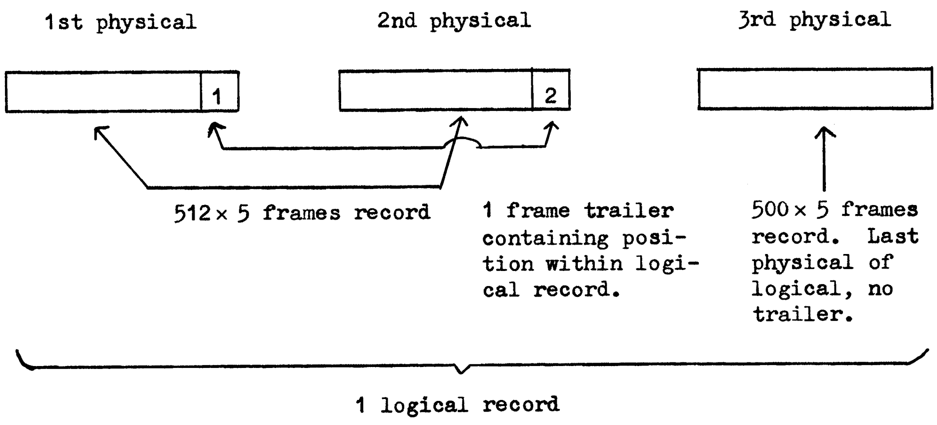
1. WRITE (I) LIST

The list items of a binary write request constitute a logical record, which may be of any length. If it is larger than 512 words, the system will split it up into as many physical records of 512 words each and a last physical record of $\leq$ 512 words.

All physical records, except the last of a logical record, will be written on tape followed by a 12-bit trailer, containing the position number of that physical record within the logical record.

WRITE (5) (A(I), I = 1,1524)

will result in the following record picture on tape (1-inch tape assumed):


1st physical
2nd physical
3rd physical
1
2
512 x 5 frames record
1 frame trailer containing position within logical record.
500 x 5 frames record. Last physical of logical, no trailer.
1 logical record

2. WRITE (I,100) LIST

All BCD or formatted I/Ø is done in the form of so-called unit records. The unit for card handling is the card image always containing 80 characters. The unit for printing is the line, always having 136 print positions. Deduced from here the I/O buffer for formatted tape handling is 140 characters. Therefore each physical record is exactly 140 characters long. The logical record structure given by the list is not maintained at all. There is no way of telling from tape what belongs together. Each logical record is equal to a physical tape record and always consists of 140 characters.

```
WRITE (5,100) A, B, C
100 FØRMAT (A10, //2A10)
```

will result in 3 physical records on tape, 140 characters each, where the first 10 characters of the first record represent the information in A, converted with format A10, followed by all blanks, the second record consists of 140 characters, all blanks, and the third record contains in its first 20 characters the information stored in B and C converted with format 2A10, followed by all blanks.

3. READ (I) LIST

4. READ (I,100) LIST

As opposite to the above.

5. REWIND I

Causes the PPU to issue a function code to bring the tape specified back to the load-point (2nd at CERN).

6. <u>END FILE I</u>

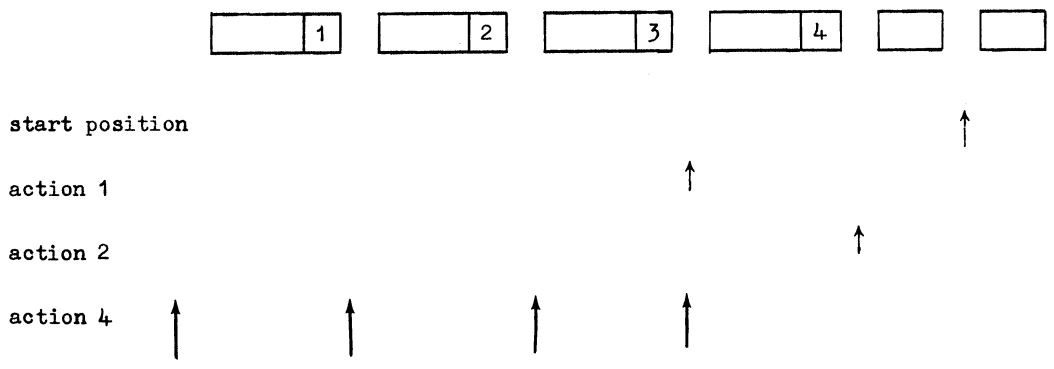
Causes the PPU to issue a "WRITE FILE MARK" request.

7. <u>BACKSPACE I</u>

Expects to position the tape to the beginning of the previous logical record on tape. As the BACKSPACE request issued by the PPU to the tape only goes back to the beginning of the previous physical record, some more action by software is necessary.

1st action:	backspace two physical records
2nd action:	read one physical record
3rd action:	was it longer than 512 words? (if no, tape positioned)
4th action:	execute as many physical backspaces as specified by the contents of the trailer.

↑ indicates the position of the read/write head on tape.



8. <u>CALL WIND (I)</u>

As there exists no function code to search for a file mark, the PPU will read physical record by physical record and always check after each record whether an EØF-mark has been encountered or not.

9. <u>CALL BACKZF (I)</u>

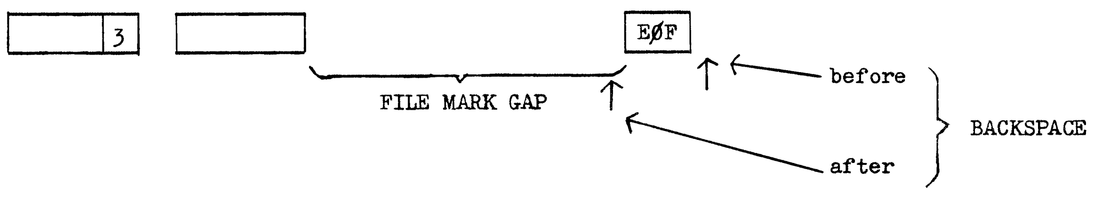
As there exists no function code to search for a file mark, the PPU will start by backspacing a physical record and reading it, check for EØF, and if not found, then continue by backspacing two, read one, check, until it finds an EØF.

10. <u>ERROR HANDLING</u>

10.1 <u>Write parity</u>

Let us assume the PPU found that it got a write parity indication back from the synchronizer when it was writing the n^{th} physical record of a logical record. The PPU will then backspace to the beginning of the bad physical record and write a file mark, which erases 6 inches of tape. It will then backspace over the file mark and try again to write the r cord. It will repeat this procedure up to five times, and if it still fails with parity error, it will give up. The tape therefore is positioned after the last bad copy of this physical record.

Note: Backspace over a file mark positions the tape in front of the actual file mark and not in the inter-record gap following the last record!



10.2 Read parity

The PPU, when getting read parity for a physical record, will backspace and read again. If after 20 times it is still failing, it will give up, transmit the contents read with the last trial, and leave the tape positioned after the bad physical record.

III. CERN'S TAPE LABELLING SCHEME

1. INTRODUCTION

At an installation such as CERN, with many thousands of tapes in constant use, it is very important to guard against the accidental erasure of information through a wrong tape being loaded. With a single job processing system as on the IBM 7090 or CDC 3800, the chance of this happening was reduced by having a tape label as well as a reel number for each tape, and the operator checking both of these against the operating form before mounting the tape. Under time-sharing or multiprocessing systems, the operator usually does not know which job requiring tapes is going to be executed next, and the system must give the operator the information which he previously found from the operating form.

This is done at CERN by program-controlled display boxes suspended above each tape unit. These can display a 6-digit reel number, density required (for ½-inch tapes), and the protect status. In principle, these could have incorporated a tape label as well. However, the rate of tape loading and unloading will increase considerably because of the increased throughput of the system (particularly for tape jobs, many of which use very little CP time). The operators cannot, as previously, find and check the tapes before they are needed by the system. Thus the tendency to check only the reel number (since they are stored by reel number) would increase under these circumstances, and errors would probably occur. Since the computer has to tell the operator which tapes to load, a better method is to allow the computer to check that the tape mounted tallies with the one requested. This means having a record on the tape containing the relevant information (one installation is considering the possibility of hardware to "read" numbers punched at the beginning of the tape, but to our knowledge no such mechanism exists at present). Any installation which introduces a scheme using the first record of a tape for this purpose has a major problem to face: what to do about existing tapes containing information still required. To copy all of these probably requires an exorbitant amount of machine time and manpower. Unlabelled tapes from other installations also present difficulties; similarly, the use of its tapes on other machines.

The scheme adopted at CERN overcomes these problems by adding a second loadpoint before the programmers loadpoint, the space between the two being used to hold, and protect, the label. The presence of the two loadpoints does not affect tape loading, as CDC tape transports search forward for the loadpoint when the "load" button is pressed, the tape having been positioned by the operator at the beginning of the leader. To use a magnetic tape with two loadpoints on another computer requires only that the tape be loaded at the second, instead of the first, loadpoint.

2. INFORMATION REQUIRED IN THE LABEL

The CERN tape label is a 100-character record, which contains the following fields:

Field Name	No.BCD Chars.	Example	Comments
Heading	10	CERN LABEL	Invariable.
Reel No.	6	013127	Checked by System.
Sequence No.	4		Order of the reel in a multi-reel file, for eventual reel changing. Not used at present and left blank.
File Name	20	LIBRARY EXPT-70	Checked by System.
Subheading	20	GRIND OUTPUT	For programmer's use.
Programmer's name	10	BARDOT.B	From JØB card, for accounting purposes.
Date label written	6	641231	Year, month, day.
Retention	4	0013	In weeks.
Spare	20		For extensions.

The sequence of operations performed by the System when tapes are loaded is:

i) set the reel number, protect status and density on the display box above the appropriate tape drive;

ii) as each tape drive becomes ready, check protect status and density (½-inch tapes only); if incorrect, unload the tape and reset its number in the display box (once only); if still incorrect abandon the job; otherwise

iii) compare the tape label against the reel number and file name given on the equipment control card and check the file protect status;

iv) if both the reel number and the file name are correct, position the tape at the second loadpoint and clear the display box;

v) if the reel number is incorrect unload the tape and reset its number in the display box;

vi) if the reel number is incorrect twice, the job is abandoned with a message WRONG TAPE(S) LOADED;

vii) if the file name indicates that the tape is not allocated, or the file name is incorrect, but the tape is expired, a new label is written and the tape accepted;

viii) if the file name is incorrect and the tape is unexpired, the job is abandoned with a message.

If the tape is accepted and information is present on the equipment card in the fields subheading and/or retention, then the label is rewritten with the appropriate information added. The subheading field allows programmers to change the use of a tape without changing the file name.

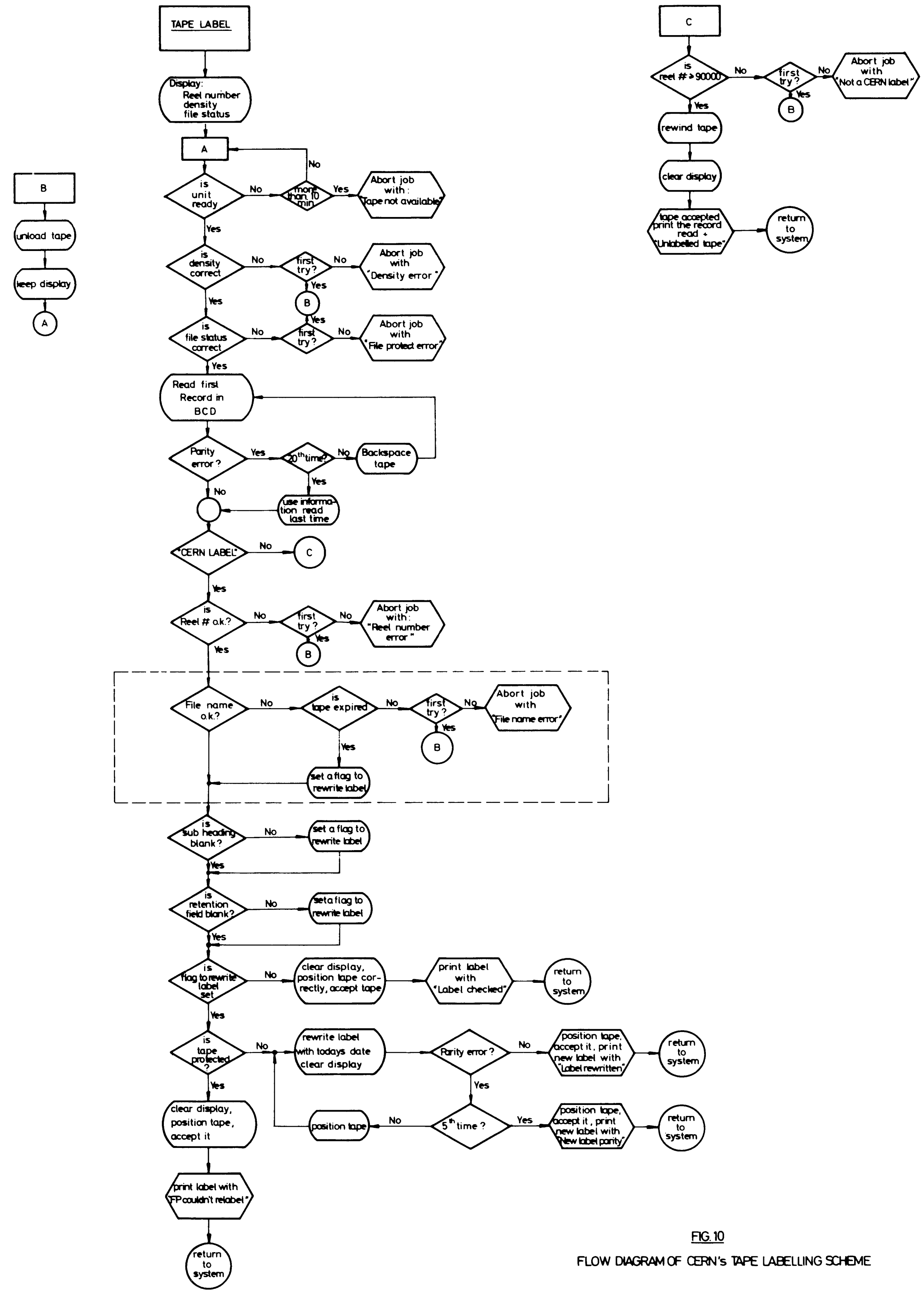
3. FLOW DIAGRAM AND CRITICAL PART OF LABEL CHECKING

The attached flow diagram (Fig. 10) illustrates the execution of that part of the system that checks the tape label. It also contains the error messages printed out by the system. The most critical part from the user's point of view is surrounded by dotted lines. The file name is checked there.

Let us assume Mr. X wants to run his program and write information on his tape. Careful as he is, he renews at that point the retention period for his tape and gives a new subheading. He wants to use tape 3001 L1547, but by error he punches on the equipment card:

$$\text{* TAPE}\quad 5\quad 5001\ \text{L1547}\quad \underbrace{\text{Mr. X HIS TAPE}}_{\text{subheading}}\quad \text{D}\quad \underbrace{12}_{\text{retention period}}$$

The System will display the reel number 5001 above a free tape unit and the operator loads the tape, which might belong to Mr. Y, as specified with a write ring. The System will then find correct protect status and reel number but a different file name, since the file name of Mr. Y's tape 5001 is L1620. If Mr. Y is now careful, the retention period of that tape will not yet expire, and the job will be abandoned with file name error. If, however, the tape has expired, it will be accepted, a new label containing the new file name will be written, the tape will be allocated for the next 12 weeks, and the contents will be destroyed



TAPE LABEL
Display: Reel number density file status
A
is unit ready
No
more than 10 min
Yes
Abort job with: "Tape not available"
is density correct
first try?
Abort job with "Density error"
B
is file status correct
Abort job with "File protect error"
Read first Record in BCD
Parity error?
20th time?
Backspace tape
use information read last time
"CERN LABEL"
C
is Reel # o.k.?
Abort job with: "Reel number error"
File name o.k.?
is tape expired
Abort job with "File name error"
set a flag to rewrite label
is sub heading blank?
is retention field blank?
is flag to rewrite label set
clear display, position tape correctly, accept tape
print label with "Label checked"
return to system
is tape protected?
rewrite label with todays date clear display
Parity error?
position tape, accept it, print new label with "Label rewritten"
5th time?
position tape
position tape, accept it, print new label with "New label parity"
clear display, position tape, accept it
print label with "FP couldn't relabel"
unload tape
keep display
is reel # ≥ 90000
Abort job with "Not a CERN label"
rewind tape
clear display
tape accepted print the record read + "Unlabelled tape"

FIG.10

FLOW DIAGRAM OF CERN's TAPE LABELLING SCHEME if Mr. X's job starts writing. What can Mr. Y do now? First of all he has lost his information on the tape due partly to his own fault. He forgot to check his tape from time to time, and to give a new retention period when the old one had expired. On the other hand, he cannot use his tape any longer, as it is now allocated by Mr. X for 12 weeks. Next time he uses the tape, his job will be abandoned with file name error, but by the subheading he might find out that Mr. X used the tape. These two gentlemen might now come to an agreement on how to use the tape further on. From System's point of view the old label can be re-written in one additional run, using the following small program:

```
*JØB,Y,000000/CØM
*TAPE  5    5001L1547           D 00
*TAPE  5    5001L1620  MR Y     D 15
       PROGRAM RELAB
       CALL EXCHZL(5)
       STØP
       END
78FINISH
```

The System will ask for loading of tape 5001, rewrite the label with a retention period of zero weeks, and the job starts execution. The call to the library routine EXCHZL causes that tape to be unloaded, and the continuation tape specified on the second tape control card (which is physically the same reel) to be loaded. Due to the different file name, the System will check the allocation of the tape, find if expired, and rewrite the tape label with the new file name L1620 and allocate it for 15 weeks.

IV. A PROGRAMMING EXAMPLE FOR EFFICIENT TAPE USAGE ON THE CDC 6600

As outlined by C.R. Symons in the Appendix "Economics of Magnetic Tapes", it is possible to use the storage capacity of a magnetic tape to 76% if one writes logical records of about 512 words in length, and there is not much benefit if one writes logical records of more than 512 words, since in any case they are broken up into physical records of 512 words maximum by the SIPROS system. Many programs are, however, coded in such a way that they write much shorter records onto tape, and thus do not only waste tape storage capacity but also, as the following programming example will show, a considerable amount of central processor time. (See Programming Example I, App. I.)

The intention of this programming example is not to show a very sophisticated and 'fail-safe' way of tape handling, but to prove that it is easy to change already existing programs with the minimum effort, so that they make better use of tape storage and are less time-consuming not only for peripheral processor time but also for central processor time.

The tape used was a 300-ft $1/2$-inch tape. In the first part of the program, the subroutine SUB1, this tape will be written in a continuous loop using a FORTRAN WRITE statement until the End-of-Tape condition is detected by use of the error procedure SETTZL. The total time necessary will be measured with routine TIMEZB, and the total number of words transmitted will be counted. In order to simulate the conditions found out by Symons, the record length will vary between 1 and 400 items. The tape is then rewound and the information is read back. Again the time is measured.

In the second part, now in the main program itself, the same process is repeated but the WRITE/READ statements are replaced by CALL statements to two subroutines, COLLZV and DISTZV. The routine COLLZV contains a 512-word buffer, and will collect the data to be written until the buffer is completely filled and then write it out on tape. The entry ENDZV allows the last buffer to be written on tape and the pointer to be reset for the buffer.

The other routine, DISTZV, contains also a 512-word buffer into which a record is read from tape and distributed to the calling routine, I-words at a time. The entry INITZV initializes the first read and sets the pointer for the buffer.

The effect of this simple, although at first sight lengthy effort, is surprising with respect to the timing:

	Words written	Write time in msec	Read time in msec
READ/WRITE	202305	13560	14220
COLLZV/DISTZV	240645	4920	4740

The gain in speed by using COLLZV/DISTZV is considerable, and due to the fact that 512 words are written/read at once, using the short-list form of a WRITE/READ statement. Also the storage capacity of the tape is increased by one-third. Please note that the length of records written in the first part varied equally between 1 and 400 list items per record. The gain would have been considerably higher if much shorter records had been written (the average binary tape record is 80 words long, as shown by the statistics on the CDC 6600). This test was also carried out, using the same 300-ft, $1/2$-inch tape, on the CDC 3800. There, of course, the length of the buffer in the two routines DISTZV and COLLZV had to be reduced to 255 words, since this is the maximum length of a physical record for FORTRAN binary I/Ø. The test gave a similar gain for tape storage, but a completely different result for the timing:

	Words written	Write time in msec	Read time in msec
READ/WRITE	190951	36283	26952
DISTZV/COLLZV	246351	43588	33795

The most obvious thing is the much faster reading than writing in both cases. This is due to the fact that the 3800 SCOPE system is always reading a physical record in advance, e.g. when the program at the beginning asks for the first record to be read, the second record is read into a system buffer immediately following. In most cases it is therefore already in memory when the program asks for the next record. This set-up is possible due to the fact that the 110 channels on the 3800 can be operated simultaneously with the CPU. The reason for the lower efficiency of the routines COLLZV/DISTZV is twofold:

a) in the SCOPE system there is not much difference between handling of the short-list form of I/Ø and the long-list form, and

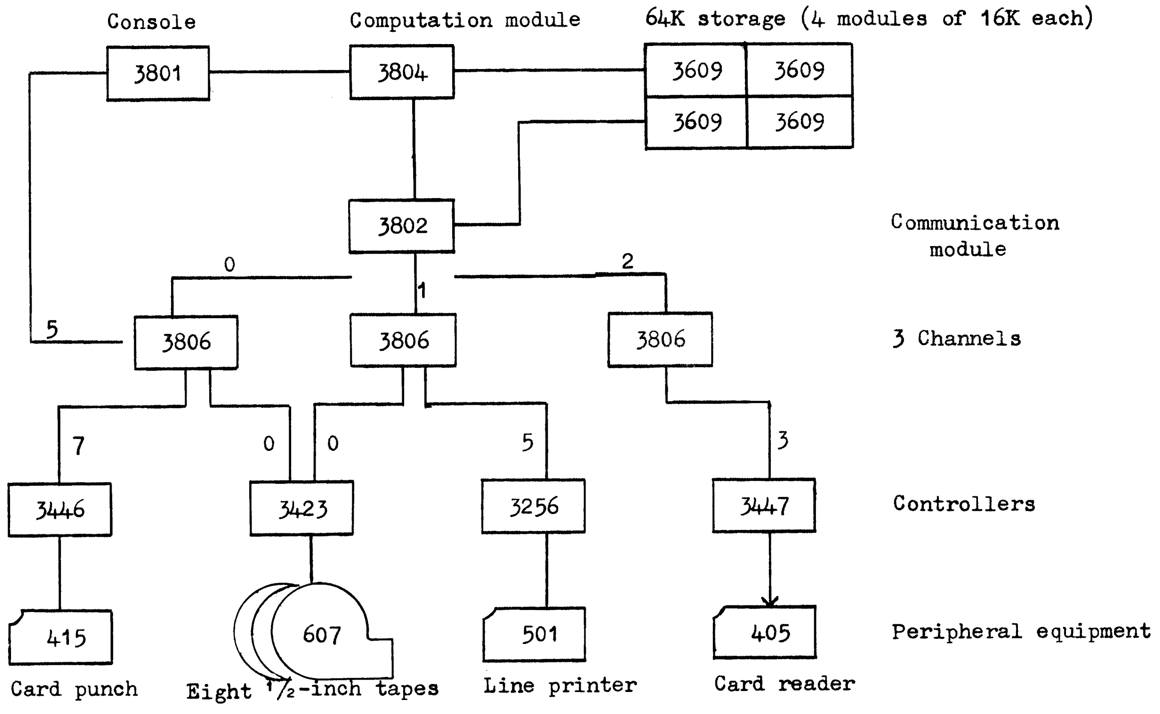
b) the system does a considerable amount of overhead work when a subroutine is called in, before the first statement of that routine is executed.

Please note that a word of the CDC 3800 is 48 bits long, and therefore results in eight frames on tape, whilst a CDC 6600 word is 60 bits long and therefore needs ten frames to be written ($1/2$-inch tape assumed). At the end of the next chapter, dealing with BUFFER IN and BUFFER OUT statements on the CDC 3800, there will be a programming example, which also allows the speeding-up of data transfer from and to tape on the CDC 3800.

V. BUFFER IN AND BUFFER OUT STATEMENTS ON THE CDC 3800

1. LAYOUT OF CDC 3800 INSTALLATION AT CERN

CERN has installed the following configuration of a CDC 3800:



2. <u>HOW TO USE GIVEN HARDWARE FACILITIES IN FORTRAN</u>

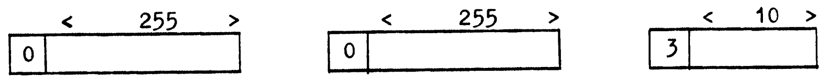
The part that interests us is that of the eight ½-inch tapes. They are controlled by a tape controller having two independent read/write controls and can control up to eight magnetic tapes (½-inch). This controller is hooked up onto two channels, channels 0 and 1, which can be operated simultaneously with each other and with the computational module. To each of these channels is connected other peripheral equipment, namely the card-punch controller to channel 0 and the line printer to channel 1. If neither of these two is operating, up to two magnetic tapes can be used for read/write operations at the same time. If the card punch is going, channel 0 is locked out for tape operation, and vice versa. The same is true for the line printer and channel 1. How can this hardware possibility now be used in order to speed up FORTRAN programs with tape handling. Let us first consider programs using READ/WRITE statements and look at the format of a binary tape.

The maximum length of a physical record, written with a FORTRAN WRITE request, is 256 words, 48 bits each, where the first word is a leader word generated by the system and the following 1 to 255 words may contain the information generated by the program. A logical record of more than 255 words will therefore be broken up into as many physical records as necessary.

Example:

```
WRITE (5) (A(I), I = 1,520)
```

will result in three physical records with the following layout:



which means that the last physical within a logical record always contains in the header an integer number that tells the system how many physical records belong to that logical record; all the other physical records contain a 0 in the header. (Note: If a logical record is equal or less than a physical record in size, the header will be 1.) In case of several write requests following each other, e.g.

WRITE (5) A,B,C
WRITE (6) D,E,F

the control will not be returned back to the program before each write request has been completed, including the tape operation. Therefore, there is no gain from that two-channel concept.

In case of reading, the system will take the following action:

When the first read request is issued from a program for a tape, the system will fulfil this request, and before returning control to the next following statement within that program the system will initialize the read of the next following physical record on that tape into a systems buffer but not wait for completion and return control. If the program now asks for that record when next it executes a read statement to that tape unit, the 1st physical record is already in core and can be transmitted directly without tape operation. In other words, the system is always reading one physical record in advance. In this way, as seen from the following examples, a considerable gain in time is possible. When now two successive reads to two different units are issued, it is theoretically possible that they will overlap for some time, depending on the housekeeping done.

Generally speaking, it is true that there is no way for a pure FORTRAN program to utilize a multiple channel concept on a machine unless new statements, as for the CDC 3800 SCOPE system, are invented. These are:

BUFFER IN (i,p) (A,B)
BUFFER ØUT (i,p) (A,B)
IF (UNIT, i) u_1, u_2, u_3, u_4
K = LENGTHF (i)

and, especially for the system in use at CERN, the

CALL EØSTAT (i, IFLAG)

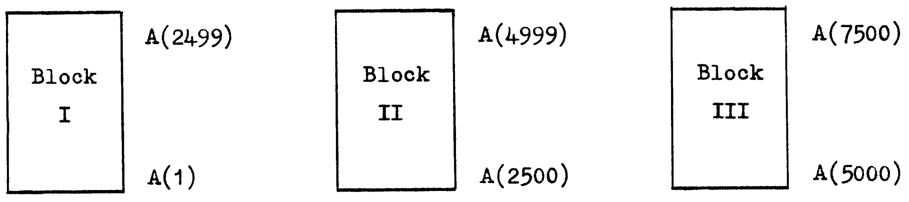
They are explained in full detail in the 3400/3600/3800 FORTRAN manual, page 10-6 and following pages, and the CERN 3800 manual, page $\boxed{4.3}$ 2.

The BUFFER IN/ØUT statement initiates a direct data transfer from/to the peripheral medium to/from the programmers own data area in core storage. There is <u>no</u> internal data transfer between systems and programmers area as for the FORTRAN READ/WRITE statement, before the data are transmitted via a data channel to the peripheral medium. As soon as the data transfer is initiated, control is given back to the program while the data transfer is still in progress. This allows the programmer to perform other calculations which do not need a data transfer and do not use the data currently in progress. The IF (UNIT, i) statement now provides a means of telling the program whether or not a data transfer has been successfully terminated.

On the other hand, as the data transfer is done directly from programmers area to peripheral medium, the physical record size of records on tape is exactly equal to the length of the specified buffer area (A,B). They are therefore non-standard records and can only be read via the BUFFER IN statement. As experience shows, this is unfortunately the wider application of these statements. Of course, the advantage of writing gapless records of maximum core size allows very economical tape usage, and as long as people are aware of associated problems we will have nothing to say against this. But this is not the only advantage they offer. The main intention, as outlined before, was to provide the programmer with an instrument by which he could now code himself in FORTRAN operations (which were previously a domain of certain people capable of coding in machine language) in order to make his program more efficient by overlapping, as far as possible, input, output, and calculations.

3. A TYPICAL EXAMPLE FOR USE OF BUFFER IN/ØUT

Let us assume that there is Mr. ABC who has to process data on tape, record by record, with almost the same calculation procedure and who has then to write out the results onto a second tape. (I should think this is also a valid assumption for CERN.) He now can code the classical way, use FORTRAN READ or WRITE statements to perform the I/Ø transfers, or he might use BUFFER IN/ØUT statements, the more adequate way for the 3800 to perform his task. Such an application has now been coded, once using READ/WRITE statements and the other time using BUFFER IN/ØUT statements. The actual computation carried out was a matrix inversion, and the order of the inversion was varied to find a maximum, so that the time for total data transfer for BUFFER IN/ØUT was just about the time for computation. The program first prepares 7500 random numbers and writes them in logical blocks of 2500 words each onto tape, 15 records totally. As the known maximum length of the input data is now 2500 words per record, a total data area of 7500 words, array A, is used in core to handle the data in the following way:



Pointers are kept so that these three buffers can be used circularly with the following steps:

1) fill block I with first set of data from tape 5;

2) fill block II from tape 5 and start calculation in block I;

3) empty block I on tape 6, fill block III from tape 5 and calculate in block II;

4) continue the same process circular as in (3) until input data are exhausted (next step would have been: fill I, empty II and perform calculations in III);

5) continue calculation and writing, until all records are handled.

The coding of this concept is given in Programming Examples II and III, App. II and III. The measured times are:

Order of matrix	Time for READ/WRITE	Time for BUFFER IN/ØUT
10	10.52 sec	5.15 sec
12	10.91 sec	4.38 sec
14	11.61 sec	4.38 sec
16	12.48 sec	4.38 sec
18	13.57 sec	4.98 sec
20	14.92 sec	6.33 sec

The maximum achieved is a ratio of ≈ 3:1 for matrices of order 16 and 18. This is an optimum, where the calculation time is just about the data transfer time. If the calculation is governing, gain will be less in per cent. If the calculation time is less, then most of the time is spent in idling on an IF (UNIT,i) statement, to check for completion of the buffer operation. (See Programming Examples II and III, App. II and III.)

4. COLLZV AND DISTZV USING BUFFER IN/ØUT

In Chapter IV it was mentioned that the concept of the two routines, COLLZV and DISTZV, did give some gain in the storage capacity of a tape on the CDC 3800, but the time used was considerably larger than with normal READ/WRITE statements. In the following programming example it is now being attempted, although not yet completely checked out, to use the buffer statements in order to speed up these routines. It is obvious that a simple exchange of the READ/WRITE statements with the BUFFER IN/ØUT statements would not gain very much: the program would always have to wait for completion of the buffered operation. As a result of this, a so-called "double buffer system" was used:

COLLZV: while collecting data into the first buffer, the second buffer is written onto tape.

DISTZV: while distributing data from the first buffer, the second buffer is filled from tape.

The same test as that described in Chapter IV was carried out for these new routines. The old results and the new ones are given in the following table:

	Words written	Write time in msec	Read time in msec
READ/WRITE	190951	36283	26952
old COLLZV/DISTZV	246351	43588	33795
new COLLZV/DISTZV	242970	24333	24005

This shows that the basic concept of collecting data into a buffer of adequate size before writing them on a peripheral medium and reading them into a buffer before distributing them into the appropriate memory locations, pays not only in storage capacity of the tape but also in speed obtained. It is, of course, necessary to use the adequate forms: the short-list form of the WRITE/READ statement in the CDC 6600 SIPROS system, and the double buffer concept, together with BUFFER IN and BUFFER ØUT statements in the CDC 3800 SCOPE system. (See Programming Example IV, App. IV.)

In order to take care of various I/Ø error conditions, the two routines COLLZV and DISTZV became larger and more complex. They have been used in the way explained in Chapter IV in order to find their effectiveness. However, they have never been checked out completely for all error conditions. It is therefore not advisable to use them as they stand, without checking.

Table 1

Specifications for the 607 and 626 tape transports

	626	607		
Tape size	1 in.	112 in.	1/2 in.	1/2 in.
Density	800 bpi	200 bpi	556 bpi	800 bpi
Tracks	14	7	7	7
Tape speed for READ/WRITE	150 in./sec	150 in./sec	150 in./sec	150 in./sec
Tape speed for rewind and unload	over 320 in./sec	over 350 in./sec	over 350 in./sec	over 350 in./sec
Start time	2.5 msec	3 msec	3 msec	3 msec
Stop time	2.25 msec	2 msec	2 msec	2 msec
Inter-record gap	1 in.	3/4 in.	3/4 in.	3/4 in.
Data bits/frame	12	6	6	6
110 rate/12 bit	8.3 μsec	66.7 μsec	24 μsec	16.7 μsec
Tape per reel	2400 ft (1200 ft)	2400 ft (1200 ft, 300 ft)	2400 ft (1200 ft, 300 ft)	2400 ft (1200 ft, 300 ft)
End of file gap	6 in.	6 in.	6 in.	6 in.

Table 2

Function and status codes for the 6622 and 607-B tape controllers (synchronizers) used.

	Type of tape controller (synchronizer)	
Functions	6622 (1 in. tapes)	607-B ($^1/_2$ in. tapes)
Select	300U	200U
Write Binary	301U	201U
Read Binary	302U	202U
Backspace	303U	203U
Rewind	306U	206U
Rewind Unload	307U	207U
Request status	310U	210U
Write BCD	-	221U
Read BCD	-	222U
Write File Mark	361U	261U

Status request	6622 (1 in. tapes)	607-B ($^1/_2$ in. tapes)
Ready	XXX0	XDX0
Not Ready	XXX1	XDX1
Parity Error	XXX2	XDX2
Load Point	XXX4	XDX4
End of Tape	XX1X	XD1X
File Mark	XX2X	XD2X
Write Lockout	XX4X	XD4X

U = unit number of a specific tape transport (0 - 7)

D = density D = 0, 800 bpi
D = 1, 556 bpi
D = 2, 200 bpi

X means unused, no value to be assumed

```
IOSPED ************** DATE 10/10/67 ******** VERSION 046 ******** PAGE      1

      PROGRAM IOSPED
      COMMON /TAPEIN/ JTAPE
      COMMON /TAPOUT/ ITAPE
      DIMENSION IA(400),IB(400)
      COMMON IEOT
      EXTERNAL IEOT1
      CALL SETTZL(435,IEOT1,438,IEOT1)
      CALL SUB1
      JTAPE=5
      ITAPE=5
      REWIND 5
      DO 1 I=1,400
    1 IA(I)=I
      CALL TIMEZB(ITIME1)
      IEOT=0
      ICOUNT=0
    2 CONTINUE
      DO 3 I=1,400
      CALL COLLZV(IA,I)
      ICOUNT=ICOUNT+I
      IF(IEOT) 4,3,4
    3 CONTINUE
      GO TO 2
    4 CALL TIMEZB(ITIME2)
      REWIND 5
      READ(5) C
      BACKSPACE 5
      CALL TIMEZB(ITIME3)
      IEOT=0
      CALL INITZV(IB,1)
      IC=0
    5 CONTINUE
      DO 9 I=1,400
      CALL DISTZV(IB,I)
      IC=IC+I
      IF(IEOT) 10,9,10
    9 CONTINUE
      GO TO 5
   10 CALL TIMEZB(ITIME4)
      REWIND 5
      READ(5) C
      BACKSPACE 5
      ITIME1=(ITIME2-ITIME1)*60
      ITIME3=(ITIME4-ITIME3)*60
      PRINT 100,ITIME1 ,ICOUNT,ITIME3,IC
      STOP
  100 FORMAT(1H1,'ALL READ/WRITE OPERATIONS ARE PERFORMED IN      BUFFERE
     1D MODE ',//1H0,'WITHIN ',I10,' MSEC IT WROTE ',I10,' WORDS BEFORE
     2REACHING EOT ',//1H0,'WITHIN ',I10,' MSEC IT READ BACK',I10,' WORD
     3S' )
      END
```

```
SUB1 ************** DATE 10/10/67 ******** VERSION 046 ******** PAGE      1
      SUBROUTINE SUB1
      DIMENSION IA(400),IB(400)
      COMMON IEOT
      EXTERNAL IEOT1
      CALL SETTZL(435,IEOT1,438,IEOT1)
      REWIND 5
      DO 1 I=1,400
    1 IA(I)=I
       CALL TIMEZB(ITIME1)
      IEOT=0
      ICOUNT=0
    2 CONTINUE
      DO 3 I=1,400
      WRITE(5)(IA(J),J=1,I)
      ICOUNT=ICOUNT+I
      IF(IEOT) 4,3,4
    3 CONTINUE
      GO TO 2
    4 CALL TIMEZB(ITIME2)
      REWIND 5
      READ(5) C
      BACKSPACE 5
      CALL TIMEZB(ITIME3)
      IEOT=0
      IC=0
    5 CONTINUE
      DO 9 I=1,400
      READ(5)(IB(J),J=1,I)
      IC=IC+I
      IF(IEOT) 10,9,10
    9 CONTINUE
      GO TO 5
   10 CALL TIMEZB(ITIME4)
      ITIME1=(ITIME2-ITIME1)*60
      ITIME3=(ITIME4-ITIME3)*60
      PRINT 100,ITIME1 ,ICOUNT,ITIME3,IC
      RETURN
  100 FORMAT(1H1,'ALL READ/WRITE OPERATIONS ARE PERFORMED IN NON BUFFERE
     1D MODE ',//1H0,'WITHIN ',I10,' MSEC IT WROTE ',I10,' WORDS BEFORE
     2REACHING EOT ',//1H0,'WITHIN ',I10,' MSEC IT READ BACK',I10,' WORD
     3S' )
      END
    COLLZV ************** DATE 23/05/67 ********* VERSION 041 ******** PAGE       1
      SUBROUTINE COLLZV(A,I)
C     THIS ROUTINE WILL COLLECT WORDS FROM ARRAY A, I WORDS AT A TIME
C     IT WILL PUT THEM INTO A BUFFER OF 512 WORDS AND WRITE ONTO TAPE
      COMMON IEOT
      COMMON / TAPOUT/ITAPE
      DIMENSION A(I),BUFFER(512)
      DATA(IPOINT=0)
      I1=I
      DO 2 J=1,I1
      IPOINT=IPOINT+1
      BUFFER(IPOINT)=A(J)
      IF(IPOINT-512) 2,1,2
    1 WRITE(ITAPE) BUFFER
      IPOINT=0
C     DID WE REACH END OF TAPE
      IF(IEOT) 4,2,4
C     SET IEOT EQUAL TO LAST ELEMENT EMPTIED
    4 IEOT=J
      RETURN
    2 CONTINUE
      RETURN
      ENTRY ENDZV
      IPOINT=0
      WRITE (ITAPE) BUFFER
      IF(IEOT.EQ.1) IEOT=-1
      RETURN
      END
    DISTZV ************** DATE 23/05/67 ********* VERSION 041 ********* PAGE       1

         SUBROUTINE DISTZV(A,I)
C        THIS ROUTINE WILL READ A RECORD OF 512 WORDS FROM TAPE AND DISTRI-
C        BUTE IT ON CALL INTO ARRAY A, I WORDS AT A TIME.
C        AN INITIALISATION IS NECESSARY WHENEVER A NEW TAPE IS STARTED, OR
C        WHENEVER A BACKSPACE OR REWIND HAS BEEN DONE ELSEWHERE TO THAT
C        TAPE.
         COMMON IEOT
         COMMON /TAPEIN/ JTAPE
         DIMENSION A(I),BUFFER(512)
       1 I1=I
         DO 2 J=1,I1
         JPOINT=JPOINT+1
         A(J)=BUFFER(JPOINT)
         IF(JPOINT-512)2,3,2
       3 JPOINT=
         READ(JTAPE) BUFFER
C        DID WE REACH END OF TAPE
         IF (IEOT) 4,2,4
C        SET IEOT EQUAL TO LAST ELEMENT FILLED
       4 IEOT=J
         RETURN
       2 CONTINUE
         RETURN
C        INITIALIZATION ROUTINE
         ENTRY INITZV
         JPOINT=
         READ(JTAPE) BUFFER
         RETURN
         END
```

```
 SUBROUTINE IEOT1
 COMMON IEOT
 IEOT=1
 RETURN
 END
```

PROGRAMMING EXAMPLE II

```
FTN 5.3                                                                  05/25/67

          PROGRAM RDWRIT
          DIMENSION A(7500),INDEX(6),INDEX1(50,2)
    C
    C PREPARE DATA IN CM
          DO 1 I=1,7500
        1 A(I)=RANF(-1)
    C
    C SET INDEX(J) TO LIMITS OF EACH DATA BLOCK OF 2500 WORDS
          I=0
          DO 2 J=1,6,2
          I=I+1
          INDEX(J)=I
          I=I+2499
        2 INDEX(J+1)=I
    C
    C TRANSFER DATA FROM CM TO TAPE 5
          DO 3 I=1,5
          DO 3 IP1=1,6,2
          K1=INDEX(IP1)
          K2=INDEX(IP1+1)
          WRITE(5) (A(J),J=K1,K2)
        3 CONTINUE
          ENDFILE 5
          REWIND 5
          REWIND 6
    C
    C START PROCESSING
          DO 1000 NS=10,20,2
          READ(5), IP1
          BACKSPACE 5
          CALL TIMEZZ(IP4)
          IP1=-1
          IP2=-3
          IP3=-5
          ICOUNT=1
        5 IP1=IP1+2
          IP2=IP2+2
          IP3=IP3+2
          IF (IP1.GT.5) IP1=1
          IF (IP2.GT.5) IP2=1
          IF (IP3.GT.5) IP3=1
          IF (IP1.LT.0) GO TO 8
          L1=INDEX(IP1)
          L2=INDEX(IP1+1)
    C
    C TRANSFER NEXT BLOCK OF DATA FROM TAPE 5 TO CM
          READ(5) (A(I),I=L1,L2)
        8 IF (IP3.LT.0) GO TO 11
          ICOUNT=ICOUNT+1
          K1=INDEX(IP3)
          K2=INDEX(IP3+1)

    C WRITE RESULTS OF LAST CALCULATION ON TAPE 6
          WRITE(6) (A(J),J=K1,K2)
       11 IF (ICOUNT.EQ.0) GO TO 30
          IF (IP2.LT.0) GO TO 13
```

```
FTN 5.3                                                                   05/25/67
          K1=INDEX(IP2)
      C
      C PROCESS DATA
          CALL MATIN1(A(K1),50,NS,50,0,INDEX1,NERROR,DETERM)
       13 IF(EOF,5) 18,17
       17 IF(IOCHECK,5) 91,5
       18 ICOUNT=-1
          IP3=IP3+2
          GO TO 8
       30 CALL TIMEZZ(IP5)
          TIME=IP5=IP4
          TIME=TIME/1000.
          PRINT 101,NS,TIME
          REWIND 5
          REWIND 6
     1000 CONTINUE
          STOP
       91 WRITE(61,103)
          STOP
      101 FORMAT(* MATRIX OF ORDER*,I3,*            TIME TAKEN  SEC*,E10.3)
      103 FORMAT(*PARITY ON READ *)
          END
```

PROGRAMMING EXAMPLE III

```
FTN 5.3                                                               05/25/67

          PROGRAM BUFFER
          DIMENSION A(7500),INDEX(6),INDEX1(50,2)
C
C PREPARE DATA IN CM
          DO 1 I=1,7500
        1 A(I)=RANF(-1)
C
C SET INDEX(J) TO LIMITS OF EACH DATA BLOCK OF 2500 WORDS
          I=0
          DO 2 J=1,6,2
          I=I+1
          INDEX(J)=I
          I=I+2499
        2 INDEX(J+1)=I
C
C TRANSFER DATA FROM CM TO TAPE 5
          DO 3 I=1,5
          DO 3 IP1=1,6,2
          K1=INDEX(IP1)
          K2=INDEX(IP1+1)
          BUFFER OUT (5,1) (A(K1),A(K2))
       23 IF (UNIT,5) 23,3
        3 CONTINUE
          ENDFILE 5
          REWIND 5
          REWIND 6
C
C START PROCESSING
          DO 1000 NS=10,20,2
          READ(5), IP1
          BACKSPACE 5
          CALL TIMEZZ(IP4)
          IP1=-1
          IP2=-3
          IP3=-5
          ICOUNT=1
        5 IP1=IP1+2
          IP2=IP2+2
          IP3=IP3+2
          IF (IP1.GT.5) IP1=1
          IF (IP2.GT.5) IP2=1
          IF (IP3.GT.5) IP3=1
          IF (IP1.LT.0) GO TO 8
          L1=INDEX(IP1)
          L2=INDEX(IP1+1)
C
C TRANSFER NEXT BLOCK OF DATA FROM TAPE 5 TO CM
          BUFFER IN (5,1) (A(L1),A(L2))
        8 IF (IP3.LT.0) GO TO 11
          ICOUNT=ICOUNT+1
          K1=INDEX(IP3)
          K2=INDEX(IP3+1)
C
C WRITE RESULTS OF LAST CALCULATION ON TAPE 6
          BUFFER OUT (6,1) (A(K1),A(K2))
       11 IF (ICOUNT.EQ.0) GO TO 30
```

```
FTN 5.3                                                                   05/25/6/
              IF (IP2.LT.0) GO TO 13
              K1=INDEX(IP2)
        C
        C PROCESS DATA
              CALL MATIN1(A(K1),50,NS,50,0,INDEX1,NERROR,DETERM)
           13 IF (UNIT,5) 13,15,14,90
           14 ICOUNT=-1
              IP2==5
              IP1==5
           15 IF (UNIT,6) 15,5
           30 CALL TIMEZZ(IP5)
              TIME=IP5=IP4
              TIME=TIME/1000.
              PRINT 101,NS,TIME
              REWIND 5
              REWIND 6
         1000 CONTINUE
              STOP
           90 WRITE(61,102)
              STOP
          101 FORMAT(* MATRIX OF ORDER*,I3,*            TIME TAKEN  SEC*,F10.3)
          102 FORMAT(* PARITY ON BUFFER IN *)
              END
```

PROGRAMMING EXAMPLE IV

```
FTN 5.3                                                                05/25/67

            SUBROUTINE COLLZV(A,I)
C           THIS ROUTINE WILL COLLECT DATA FROM ARRAY A, I WORDS AT A TIME,
C           INTO A DOUBLE BUFFER SYSTEM AND WRITE THEM OUT ON TAPE AS THE BUF
C           FER GETS FILLED.
C
C           THE MEANING OF THE FLAG WORDS IN BLOCK /COL1ZV/ IS
C
C           JERR = 0  NO ERROR     IND = 1  BUFOU1+2 , = 3 BUFOU1 ONLY
C           JERR = 1  END OF TAPE  IND = 2  BUFOU2+1 , = 4 BUFOU2 ONLY
            DIMENSION A(I)
            INTEGER OUTAP
            COMMON/COL1ZV/JERR,OUTAP,JEMPT,IND
            COMMON /COL2ZV/ JPOINT,BUFOU1(255),BUFOU2(255)
            I1=I
            JEMPT=0
            L=1
            GO TO JUMP1,(5,11,14)
C           BUFFER TWO HAS BEEN FILLED. SWITCH TO BUFFER ONE.
          1 L=JEMPT+1
            JPOINT=0
C            CHECK THE PREVIOUS BUFFER OUT OPERATION BEFORE ISSUING THE NEXT
          2 IND=1
            IF(UNIT,OUTAP) 2,3
          3 CALL EOSTAT(OUTAP,JERR)
C           DID WE REACH END OF TAPE
            IF(JERR) 13,4,13
C           ISSUE  BUFFER OUT FOR BUFFER TWO
          4 BUFFER OUT (OUTAP,1) (BUFOU2(1),BUFOU2(255))
            ASSIGN 5 TO JUMP1
C           COLLECT DATA FROM ARRAY A INTO BUFFER ONE.
          5 DO 6 JEMPT=L,I1
            JPOINT=JPOINT+1
            BUFOU1(JPOINT)=A(JEMPT)
            IF (JPOINT-255) 6,7,6
          6 CONTINUE
            RETURN
C           BUFFER ONE HAS BEEN FILLED. SWITCH TO BUFFER TWO
          7 L=JEMPT+1
            JPOINT=0
C           CHECK PREVIOUS BUFFER OUT OPERATION BEFORE ISSUING NEXT.
          8 IND=2
            IF (UNIT,OUTAP) 8,9
          9 CALL EOSTAT (OUTAP,JERR)
C           DID WE REACH END OF TAPE
            IF(JERR) 13,10,13
C           ISSUE BUFFER OUT FOR BUFFER ONE
         10 BUFFER OUT (OUTAP,1) (BUFOU1(1),BUFOU1(255))
            ASSIGN 11 TO JUMP1
C           COLLECT DATA FROM ARRAY A INTO BUFFER TWO.
         11 DO 12 JEMPT=L,I1
            JPOINT=JPOINT+1
            BUFOU2(JPOINT)=A(JEMPT)
C           IS BUFFER FILLED
            IF(JPOINT-255) 12,1,12
         12 CONTINUE
            RETURN
```

```
C
C     ERROR HANDLING
C
C     END CF TAPE SENSED
   13 JERR=1
      ASSIGN 14 TO JUMP1
      BACKSPACE OUTAP
      ENDFILE OUTAP
      RETURN
C
C     EOT ERROR HAS NOT BEEN HANDLED BY CALLING ROUTINE. TERMINATE
C
   14 PRINT 100,OUTAP
      STOP
  100 FORMAT(*1NO CARE WAS TAKEN OF THE END OF TAPE CONDITION INDICATED
     1BY -COLLZV- FOR UNIT *,I4,* RUN TERMINATED.*)
C
C     INITIATE POINTERS ON FIRST CALL
C
      ENTRY INOUZV
      ASSIGN 5 TO JUMP1
      JPOINT=0
      IERR=0
      RETURN
C
C     WRITE LAST BUFFER ON TAPE, PUT AN END OF FILE AND RETURN
C
      ENTRY ENDZV
      IND==IND+3
      ASSIGN 5 TO JUMP1
      JPOINT=0
C     CHECK LAST BUFFER OPERATION
   15 IF(UNIT,OUTAP) 15,16
   16 CALL EOSTAT(OUTAP,JERR)
C     DID WE REACH END OF TAPE
      IF(JERR) 13,17,13
C     WHICH IS THE LAST BUFFER
   17 GO TC (19,18),IND
C     OUTPUT FROM ONE
   18 BUFFER OUT (OUTAP,1) (BUFOU1(1),BUFOU1(255))
      IND = 3
      GO TC 20
C     OUTPUT FROM TWO
   19 BUFFER OUT (OUTAP,1) (BUFOU2(1),BUFOU2(255))
      IND=4
C     CHECK THE LAST BUFFER OPERATION IS COMPLETED
   20 IF(UNIT,OUTAP) 20,21
   21 CALL EOSTAT(OUTAP,JERR)
C     DID WE REACH END OF TAPE
      IF(JERR) 13,22,13
   22 ENDFILE OUTAP
      RETURN
C
C     HANDLING OF EOT CASE, ASSUMING A NEW TAPE HAS BEEN MOUNTED
C
      ENTRY EOTZV
```

```
C      ARE THERE TWO BUFFERS LEFT, OR JUST ONE.
       GO TO (30,33,30,33),IND
   30  BUFFER OUT (OUTAP,1) (BUFOU1(1),BUFOU1(255))
   31  IF(UNIT,OUTAP) 31,320
  320  CALL EOSTAT (OUTAP,JERR)
C      AGAIN EOT, TERMINATE
       IF(JERR) 14,32,14
   32  GO TO (33,37,37,37),IND
   33  BUFFER OUT (OUTAP,1) (BUFOU2(1),BUFOU2(255))
   34  IF (UNIT,OUTAP) 34,35
   35  CALL EOSTAT (OUTAP,JERR)
       IF (JERR) 14,36,14
   36  GO TO (37,30,37,37),IND
   37  ASSIGN 5 TO JUMP1
       JPOUNT=0
       RETURN
       END
```

```
FTN 5.3                                                       05/25/67

      SUBROUTINE DISTZV(A,I)
C     THIS ROUTINE WILL READ DATA INTO A DOUBLE BUFFER SYSTEM AND DISTR≠
C     BUTE THEM ON CALL INTO ARRAY A, I WORDS AT A TIME
      DIMENSION A(I)
      COMMON /DIS1ZV/ IERR,INTAP,LENGTH,JFILL,IND
      COMMON /DIS2ZV/JPOINT,BUFIN1(255),BUFIN2(255)
      I1=I
      JFILL=0
      L=1
      GO TO JUMP1,(2,5,11,17)
C     BUFFER TWO HAS BEEN EMPTIED.SWITCH TO BUFFER ONE.
    1 L=JFILL+1
      JPOINT=0
C     CHECK PREVIOUS BUFFER IN OPERATION BEFORE ISSUING THE NEXT
    2 IND=1
      IF(UNIT,INTAP) 2,3,14,15
    3 CALL EOSTAT (INTAP,IERR)
C     DID WE REACH END OF TAPE
      IF (IERR) 13,4,13
    4 LENGTH=LENGTHF(INTAP)
      IF(LENGTH.NE.255) GO TO 16
C     ISSUE BUFFER IN FOR BUFFER TWO
      BUFFER IN (INTAP,1) (BUFIN2(1),BUFIN2(255))
      ASSIGN 5 TO JUMP1
C     DISTRIBUTE DATA FROM BUFFER ONE INTO ARRAY A.
    5 DO 6 JFILL=L,I1
      JPOINT=JPOINT+1
      A(JFILL)=BUFIN1(JPOINT)
C     IS BUFFER EXHAUSTED
      IF (JPOINT-255) 6,7,6
    6 CONTINUE
      RETURN
C     BUFFER ONE HAS BEEN EMPTIED.SWITCH TO BUFFER TWO.
    7 L=JFILL+1
      JPOINT=0
C     CHECK PREVIOUS BUFFER IN OPERATION BEFORE ISSUING THE NEXT
    8 IND=2
      IF(UNIT,INTAP)  8,9,14,15
    9 CALL EOSTAT (INTAP,IERR)
C     DID WE REACH END OF TAPE
      IF (IERR) 13,10,13
   10 LENGTH=LENGTHF(INTAP)
      IF(LENGTH.NE.255) GO TO 16
C     ISSUE BUFFER IN FOR BUFFER ONE.
      BUFFER IN (INTAP,1) (BUFIN1(1),BUFIN1(255))
      ASSIGN 11 TO JUMP1
C     DISTRIBUTE DATA FROM BUFFER TWO INTO ARRAY A.
   11 DO 12 JFILL=L,I1
      JPOINT=JPOINT+1
      A(JFILL)=BUFIN2(JPOINT)
C     IS BUFFER EXHAUSTED
      IF (JPOINT-255) 12,1,12
   12 CONTINUE
      RETURN
C
C     ERROR HANDLING
```

```
C
C     END OF TAPE SENSED.
   13 IERR=1
      JUMP=1
      ASSIGN 17 TO JUMP1
      RETURN
C     END OF FILE SENSED
   14 IERR=2
      JUMP=2
      ASSIGN 17 TO JUMP1
      RETURN
C     READ PARITY ERROR
   15 IERR=3
      JUMP=3
      ASSIGN 17 TO JUMP1
      RETURN
C     READ LENGTH ERROR
   16 IERR=4
      JUMP=4
      ASSIGN 17 TO JUMP1
      LENGT=LENGTH
      RETURN
C     LAST INDICATED ERROR HAS NOT BEEN HANDLED BY CALLING ROUTINE.
C     TERMINATE JOB WITH MESSAGE.
   17 GO TO (18,19,20,21),JUMP
   18 PRINT 100,INTAP
      STOP
   19 PRINT 101,INTAP
      STOP
   20 PRINT 102,INTAP
      STOP
   21 PRINT 103,INTAP,LENGT
      STOP
C
C     INITIATE FIRST READ AND SET POINTERS. (AT BEGIN AND AFTER ERROR)
C
      ENTRY ININZV
      JPOINT=0
      ASSIGN 2 TO JUMP1
      BUFFER IN (INTAP,1) (BUFIN1(1),BUFIN1(255))
      IERR=0
      RETURN
C
C     ERROR MESSAGES
C
  100 FORMAT (*1NO CARE WAS TAKEN OF THE END OF TAPE CONDITION INDICATED
     1 BY =DISTZV- FOR UNIT *,I4,* RUN TERMINATED.* )
  101 FORMAT (*1NO CARE WAS TAKEN OF THE END OF FILE CONDITION INDICATED
     1 BY =DISTZV- FOR UNIT *,I4,* RUN TERMINATED.* )
  102 FORMAT (*1NO CARE WAS TAKEN OF THE READ PARITY CONDITION INDICATED
     1 BY =DISTZV- FOR UNIT *,I4,* RUN TERMINATED.* )
  103 FORMAT (*1NO CARE WAS TAKEN OF THE READ LENGTH ERROR INDICATED BY
     1-DISTZV- FOR UNIT *,I4,/* THE LENGTH WAS *,I10,*.RUN TERMINATED.*)
      END
```

FORTRAN COMPILERS

by

J. Garratt

CONTENTS

1. INTRODUCTION

Before dealing with construction of FORTRAN programs and methods of reducing the time taken by the executing program, it is worth talking a little about the actual process of compilation, i.e. the translation of FORTRAN statements into a form acceptable by a particular machine.

Compilers vary considerably in sophistication from simple statement translators to multi-pass systems which scan the source statements (or their internal equivalent) again and again, determined to squeeze all redundant instructions from the final output object program.

The more powerful the compiler the less the user has to worry about optimization at the source level, but there is always an area which the compiler can never optimize as it does not know the <u>purpose</u> of the source program, merely its statement-by-statement structure.

At CERN, we have been dealing with simple compilers on the CDC 6600/6400. Both the SIPROS compiler and the SCOPE compiler are essentially one-pass compilers which translate statements as they are encountered, and neither goes into a complicated scan of, for example, DO-loops. Simple compilers normally have the advantage of fast compilation speeds.

In Appendix I, a very broad outline of a compiler is shown. This is a simplified flow chart but gives the main components, i.e. input of source statements, analysis of declarations, breaking down arithmetic expressions, output to assembler of symbolic machine code (not always necessary, though many compilers do this), final output to loader (and/or card punch) of relocatable machine instructions.

2. PROGRAM PLANNING AND CONSTRUCTION

It is very difficult to lay down hard and fast rules for program construction, but it is possible to give some general pointers. The first questions the programmer should ask himself are:

- Does this program have to be debugged quickly and then after several runs be of no further use?
- Is it a program which will run for some time, using a considerable amount of machine-time during its lifetime?

In other words, is a quick answer the desired goal, or a slick, sophisticated routine which will run reasonably efficiently for a long period? Obviously the criteria governing the construction of the former are different from those governing the latter. In the first case, no time should be spent on examination of DO-loops for redundant sub-expressions, complex EQUIVALENCE structures should be avoided, the programmer should be generous with temporary PRINT statements (in default of a symbolic debugging package), the statements should be simple (no 10 continuation card arithmetic expressions).

SIS/kw/gs

Most of these things help compile time, as a normal compiler spends an inordinate amount of time chasing along EQUIVALENCE chains and decoding massive arithmetic expressions.

For the production program, of course, the goal is perhaps twofold -- to produce results fast, and also not to be oversophisticated so that the programmer who may one day inherit the job will not spend his entire career understanding what it does.

The following points are basically common-sense but perhaps worth iterating:

i) Program design is a logical procedure -- it is worth spending a fair proportion of one's time in the planning stage (especially for the production routines). Many people, including the writer, find flow charts invaluable.

ii) It is worth checking the library to see whether any method has been programmed already. In this context, the Programming Enquiry Office is there to help at the design stage as well as during the "debugging" stage.

iii) Keep up to date with system documentation. It is often very useful to know about the latest aids which are often designed to cut your through-put time.

iv) Do not be mean with comments. Without them it is particularly difficult to remember how a program you wrote two years ago really functions.

v) For all classes of user, it is generally true that simple FORTRAN statements produce the best results; in compilation-time, execution-time and, perhaps most important, "debugging"-time. There is a temptation occasionally to treat FORTRAN programming as a challenge to outwit the compiler and to "bend" the language. No compiler is perfect, and it is not hard to find ways of "breaking" it or deceiving it. However, the next FORTRAN compiler will have its own set of idiosyncracies which will probably cause you trouble had you pushed the language to the limit. An example of this is:

```
      .
      .
      END
     1 FILE 6          (continuation card).
```

3. <u>FORTRAN PROGRAMMING TECHNIQUES AND FORTRAN PROBLEMS</u>

In this section an attempt has been made to list points according to type of statement involved. This is a "rag-bag" of explanations of facilities which have caused problems in the past together with hints as how to improve object-time efficiency.

3.1 <u>Declarative statements</u>

3.1.1 <u>DIMENSION</u>

During the testing stage of a program, many people specify dimensions which are too large, subsequently failing to reduce them when a job goes into production. As space is important in a multi-programming system, it is worthwhile checking dimensions to see if they cannot be reduced. This also applies to the TYPE declarations and COMMON.

<u>Variable dimensions</u>

Many people have used variable dimensions with a feeling that in some mysterious way they save space. They do not. Space is allocated at compilation-time in the calling program and is never released at execution-time.

Originally the main reason for variable dimensions was to use a standard subroutine with varying calling routines. However, for those who are anxious about execution-time, these statements cause extremely laborious code to be generated, as the following example shows:

```
SUBROUTINE VARDIM (A,I,J)
DIMENSIØN A(I,J)
         .
         .
         .
A(K,M) = 3.0 .
```

To calculate the address referenced by A(K,M) requires the following:

$$\text{Addr of array } A + I * (M - 1) + (K - 1)$$

and this is even worse for three-dimensioned arrays. These calculations have to be performed at execution-time as the dimensions are <u>variable</u> during compilation of the subroutine.

If possible, always declare the full dimensions or the appropriate reduced size in <u>constant</u> form in the subroutine, unless it is required to change the row/column relationship, i.e.

```
DIMENSIØN A(10,10) in main program
           :
     CALL SUB (A,5,6)
```

3.1.2 <u>EQUIVALENCE</u>

This statement is normally used for the following reasons:

i) <u>Space-sharing</u> (sequentially) i.e.

```
EQUIVALENCE (A,B)
DIMENSIØN A (1000), B(1000).
```

A and B are not used at the same time.

ii) <u>Mixed-mode arrays</u>

An array containing fixed-pt and floating-pt numbers may be given a fixed <u>and</u> a floating name, and elements be referred to in either mode.

iii) <u>Ease of programming</u>

Mnemonic reference to array elements, e.g.

```
DIMENSIØN A (1000)
EQUIVALENCE (STOPFG, A(3)), (GOFAG, A(4)), .......
```

Over-elaborate use of equivalence tends to increase debugging problems and has an effect both on compilation- and execution-time, particularly the latter.

If the compiler is comparatively simple, it will not attempt to optimize code in which equivalence variables appear, as the following example shows:

```
EQUIVALENCE (A,Z).
Z = B + C * A
B = A/Q
```

The compiler will often try to remove redundant references to the same variable when translating to machine code. In the above example, if Z was not equivalenced, the compiler would load A into a register in the first statement and use the register again in the second reference to A. As Z is equivalenced it cannot do so, and the problem is that it requires a great deal of work to decide if a variable is equivalenced, whether all this sort of optimization should be stopped or not. Most compilers do not bother -- they just say the variable is equivalenced and stop it automatically.

3.1.3 COMMON

One advantage in using blank COMMON (apart from the possibility of redefining its size in later subroutines) so that in some systems it overlaps (or uses the same space as) the loader (i.e. the system program which accepts compiler output and lays it out in memory, linking all referenced routines, etc.).

Labelled COMMON is certainly more elegant, and has the advantage of programming ease. It also has the advantage of saving space in sequenced jobs where a particular labelled COMMON block is only referred to in a particular segment, whereas blank COMMON reserves space all the time.

3.1.4 DATA statement

Some points to emphasize.

i) The type of the constant takes precedence over the type of the array or variable.

ii) The use of octal constants as test patterns, e.g.

```
DATA (A = 616161B)
```

is system-dependent when used to compare, say, BCD characters on input. H-fields should be used for this sort of thing.

iii) Some compilers do not diagnose "spill" conditions, e.g.

```
DATA (Z = 1.0, 2.0)
```

(where Z is unsubscripted).

3.1.5 EXTERNAL statements

To pass the name of a subroutine or entry point to another routine, the EXTERNAL statement must be used, e.g.

```
EXTERNAL    EEE
CALL SETTZL (431,EEE).
```

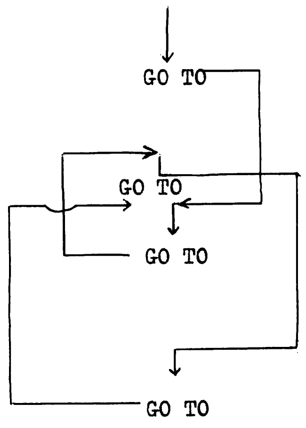
If it is not used, a local variable, EEE, will be assigned to the calling routine, and the called routine will transfer control to it, usually with disastrous results.

3.2 Control statements

3.2.1 GO TO statements

GO TO 100	- normal GO TO
GO TO (10,20,30,40),I	- computed GO TO
GO TO I	- assigned GO TO .

A general remark about transfer statements. Try to avoid "birds-nest" coding which has the following appearance.



This sort of thing tends to occur during development, and if this is a one-time job it is not important. However, all transfer statements inhibit optimization and the more there are, the slower the program.

The choice of ASSIGNED GO TO, COMPUTED GO TO is largely a matter of taste, but there are points to bear in mind:

The code generated for the ASSIGNED GO TO is invariably shorter than that generated for the COMPUTED GO TO, which has a "switch-table". However, the COMPUTED GO TO normally has a built-in check which is safer.

3.2.2 Subroutine CALLS

The information which may be passed to a subroutine via a CALL statement is of the following type:

Variable
Constant
Expression
Array
Array element
External symbol
H-string

The following points may be of interest:

i) When passing a constant, e.g. CALL SUB (2,3), watch for redefinition in the subroutine, i.e.

```
SUBROUTINE XYZ (I,J)
I = Y
J = Z
   .
   .
   .
```

This will have the effect of altering <u>all</u> references to 2,3 in the calling routine to the value of Y,Z.

ii) The habit of dividing a program into many subroutines, each of which do very little and pass on arguments to the next in the chain, is rather slow in execution-time.

iii) Where possible, use COMMON for passing arguments rather than formal parameters, especially arrays and expressions. Most compilers are more efficient in code generation where COMMON is used. This is because information is usually passed to a subroutine in such a way that to reference an argument <u>not</u> passed in COMMON requires approximately twice the time during execution than a reference to COMMON.

3.3 <u>IF statements</u>

In most compilers, the logical-IF statement is more elegant and even more efficient than the arithmetic-IF. Currently, in SIPROS, the arithmetic-IF is more efficient. The power of the logical-IF is to combine several conditions rather than to string such conditions in sequential arithmetic-IF's, e.g.

```
IF(A.EQ.B.OR.C.NE.D.AND.G.GE.H) DO THIS ...
```

A good compiler will ignore all subsequent tests as soon as possible, e.g. in the above expression, if A = B at execution-time the rest of the statements are ignored. Arithmetic-IF statements require <u>more statement numbers</u>, a factor affecting the compiler optimization performance.

3.4 <u>Arithmetic statements and expressions</u>

All, or nearly all other statements may have some form of arithmetic expression involved within, e.g. IF, CALL, and a reasonable compiler will have a standard set of routines which "break-down" such expressions. The form of expressions presented to the compiler has a significant effect upon its performance.

The following points may help to speed up execution:

i) The compiler is not of infinite size, and therefore it has built-in table limits. Those programmers who insist on constructing expressions covering several continuation cards are probably <u>not</u> getting the best out of the optimization process. This is because the compiler tries to produce as good code as possible up to the limit of information it can hold at one time. After that, it will tend to give up and revert to unoptimized mode. Moreover, it is not particularly easy to "debug" this sort of program.

ii) If one is really worried about execution-time speed, it may be worthwhile to look at function references, which cause a break in the optimizing sequence. If they can be extracted from the middle of expressions and used before, i.e.

```
Z = A(I)*3.142+SIN(Q)/X+A(I)/C
```

becomes

```
QQ = SIN(Q)/X
 Z = A(I)*3.142+QQ+A(I)/C.
```

iii) There should be no "side-effects" in function calls, i.e. a function should return one result and not alter any arguments or COMMON variables. Some compilers assume this to be the case.

iv) On the 6000-series machines, constant divisors should be avoided, if possible. Some compilers do this for the programmes, our current compilers do not,

e.g. `Y = X/2.0` becomes `Y = X* 0.5` .

Division on 6600 : 2.9 microsec
Multiplication " : 1.0 microsec (also can be overlapped).

v) Subscripts. It is difficult to draw a line between what the programmer should worry about, and what the compiler should do for him. In this less than perfect world, the programmer who wishes to get the best out of his program is advised to try to rearrange some subscripts (especially in DØ-loops) to help the compiler. Compilers tend to optimize subscript expressions which vary only in the constant part, i.e.

```
DIMENSION A(10,10), B(10,10).
```

a) `A(I,J) = B(I,J+3).`

The evaluation of the address A(I,J) is

Addr A+[10*(J-1)+I-1].

The evaluation of the address B(I,J+3) is

Addr B+[10*(J+2)+I-1].

The only difference is constant and the compiler will arrange to use the expression (10*J+I) in an optimized way, i.e. it will not recalculate.

b) `A(I,J) = B(J,I).`

Here the variable portion of the subscript differs and the compiler will probably not optimize.

The compiler can be helped, as the following example shows:

```
      DIMENSION H(10,10,10)
D(M,M) = H(L,J,N) + H(I,J,N) + H(L,J,K) + H(I,J,K) + H(L,M,N)
      + H(I,M,N) + H(L,M,K) + H(I,M,K).
```

The compiler will not recognize (L,J,N), (I,J,N) as partially similar, but if one sets

```
MM = L + 10(J-1) + 100 (N-1)
NN = MM - L + I
```

then

```
H(L,J,N) becomes H(MM,1,1)
H(I,J,N) becomes H(NN,1,1)
```

and similarly for

```
H(L,J,K), H(I,J,K)
H(L,M,N), H(I,M,N)
H(L,M,K), H(I,M,K).
```

This cuts out <u>two</u> multiplications for each subscript pair.

It should be emphasized that this sort of thing is only for those programmers who are anxious to speed-up sensitive areas of production programs.

vi) Avoid redundant parentheses.

vii) Avoid redundant statement numbers.

3.5 <u>DO-loops</u>

The general rule for making a DO-loop more efficient is:

> only leave <u>within</u> the loop such information which is directly or indirectly dependent upon the DO-index, or information which changes during the processing of a loop.

<u>Remember</u> the DO-index should <u>not</u> be altered during the processing of a DO-loop. Particularly important to watch when a DO has an <u>extended range</u>, i.e.

```
       DO 10 I=1,5
            .
            .
     CALL   XYZ(I,A(I)).
            .
            .
   10 CONTINUE
```

It is probably safer to say ITEMP = I and send ITEMP across to XYZ, or get XYZ to set ITEMP from I.

The extended range of a DO ruins all the compilers good intentions to optimize your loop as <u>any transfer statement</u> causes such optimization to cease.

<u>Remember</u> the following transfer situation is ambiguous and although not specifically disallowed will cause trouble:

```
              DO    100   I=1, 10
                     :
                     :
 ┌──────────── GO    TO  100
 │             DO    100   J=1, 10
 │                    .
 │                    .
 │             DO    100   K=1, 10
 └──────────→ 100 CONTINUE
```

This will cause a transfer to the beginning of the innermost DO(DO 100 K=1,10) as the CONTINUE statement expands as follows:

```
100     K = K+1
       IF  (K.LE.M2) GO TO K-LOOP
        J = J+1
       IF  (J.LE.M2) GO TO J-LOOP
        I = I+1
       IF  (I.LE.M2) GO TO I-LOOP
```

The CONTINUE statement in this context should be thought of as part of the innermost loop and therefore such a transfer is one from outside a loop into the range of such a loop.

3.6 Extracting information

i) Look for constant statements:

e.g.
```
          DO   100  I = 1,20
                X = 3.0
      100       A(I) = X*B(I)
```
X = 3.0 should be extracted from the loop.

ii) Look at statements for divisions:
```
          DO   100 I=1,20
     100   A(I) = B(I)/Z
```
Why not say ZZ = 1/Z and rewrite
```
          DO   100 I=1,20
     100   A(I) = B(I)*ZZ
```

iii) Look for static sub-expressions in statements:
```
          DO     100  I=1,20
     100   A(I) =  C(I)*SIN(Z)*X.
```
SIN(Z)*X is static, should be:-
```
          CC = SIN(Z)*X
          DO 100  I=1,20
     100  A(I) = C(I)*CC
```

iv) Look at subscripts:
It is recommended that this is only done by those programmers who have a definite need to optimize particular loops in FORTRAN. Let us assume we have the following situation:
```
DIMENSION A(10,10)
     .
DO 10 J = 1,10
     .
     .
DO 20 I = 1,10
A(I,J) = ......
```
If nothing is done by the programmer, a simple compiler will recalculate A(I, J) each time.
However, we know J is constant in the inner loop. So, if in the outer loop we set

JJ = 10(J-1) + (Value of m_1 , first indexing para).

KK = JJ + (m_2 - m_1)

then the loop may be rewritten:
```
DO  20   I = JJ,KK
A(I,1) = .........
```
and the calculation of (10(J-1) + I-1) is kept outside the loop.

A more difficult case is:

```
DO 100  K = ....
   :                        DIMENSION A(10,10,10).
DO 200  J = ....
   .
   .
DO 300  I = ....
A(J,K,I) = .....
```

where I varies in the subscript by another factor than m_3. A(J,K,I) can be defined as:

$$\text{Addr A} + [(J-1) + 10(K-1) + 100(I-1)].$$

This calculation takes place each time around the inner loop, and it is worth getting outside. Therefore I varies by a factor which is a function of D_1, D_2, the first two dimensions of the array. If we were compilers, we would try to arrange for an index IMOD, say, which alters by such a factor that within the loop one could say

A(IMOD,1,1) <u>NOT</u> A(J,K,I)

and setting the initial IMOD value outside the loop. This can be done by setting

IMOD = Invariable part of index (i.e. J,K part) + $(m_1 - 1)$* I-factor $(D_1 {*} D_2)$.

<u>Example</u>

```
DIMENSION A(10,10,10)
DO    300 I =   3,10
A(J,K,I) = .....
     :
     ↓
becomes:
DIMENSION A(10,10,10)
IMOD = J+10*(K-1)+(3-1)*100      (3 = m1)
DO 300 I = 3,10
A(IMOD,1,1) = .....
IMOD = IMOD + α         (α = 10*10) (set outside the loop).
   :
   :
```

4. <u>CONCLUSION</u>

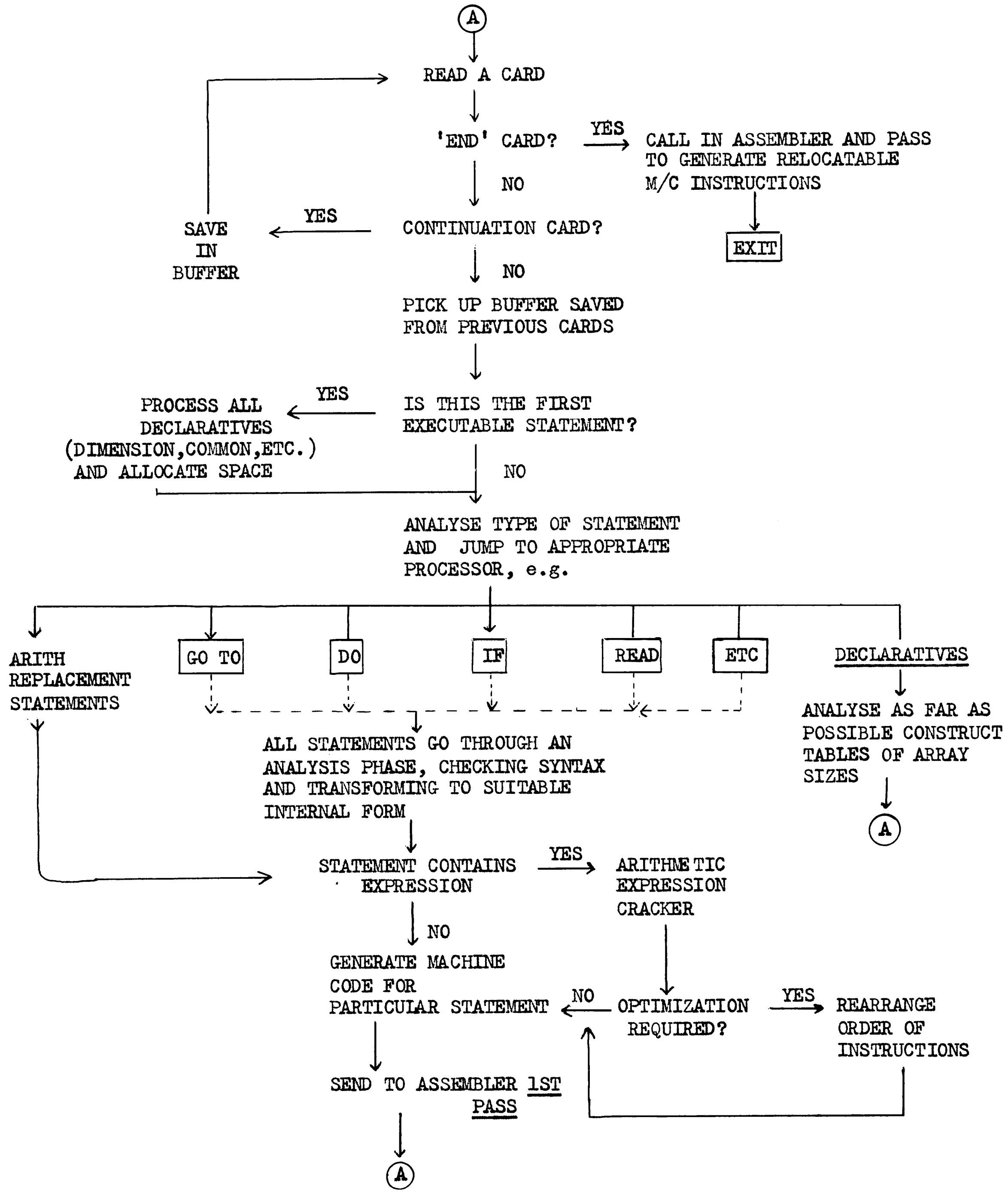
As stated earlier, this discussion can hope to do no more than touch on a miscellany of odd points which may help the FORTRAN programmer. However, the purpose of these seminars is to encourage the computer user to consider ways in which he can help the turn-round situation by writing more efficient programs, using the latest system facilities and gaining a deeper insight into the way in which various parts of the various systems do their job. Ultimately it is the user's responsibility to decide how he wants to construct his jobs, and what he wants to do with them.

Well-constructed FORTRAN programs are largely a matter of common sense. Decisions as to whether to concentrate on speed, space-saving, or rapid debugging are a matter for the user, but it is the writer's contention that simple FORTRAN will normally give him all three (except in the rather sophisticated techniques described for DO-handling which are not recommended for general use).

APPENDIX I

GENERAL FLOW CHART OF A FORTRAN COMPILER

This flow chart is based on the current SIPROS compiler.


A
READ A CARD
'END' CARD?
YES
CALL IN ASSEMBLER AND PASS TO GENERATE RELOCATABLE M/C INSTRUCTIONS
EXIT
NO
CONTINUATION CARD?
YES
SAVE IN BUFFER
NO
PICK UP BUFFER SAVED FROM PREVIOUS CARDS
IS THIS THE FIRST EXECUTABLE STATEMENT?
YES
PROCESS ALL DECLARATIVES (DIMENSION,COMMON,ETC.) AND ALLOCATE SPACE
NO
ANALYSE TYPE OF STATEMENT AND JUMP TO APPROPRIATE PROCESSOR, e.g.
ARITH REPLACEMENT STATEMENTS
GO TO
DO
IF
READ
ETC
DECLARATIVES
ANALYSE AS FAR AS POSSIBLE CONSTRUCT TABLES OF ARRAY SIZES
A
ALL STATEMENTS GO THROUGH AN ANALYSIS PHASE, CHECKING SYNTAX AND TRANSFORMING TO SUITABLE INTERNAL FORM
STATEMENT CONTAINS EXPRESSION
YES
ARITHMETIC EXPRESSION CRACKER
NO
GENERATE MACHINE CODE FOR PARTICULAR STATEMENT
NO
OPTIMIZATION REQUIRED?
YES
REARRANGE ORDER OF INSTRUCTIONS
SEND TO ASSEMBLER 1ST PASS
A

FORTRAN INPUT-ØUTPUT STATEMENT AND ERROR RECOVERY PROCEDURES

by

J. Garratt

CONTENTS

1. INTRODUCTION

The purpose of this paper is to attempt to give the FORTRAN programmer an insight into the mechanics of input/output sequences of operations initiated by the FORTRAN statements, READ, READ(i), WRITE(i), PRINT, etc. The uses of ENCØDE/DECØDE are briefly discussed, and their relative merits for certain tasks examined. The philosophy discussed is that. applicable to SIPROS, but much of the general logic is common to other systems. A short description of the error recovery procedures available in the SIPROS follows the main description, in order to clarify their use with I/Ø operations.

2. BINARY INPUT/ØUTPUT

In the SIPROS system, binary information is stored on tape in logical records, which may consist of one or more physical records. A logical record represents the amount of information contained in the list of a WRITE(i) statement. A physical record normally consists of 512 full central memory words of information, plus a 12-bit "byte" containing the number of the physical record within the logical record. The last physical record (or the only physical) of a logical record may contain 1 to 512 CM words with no trailing byte. Thus a logical record is a well-defined entity when stored on tape.

The SIPROS system is also capable of reading/writing tapes in IBM 7090 format.

Binary input/output is achieved by use of the FORTRAN statements, READ(i) list and WRITE(i) list, and we shall now examine these statements in more detail.

Let us take an example of a READ(i) list statement

```
READ(5)A, B(3), (C(I), I = 1, 100).
```

Here the programmer wishes to read 102 words from a logical record into the elements A, B(3), and the first 100 locations of C.

The FORTRAN compiler, on meeting a statement of this form, will generate calls to execution-time routines which will deal with the transfer of information from the next logical record on log. unit 5 to the list, as follows:

```
(1)                    CALL IØRWB1(5, code)
(2)                    CALL FØLIB(A, B(3))
(3)                    DØ x I = 1, 100
(4)                 x  CALL FØLIB(C(I))
(5)                    CALL IØRWB3.
```

Let us now analyse these calls a little further:

(1) The first call to IØRWB1 is an initialization call which sets up internal flags and counts within the execution-time I/Ø routines and checks the validity of the log. unit number. The "code" defines the operation.

(2) This is followed by one or more calls to FØLIB, dependent on the list structure, which is the routine responsible for transferring data from the record to the list. It processes as many items per call as possible and returns only in the following situations:

SIS/kw/gs

a) A DØ-loop structure (e.g. C(I), I = 1, 100) is encountered in the list

b) On reading, a list of the form I, A(I),... where the value of I will change before A(I) is filled.

A general flowchart of FØLIB, IØRWB1 and IØRWB3 is given in Appendix I.

Generally speaking, FØLIB is a simple loop, which reads a physical record from the tape into a CM buffer of 512 words and then transfers data from this buffer to the list, asking for another physical when the buffer is empty, or returning control to the calling program when no more list items remain to be processed. When it finds it necessary to ask for a new physical record, control is taken away from the program until such time as the record has been transferred to the buffer by the peripheral processor program which reads the tape.

(3), (4) In the case of DØ-structures, one item only is passed at a time to FØLIB, the DØ-loop itself being held in the calling program.

(5) IØRWB3 is a terminating routine which "skips" the rest of the logical record (if necessary) when the list is exhausted. This is carried out by successive reads until a "short" physical record (i.e. not 512 + byte) is encountered. The tape is now positioned to be read again at the beginning of a new logical record.

Error conditions are discussed in the section on "Error Procedures".

Similarly a WRITE(i) list statement of the following form:

```
WRITE(5)A, B(3), (C(I), I = 1, 100)
```

would be expanded to the following calls by the compiler

```
    CALL IØRWB1(5, code)
    CALL FØLIB(A, B(3))
    DØ x I = 1. 100
x   CALL FØLIB(C(I))
    CALL IØRWB3 .
```

The "code" in the initialization call to IØRWB1 defines the operation as read or write.

The same remarks about the list structure apply to writing as to reading.

For writing, FØLIB acts in the reverse direction, taking data from the list and storing it into the CM buffer until the buffer is full, then issuing a request to write one physical record on log. unit 5. For each physical written by FØLIB (i.e. not the last physical of a logical) the physical record number is attached to the end of the 512 word data blocks.

IØRWB3 writes out the last physical record from the data left in the buffer from the last call to FØLIB.

3. SUMMARY

(1) Within SIPROS, all FORTRAN binary reading/writing is performed by the routines, IØRWB1, FØLIB, IØRWB3.

(2) Information is transferred to and from a CM buffer of 512 words in length, physical record at a time, control being taken away during the transfer time.

(3) List structures of the form C(I,...), I = ... tend to cause slow execution as they are dealt with word at a time.

(4) NB. The call to FØLIB, although shown above as a FORTRAN CALL, is non-standard and cannot be simulated.

4. SHORT FORM OF ARRAY

Statements of the form READ(i)A, WRITE(i)A, A being an array, will be executed much faster than the alternative DØ-form. This is because the size of the array is passed in one call to FØLIB which can organize a very fast transfer loop rather than satisfying the list, one element at a time.

5. BCD INPUT/ØUTPUT

BCD data may come from or be written to the following media:

(1)	Magnetic tape
(2)	Card reader
(3)	Printer
(4)	Card punch .

We shall consider the input/output of information to and from magnetic tape in this section, although the action of the CP routines is virtually identical for the other media (the only difference being the conventional size of unit record for each medium).

Items (2), (3), (4) above are transferred to and from disk as far as the BCD I/Ø routines are concerned, i.e. the routines never "read a card" direct from the card reader or "print a line" direct to the printer. All transfers are carried out via a CM buffer of 512 words to and from the disk.

However, information on tape is transferred directly, record at a time, to and from a small 14 word buffer held within the I/Ø execution-time routines. This buffer is also used to hold individual records obtained from the large CM buffer for card reading/printing, etc. A simple diagram is shown in Appendix II.

The size of a BCD record on tape is 14 CM words, and conventionally a physical record is equivalent to a logical record in BCD operations (for backspacing, etc.).

From this point, the notion described is identical for BCD records handling irrespective of the medium.

Let us take an example of a typical READ(i,n)... statement

```
      READ(5, 10)IA, B, C
10    FØRMAT(5x, I3, A4, E10.3).
```

The next record on log. unit 5 contains the following information

```
                         BCD chars.

          word 1  :   b b b b b 1 0 4 A B
               2  :   C D b b b b b 2 1 3
               3  :   . 4 b b b b b b b b
            4-14:        all blank
```

A simplified flowchart of the I/Ø routines (BCD) appears in Appendix III.

Under SIPROS, the READ(i,n) statement is compiled into a sequence:

(1)	CALL FRMTAN1(5, 10, code)
(2)	CALL FØLIØ(IA, B, C)
(3)	CALL FRMTAN3 .

(1) FRMTAN1 is the initialization routine which checks the log. unit number, picks up the FØRMAT conversion information necessary, and initiates the first read from the specified log. unit.

When FRMTAN1 returns control, the situation is as follows:

a) the record described earlier is in the buffer as shown;

b) the FØRMAT statement has been scanned and the 5x has been obeyed, and the first conversion specification I3 is ready for the first conversion.

Assume a pointer to the buffer, it would appear

$$\text{word 1:}\quad \overbrace{\text{b b b b b}}^{(5x)}\ \underset{\uparrow}{1}\ 0\ 4\ A\ B$$

(next character to be taken).

(2) FØLIØ is the routine which takes one conversion specification from the FØRMAT, applies it to the data within the buffer, calls the appropriate conversion routine, and stores the result in the next item in the list. It then calls the routine which gets the next FØRMAT specification and loops until the list is exhausted.

The sequence in the above example would be:

a) Pick up three characters, 104, from buffer according to I3.
b) Convert to binary and store in IA.
c) Pick up next conversion specification, A4.
d) Pick up next four characters in buffer, ABCD.
e) Store left justified in B, i.e. ABCDbbbbbb.
f) Pick up next 10 characters, bbbbb213.4.
g) Convert according to E.format.
h) Store in C.
i) Pick up next conversion specification: ")"(end-of-record).
j) List exhausted, so return.

(3) FRMTAN3 is a termination routine which resets the buffer, etc. For output, e.g. [WRITE(i,n)...] it would write any output record set up in the buffer by FØLIØ.

The sequence of events for WRITE(i,n) list is very similar except of course, that the transfer of information is from list to buffer, e.g.:

```
         WRITE(5, 10) IA, B, C
10       FØRMAT(5x, I3, A4, E10.2)
```

would be compiled into the sequence:

```
CALL FRMTAN1(5, 10, code)
CALL FØLIØ(IA, B, C)
CALL FRMTAN3.
```

In the case of writing FRMTAN1 sets up an empty buffer, into which FØLIØ transfers characters according to the appropriate conversion specification. Whenever the FØRMAT, is completed, i.e. the last right parenthesis is encountered and more information remains to be output, a record is written from the buffer to the output medium. This also applies, of course, if slashes (/) are encountered within the FØRMAT.

FRMTAN3 outputs the last record.

The transfer of records is governed only by the FØRMAT specification, not by any overflow in the CM buffer, e.g.:

```
FØRMAT(160A1).
```

This is treated as a FØRMAT error condition at execution-time.

6. SUMMARY

(1) Under SIPROS, all FORTRAN BCD input/output is achieved by the routines, FRMTAN1, FØLIØ, FRMTAN3.

(2) Information is dealt with record at a time by these routines. Control is taken away whenever the central program would have to wait for a transfer to/from an external medium.

(3) Each list element associated with a FØRMAT specification requires a conversion process to complete the transfer to and from the record. This applies even to A-format, where strictly speaking no conversion is carried out, but information is transferred a character at a time. All conversions are done in the central processor -- information passed to the appropriate PP routines is in external character form.

(4) Short forms of arrays are handled more efficiently than the alternative DØ-structure although this has little effect on the total time taken to transfer BCD data because of the conversion/packing process.

(5) Where possible, BCD operations should not be performed where they can be replaced by binary operations, for the following reasons:

a) on tape, the packing density under SIPROS is far inferior (a maximum record size of 14 words);

b) the FORTRAN I/Ø routines to handle such operations are considerably slower for the reasons mentioned earlier.

It is recognized that tapes are produced which have to be printed or used outside CERN. There appears to be no reason why BCD records should be written on to 1" tape.

7. ENCØDE/DECØDE

The main point to be remembered about these statements is that they are essentially BCD I/Ø statements with the exception that the transfer of records is memory to memory, rather than memory to/from some external medium (e.g. tape).

The statements generate calls to the same I/Ø routines mentioned above, i.e. FRMTAN1, FØLIØ, FRMTAN3.

They are therefore no faster than any other BCD Input/Output as far as CP time is concerned.

DECØDE can be used for records of variable format, in order to decipher some keyword within the record and then convert the rest of the information according to such a keyword.

<u>Example</u>

```
          DIMENSION IX(4)
          DATA(IA = 6H HEADER)
    1     READ(5, 100)IX
          IF(IX(1).EQ.IA)GØ TØ 2
          DECØDE(36, 200, IX)X,Y,Z
             :
    2     DECØDE(36, 300, IX)A,B,C
             :
  100     FØRMAT(4A10)
  200     FØRMAT(10X, E15.3, I5, A6)
  300     FØRMAT(10X, 2F10.0, I6).
```

The record is read in with a normal READ(i,n) statement in A-format, and different DECØDE statements are used according to the information in the first word.

8. ENCØDE

Programmers who use ENCØDE for packing into records in A-format and writing BCD records in this way might consider whether they could equally use the system "packing/unpacking" routines, BITSZA AND IMBDZA, and write the information in binary mode.

ENCØDE is also used for creating such tools as variable FØRMATS.

An example of this usage follows, together with an alternative method which uses more space, but is considerably faster in execution.

Example

To create a variable FØRMAT capable of printing I/J where J may vary in size from 2 to 1000, without right-justifying J within an I4 field, e.g. 1/bbb2. We require a FØRMAT containing I1, I2, I3 or I4 dependent on size of J. Assume IA contains size of I-filled required.

9. USING ENCØDE

Assume array IVAR contains (before printing I/J)

```
word 1:   (5X, I2, 1bb
word 2:   H/, In) .
```

Using ENCØDE as follows:

```
ENCØDE(16, 100, IVAR)IA
FØRMAT(14H(5X, I2,1 bb H/, I, I1, 1H)).
```

This will construct the array as shown above, with n = current value of IA.

I/J can be printed out using

```
PRINT IVAR, I, J.
```

A faster method:

```
DATA(IVAR:   10H(5X, I2, 1, 4HH/, I, 1Hb, 1H))
```

sets up IVAR as follows:

```
    IVAR(1):  (5X, I2, 1
    IVAR(2):  H/, I
    IVAR(3):  (blanks)
    IVAR(4):  )
DATA(IBCD:  1H1, 1H2, 1H3, 1H4).
```

The following sequence will produce the same result:

```
IVAR(3): IBCD(IA)
PRINT IVAR, I, J.
```

Generally speaking, the use of ENCØDE/DECØDE should be very carefully considered where CP execution-time is important. Where the same result can be achieved by some other method, it is advised that another method be chosen.

10. ERROR PROCEDURES

The use of SETTZL/EXEMZL, the general error routines, is described in the CERN CDC 6600 computer Operations File (Section 3.2). The following is a brief description of the method by which these routines interface with the FORTRAN I/Ø routines and the users recovery routine.

If the user wishes to recover from a particular error condition (e.g. read parity) he will call SETTZL with an error number and the name of a routine to which he wishes control to be passed if the error occurs (or RETURN if he wishes standard recovery to be taken).

SETTZL stores the error number and the routine address in a table of "activated error conditions".

If the FORTRAN I/Ø routine now senses a read parity indication (returned from the PP routine reading the tape) it calls EXEMZL with an error number, a message, and information concerning where EXEMZL should return control within the I/Ø routine.

EXEMZL will check the error number against those set by SETTZL and, if present, pass control to the user's routine specified. The user may now:

a) set an indication for his own calling program (e.g. set a common flag);

b) return <u>or</u> exit by calling another routine which will re-initiate the whole sequence of his program.

If the user wishes to accept the data he must return to EXEMZL by a normal RETURN statement, because the list will remain unsatisfied unless he does so. Moreover, if he wishes to return, he must <u>not</u> call further I/Ø statements within his recovery procedure, because these may in turn cause errors which transfer again to EXEMZL, destroying the linkage, e.g. if he now tries to return via EXEMZL, he will loop.

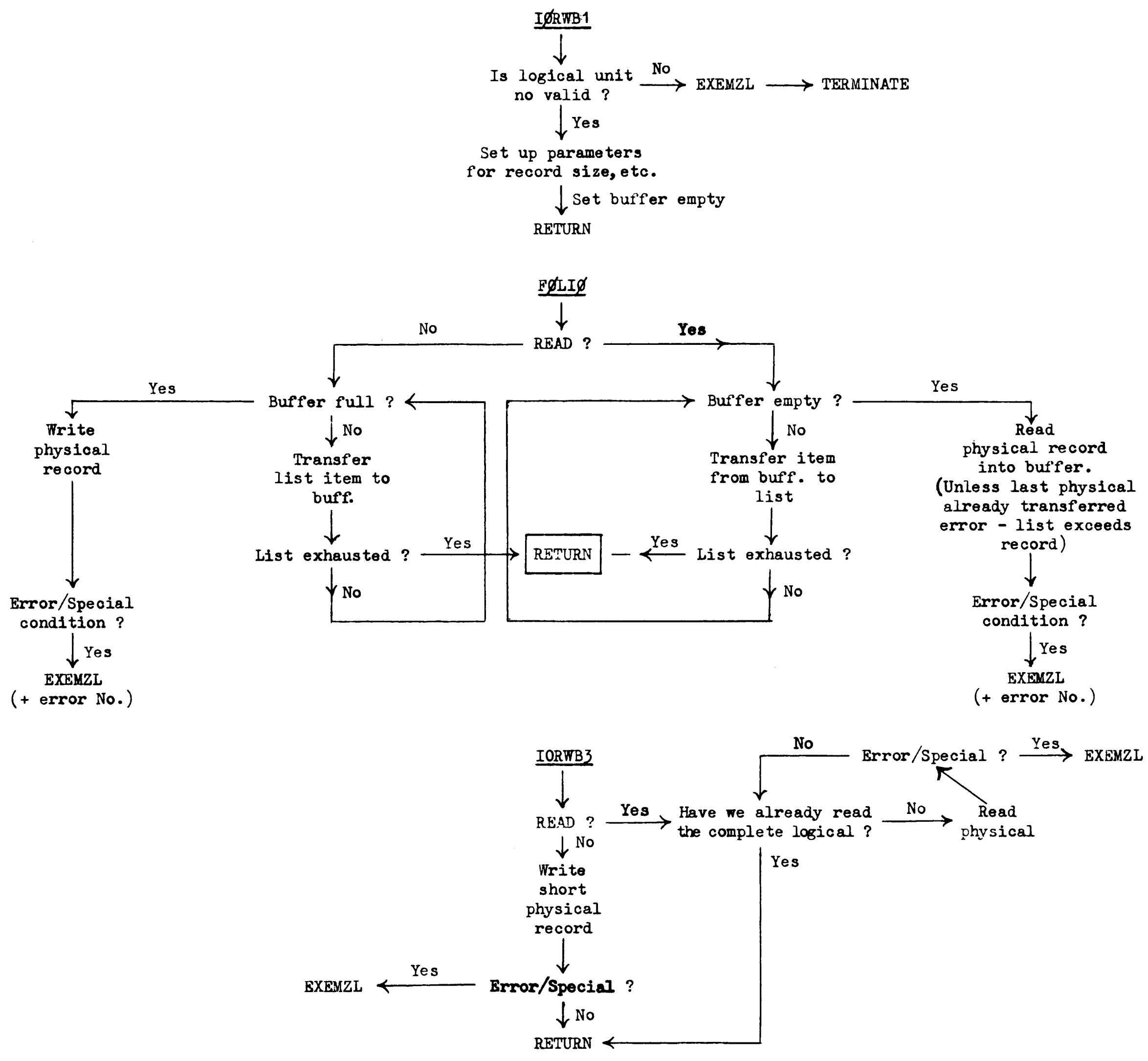
Similarly, if he tries to use the same I/Ø statement within the recovery routine, he will loop when he tries to return at the I/Ø statement level.

Generally speaking, if the user intends to exit from his recovery routine, he may use further I/Ø statements.

If he wishes the I/Ø statement to be completed normally, then he should return via EXEMZL, as the error is passed to the error routine <u>when it is encountered</u>, which means that the operation may not be completed at that point.

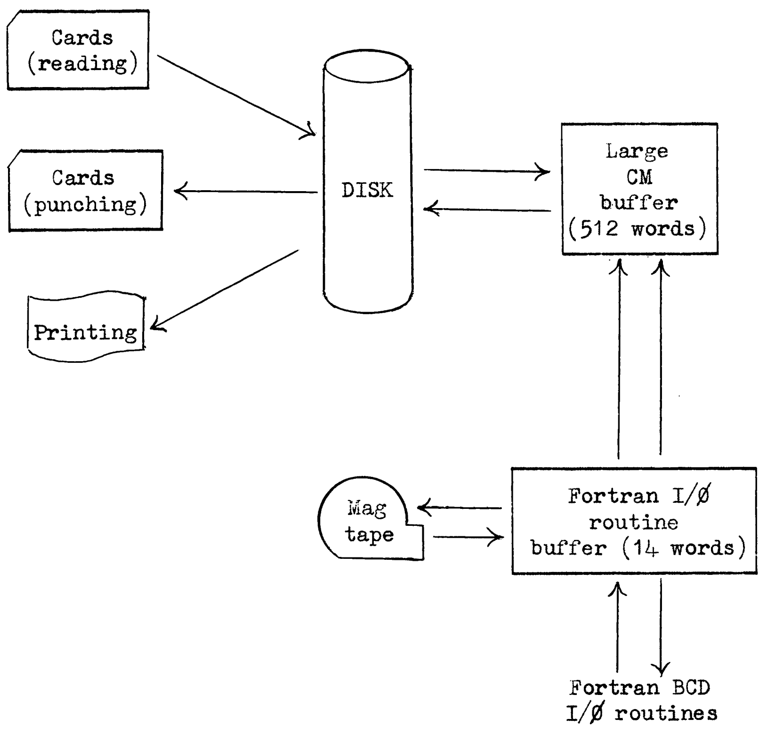
APPENDIX I

GENERAL FLOW OF FORTRAN I/Ø ROUTINES
(IØRWB1, FØLIB, IØRWB3) (BINARY)


IØRWB1
Is logical unit no valid ?
No
EXEMZL
TERMINATE
Yes
Set up parameters for record size, etc.
Set buffer empty
RETURN
FØLIØ
READ ?
No
Yes
Buffer full ?
Yes
Write physical record
Error/Special condition ?
Yes
EXEMZL (+ error No.)
No
Transfer list item to buff.
List exhausted ?
Yes
No
RETURN
Buffer empty ?
No
Transfer item from buff. to list
List exhausted ?
Yes
No
Yes
Read physical record into buffer. (Unless last physical already transferred error - list exceeds record)
Error/Special condition ?
Yes
EXEMZL (+ error No.)
IORWB3
READ ?
Yes
Have we already read the complete logical ?
No
Read physical
Error/Special ?
Yes
EXEMZL
No
Yes
No
Write short physical record
Error/Special ?
Yes
EXEMZL
No
RETURN

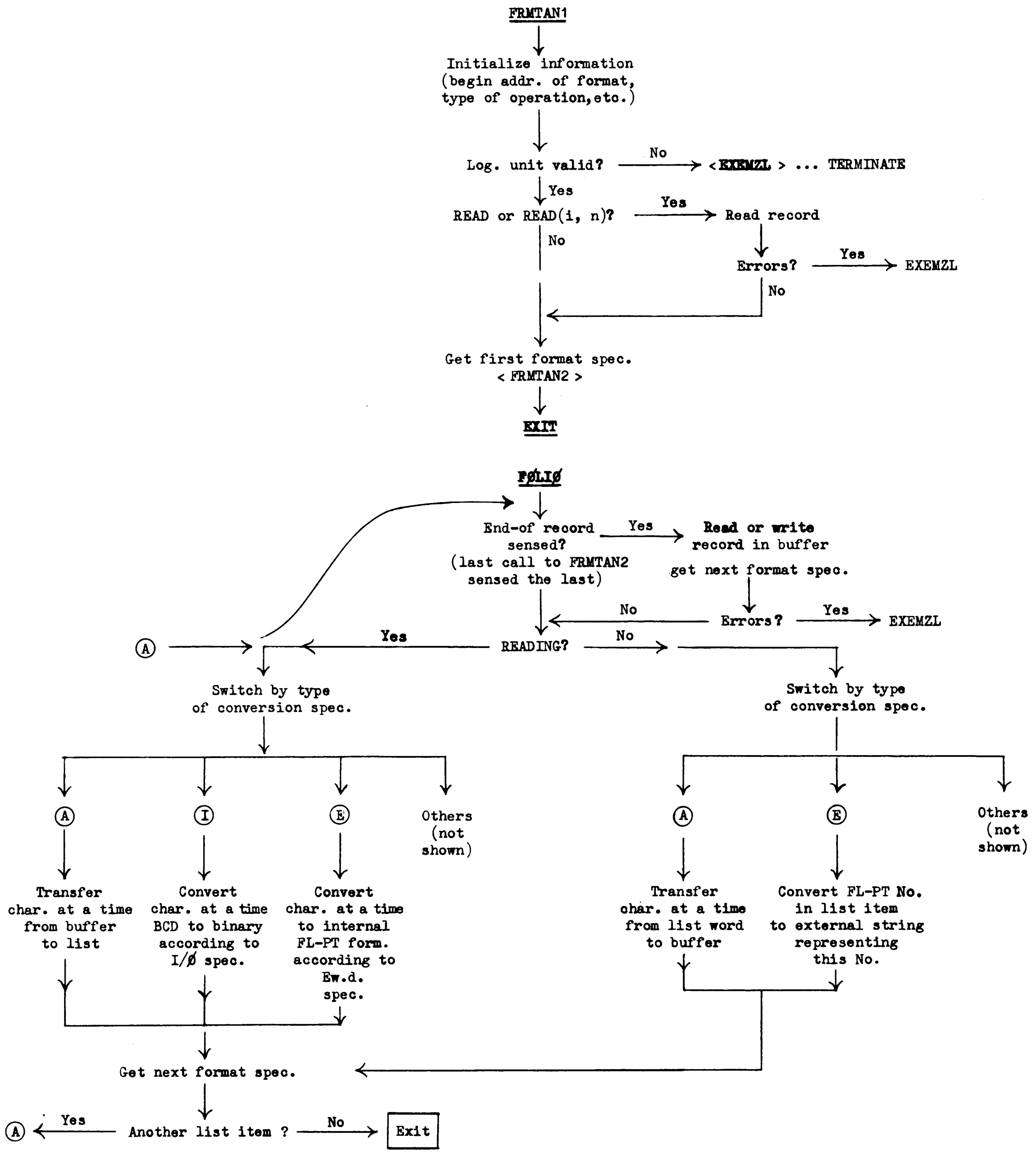
APPENDIX II

INTERFACE BETWEEN FORTRAN BCD I/Ø ROUTINES AND EXTERNAL MEDIA

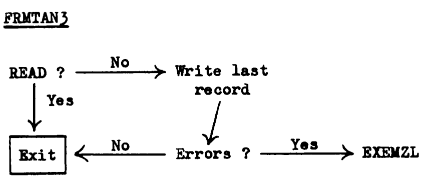


APPENDIX III

GENERAL FLOW OF BCD I/Ø ROUTINES (FRMTAN1, FØLIØ, FRMTAN3)


FRMTAN1
Initialize information (begin addr. of format, type of operation,etc.)
Log. unit valid?
No
<EXEMZL> ... TERMINATE
Yes
READ or READ(i, n)?
Yes
Read record
No
Errors?
Yes
EXEMZL
No
Get first format spec. < FRMTAN2 >
EXIT
FØLIØ
End-of record sensed? (last call to FRMTAN2 sensed the last)
Yes
Read or write record in buffer get next format spec.
No
Errors?
Yes
EXEMZL
A
Yes
READING?
No
Switch by type of conversion spec.
A
I
E
Others (not shown)
Transfer char. at a time from buffer to list
Convert char. at a time BCD to binary according to I/Ø spec.
Convert char. at a time to internal FL-PT form. according to Ew.d. spec.
Switch by type of conversion spec.
A
E
Others (not shown)
Transfer char. at a time from list word to buffer
Convert FL-PT No. in list item to external string representing this No.
Get next format spec.
A
Yes
Another list item ?
No
Exit

NB. If a character is sensed which is invalid according to its conversion specification, EXEMZL is called, and the return will replace such a character by blank.
If the buffer overflows, a FØRMAT error is diagnosed.



COS--CHIPPEWA OPERATING SYSTEM

by

A. Maver

CONTENTS

1. INTRODUCTION

This paper describes some fundamental characteristics of the Chippewa Operating System (COS)[*]. The description is based on an example of a job processed by the system from the initial phase when it is loaded from the card reader to the termination phase when the output produced by the job is printed and punched. The example of the job is shown in Fig. 1.

```
JØB.
CØMMØN FILE1.
CØMMØN FILE2.
REWIND(FILE1)
REWIND(FILE2)
RUN(S)
LGØ.
RELEASE FILE2.
EØR
         PROGRAM EXAMPLE (INPUT, ØUTPUT, FILE1, FILE2, TAPE3 = INPUT,
      1                   TAPE1 = FILE1, TAPE2 = FILE2)
         ------------------------------------------------------------
         STOP
         END
EØR
DATA SET
EØR
EØF
```

Fig. 1: Example of a COS job

The first card of a COS job must be the job card. It contains as a minimum of information the job name (JØB in our case). Other information like field length, priority and time limit may be specified on the job card. For the sake of simplicity they will not be considered here. In our case the system itself sets this information to fixed system parameters. The job card is followed by a set of job control cards, terminated by an end-of-record card (EØR). The control cards determine which programs are to be executed as part of the job and under which conditions (equipment, central memory space, etc.) they are to be executed. Then a program card follows and all the cards constituting the (FORTRAN in our case) program. The program is terminated by an EØR card. A set of data cards follows, terminated by an EØR card. The end of the job is indicated by an end-of-file (EØF) card.

2. SYSTEM ORGANIZATION

To understand better the processing of our job, some words must be said about the system organization: about the system components and about the communication between them.

COS is initially loaded from the system tape, and during the loading its components are stored in the central memory, on the 10 peripheral memories, and on the magnetic disk. The

*) COS is an early version of the SCOPE system currently in use on the CDC 6000 series computers. The text in this paper is therefore out of date if referred to the SCOPE system run at CERN, and should be completed with the appropriate documentation, mainly: 6000 Series SCOPE Handbook and 6000 Series SCOPE General Reference.

SIS/kw/eg

most frequently used programs and tables are stored in the central memory and in the peripheral memories, the others on the disk.

The central memory contains permanently the Central Memory Resident (CMR), which is composed essentially of tables, some frequently used central processor and peripheral processor routine libraries, and pointers to these tables and libraries. It also contains two small central processor routines: the MØVE routine used to re-arrange the assigned central memory space in contiguous form as jobs are terminated and new jobs arrive for execution, and the IDLE routine, executed whenever there is no other job which can use the central processor. The CMR occupies about 6K words. The rest is left for user jobs. Figure 2 shows a simplified layout of the central memory organization.

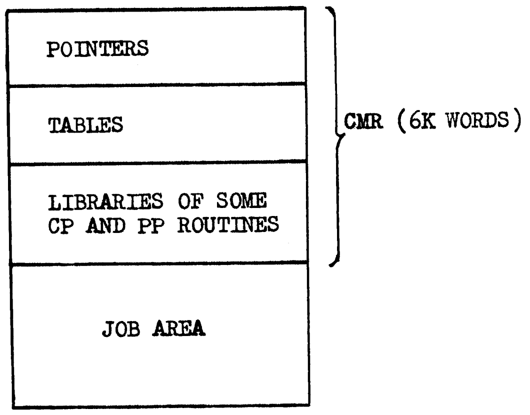


Fig. 2: Central memory organization

Of the 10 peripheral processors (Fig. 3), one (PP0) is assigned permanently to the System Monitor (MTR) and one (PP9) to the System Display Program (DSD). These two processors cannot be used for any other purpose.

MTR acts as the supervisor of all the system activity: it initiates and terminates the execution of jobs in CM; it monitors central processor programs for I/Ø requests and assigns peripheral processors to execute these requests; it assigns data channels and peripheral equipment; it administers the disk files, maintains the system time accounting, and so on.

DSD is responsible for the communication with the display console; it accepts commands from the operators and sends requests to the operators; it displays the actual status of the system components.

The peripheral processors P1, P2, ..., P8, constitute the pool of free processors. They have no fixed assignment. Their activity is subordinated to the monitor which asks them to perform specific functions. The communication between the peripheral processors and the MTR is handled by the peripheral processor resident, stored permanently in each of the 10 peripheral processors' memories. For every MTR request, the peripheral processor resident loads a system routine and executes it. Every time a routine has to be executed by a peripheral processor, it is loaded expressly and then executed. When the execution is terminated, the peripheral processor is returned to free status and can be assigned by MTR to perform another function. The routines loaded by the peripheral resident are called transient routines. They can, in their turn, load overlay routines to perform more specific functions.

In this way all users' I/Ø operations are treated, as well as operations concerned with the loading and termination of jobs and generally with the handling of the job flow through the system. As an example, consider the transient routine CIØ (Combined Input/Øutput). A job wanting to perform an I/Ø operation sends a request for CIØ to MTR, specifying also the type of operation (read, write, etc.). MTR sends the request to a free peripheral processor which loads CIØ. CIØ calls first the overlay 2BP to check the buffer parameters and the legality of the operation, and then the specific overlay to perform the operation requested (2RD to read from disk, 2WD to write to disk and so on).

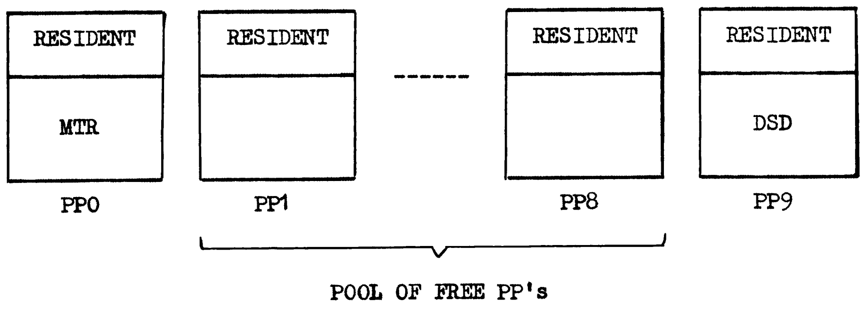


Fig. 3: Peripheral processor assignment

For each job executing or waiting for execution in central memory, COS maintains some basic information which permits it to switch the central processor from one job to another at any time. This basic information is kept in appropriate areas, called the <u>Control Point Areas</u> (CPA).

The information in the CPA includes the register contents (exchange-jump package) the job name and priority, information about the time and space used by the job, the equipment and PP's assigned to the job, the job control statements as shown in Fig. 4.

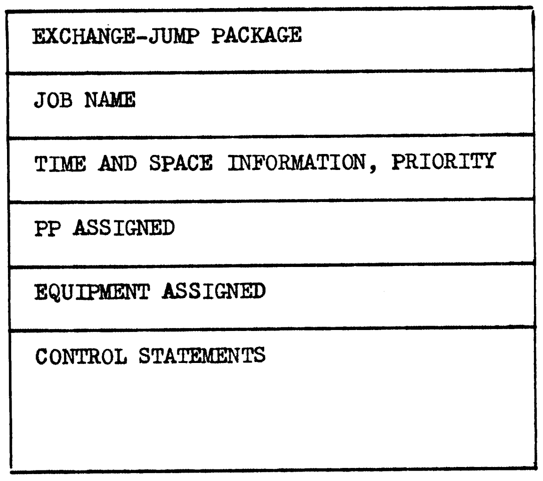


Fig. 4: Control point area

There are seven control point areas, numbered from 1 to 7. The numbers associated with the control point areas are called the <u>control points</u>. A job, the basic information of which is contained in one of the control point areas, is said to be assigned to the (corresponding) control point. Only one job can be assigned at any time to a control point, so that at most seven jobs can be executed concurrently under the control of COS.

Not only CP jobs are assigned to control points. Jobs using only PP's are assigned to control points as well. Examples of such jobs are the Print and Card Read routines each of which resides on a proper control point. These are system routines, but any other job using exclusively PP routines would also be assigned to a control point. To conclude: any CP or PP activity must be assigned to a control point. Figure 5 shows an example of control point assignment to various jobs.

Control point	Reference address	Field length	Type of job
1	14000	3000	PP using CM
2	17000	10000	CP
3	27000	30000	CP
4	57000	0	PP not using CM
5	57000	0	Unassigned
6	57000	2000	PP using CM
7	61000	15000	CP

Fig. 5: Example of control point assignment

3. <u>COS FILE SYSTEM AND JØB EXECUTION</u>

Under COS all I/Ø information pertinent to users as well as to the system is transmitted in the form of <u>named files</u>. Every file handled by COS must have a name and any reference to a COS file must be made via the file name.

Externally, from the users' standpoint the name identifies the file uniquely; internally, for the system, the control point which the file is assigned to is also necessary to identify users' private files (different users may have the same names for different files). When a user defines a file, assigning it a name, he can also specify the actual file medium. The files not assigned specifically to a particular medium are assumed to be disk files. In a FORTRAN program, for example, all input/output statements refer to disk files unless explicitly assigned to another medium by the user. So the statements originally intended for tape operations, like READ(u,n), WRITE(u), and so on, actually refer to disk files, the names of which are TAPE plus the unit number specified in the statement (TAPE1, TAPE2, and so on). The statements READ, PRINT, and PUNCH refer to (disk) files with special names (INPUT, ØUTPUT, PUNCH) which will be examined extensively in the following description.

The assignment of a physical unit to a file is controlled by job control cards and does not depend on the central processor coding. Any device under the system control can be assigned to a file. How this is actually performed will not be examined in this paper.

Some names like INPUT, ØUTPUT, PUNCH are reserved by the system for special files. They cannot be adopted by the users for other purposes. The following description will clear the meaning of INPUT and ØUTPUT files, the extension to PUNCH files being quite obvious.

Programs being executed under COS cannot refer to the card reader to read input data, nor can they refer to a line printer to output results on-line*). Instead, the cards constituting the job are transmitted to a section of the disk before the program starts execution. The card-to-disk conversion is done automatically under control of COS while other jobs may be in execution. The section of disk on which the job has been recorded is logically similar to a magnetic tape file. It is called the INPUT file for the job.

Results to be printed are sent to another section of the disk called the ØUTPUT file of the job, and when the job has been terminated the information on this file will be printed automatically under control of COS as soon as a line printer becomes available.

Our FORTRAN program refers to the INPUT file by means of READ statements and to the ØUTPUT file by means of PRINT statements. Equating the file TAPE3 to the INPUT file has the effect that all the statements using TAPE3 will actually refer to the INPUT file.

The INPUT and ØUTPUT files, as all other files, are logically similar to magnetic tape files. A pointer is associated with each file which takes the place of the read/write heads of the magnetic tape files. Read and write operations referring to a disk file move the pointer as they transmit data to or from the file. The information in COS files is subdivided into <u>logical records</u>. In the case of our job INPUT file, there is one logical record per group of cards terminated by an EØR card.

For each file defined in the system, COS maintains an entry in a special table (kept in CMR), the <u>File Table</u>. The entry contains various types of information about the file such as: the file name and type, the equipment number, and the control point number to which the file is assigned, if any. For disk files it also contains the starting address and the current position of the above pointer. The type of file is determined by its use. When COS transmits a job from the card reader to the disk it generates a file of <u>type input</u>. The files waiting to be printed are of <u>type output</u>. The files of type input and those of type output constitute two stacks of files waiting to be processed by special system peripheral processor programs. These programs treat the <u>stack</u> elements <u>as data</u> and select them in the order determined by their priority. The programs are: The <u>Begin Job</u> (1BJ) routine which selects the next job to be executed and the PRINT routine which handles the printing of the elements in the other stack. The 1BJ routine will be described in more detail shortly; before that, two other types of files have to be mentioned: the files of <u>type local</u> and <u>type common</u>. The files of type local are files assigned to a particular job and dropped at the end of the job. They have the control point number registered in the File Table entry corresponding to the file. Local to the job are, for example, the files INPUT and ØUTPUT. The files of type local are discarded at the completion of the job (INPUT and all other files except PUNCH and PRINT), or when they have been output (PRINT, PUNCH). The files of type common can be transmitted from one job to another. While they are assigned to a job, they have the control point number of the job registered in the File Table entry and are not accessible to any other job. At the end of the job they are not dropped, unless explicitly requested by a job control card (RELEASE). In our example, the files FILE1 and FILE2 are both defined to be common at the beginning of the job. They are supposed to be prepared previously by another job. At the

*) This is not true in principle, as any equipment can be assigned to any job file. Under <u>normal</u> circumstances it may nevertheless be assumed that no job is <u>allowed</u> to get <u>hold</u> of a printer or card reader for private use, which would lock out the other users.

end of the job the file FILE1 remains in the system as common file and can be used successively by other jobs while the file FILE2 is dropped.

Let us now return to the initiation of our job. When a control point becomes free, COS loads the scheduling program 1BJ (Begin Next Job) into a PP, and assigns it to that control point. 1BJ scans the File Table, looking through the stack of input files, and selects an input file with the highest priority. Eventually our job will be selected. It is, in fact, registered as a file of type input, generated when the job deck was input from the card reader. The name of the job file is the job name taken from the job card (JØB in our case) completed with an integer giving the sequence number of the job in the series of jobs presented since the system dead start. If our job was presented as the 711th job, the job file name would be JØB711.

During the first initialization process, 1BJ reads the job file, up to the first EØR card, into a section of the control point area and leaves the file (pointer) positioned in front of the next record, i.e. in front of the FORTRAN program. It also sets a pointer, in the same area, to the first control statement. At this stage the central memory space, requested by the job on the job card, is assigned to the job. If the job card does not specify a central memory request, as in our case, a fixed number of central memory words is assigned by the system. It should be noted that the job area thus assigned is distinct from the control point area of the job. While the control point area contains information requested by COS to control the job, the job area is used to hold the programs which are required by the job. An erroneous program may invalidate the information in its job area, but it cannot write into its control point area.

When 1BJ assigns the job to the control point (the same to which 1BJ itself is assigned), it changes the file name from JØB711 to INPUT and the file type from input to local. The file becomes assigned to the control point: the control point is recorded in the File Table entry for the file in order to distinguish it from the files assigned to other control points.

When 1BJ has initiated the job COS calls on another system routine, 2TS (Translate Statements) to translate the control statements. These are translated one at a time and in the order presented in the card deck. The first statement translated is in our case the CØMMØN statement.

There are two kinds of control statements available under COS: execution control statements and program call control statements. Referring to our job: CØMMØN and RELEASE are statements of the first type; REWIND, RUN, and LGØ are of the second type. In general, the execution control statements have fixed control symbols recognized directly by the system, while program call control statements have any symbols chosen by the users to designate their programs. In the case of our job, RUN and REWIND are two particular names designating the system FORTRAN compiler and a system routine used to position files at their start address. LGØ (load-and-go) is a file containing the binary version of the compiled program. The problem will be briefly discussed later; let us return to our job.

The first control statement translated by 2TS is CØMMØN and then CØMMØN again. For each CØMMØN statement, 2TS searches the File Table for a file with the name specified (in our case FILE1 and FILE2). If the file exists, if it is of type common and is not assigned to another job, then it is assigned to our job, until job termination when it is returned to common unassigned status. If the file is assigned to another job or does not have common status, our job must wait until the file becomes available. Finally, if the file were (this is not our case) a local file for our job, it would become a common file and would not be deleted at the end of the job.

The next two statements (REWIND) are self-explanatory: the file is positioned at the very beginning. REWIND is the name of a system program and the statement is the call for that program. FILE1 and FILE2 are arguments transmitted to the program. A program called by a control statement may, in fact, refer to one or more program parameters. These parameters are placed at the very beginning of the job area. The parameters of a FORTRAN program are names of files and are specified in the PRØGRAM statement. In our case, the parameters (files) specified are: INPUT, ØUTPUT, FILE1, FILE2. The parameters may be equated to other parameters. Two parameters equated are equivalent within the program. In our case, any operation on TAPE3 will, in fact, concern the file INPUT, and any operation on TAPE1 (TAPE2) will concern FILE1 (FILE2). When a program execution is initiated by a program call statement, the call may specify one or more arguments. These are transmitted verbatim to the program called and replace the original arguments. In our example, FILE1 (FILE2) in the REWIND statement replaces the original parameter of the REWIND program which acts so precisely on FILE1 (FILE2). The same happens with the next control statement. RUN(S) initiates the system compiler replacing the first parameter of this program by the identifier S. The compiler, in fact, uses several parameters to allow the user to specify different compiler options, such as: compile and execute, compile and punch the binary version of the program, compile with source and object list and do not execute, and so on. Our call changes the first parameter to S and leaves the other parameters unaltered. Under those conditions, RUN expects a FORTRAN source program on the INPUT file, stores the listing of this program on the file ØUTPUT, and writes the binary version of the program in the file LGØ.

Note that all input and output for all jobs is controlled by COS. Each I/Ø request submitted to COS specifies the name of the file involved in the I/Ø operation. When COS receives a write request for a file whose name is as yet undefined, it will automatically generate a disk file which is of type local and has the specified name. Thus, while RUN is executed as part of our job, local files ØUTPUT and LGØ are generated automatically. These files are not rewound at the end of the compilation, i.e. when the compiler input reaches the second EØR card on the file INPUT. The INPUT file is so positioned at the end of our compilation that the next read operation will obtain the data cards.

2TS is now called again to process the next control card, i.e. LGØ. Like RUN, LGØ is not recognized as an execution control statement by 2TS and is therefore taken as a program call.

Program calls may be either calls of a user program or calls of a library program. To distinguish these two cases, the files associated with the control point are searched first. If a file is found whose name is the name of the specified program, then COS rewinds that file and reads it into the job area. In our example, the call of LGØ effectively loads the binary program which the RUN compiler has written to file LGØ beforehand.

This program, when executing READ statements, produces an input request to file INPUT and thus finds the data cards at the end of our job. Execution of PRINT statements produces an output request to file ØUTPUT and thus adds the programs' results after the listing of the source program which RUN had written onto ØUTPUT.

The next call of 2TS makes the file FILE2 be released: when the job is terminated this file is deleted as any other local file.

When 2TS is now called again, it finds that all control statements of our job have been executed and therefore calls upon 1AJ, a system program used to terminate a job. 1AJ deletes the local files INPUT and LGØ from the File Table, whereas ØUTPUT (by virtue of its name) becomes an unassigned file of type output whose name is changed to JØB711. This file will therefore be printed on a line printer under control of COS. Printing terminates with a short series of messages which specify the control statements recognized by COS, and the central processor and peripheral processor execution times of the job. These messages are called the dayfile messages of the job.

PART II

MATHEMATICAL TOPICS

<u>MINIMIZATION AND CURVE FITTING</u>

by

G.C. Sheppey

<u>CONTENTS</u>

1. INTRODUCTION

The usual problem at CERN is to find the minimum with respect to x, a column vector of n components, of

$$F(x) = \sum_{i=1}^{N} \left[f^{(i)}(x) \right]^2 .$$

One of the methods to be given will be specifically for functions of this type, but the others will be applicable to general functions of many variables.

It should be noted that when all the $f^{(i)}(x)$ are linear functions of x (i.e. when $\partial f^{(i)}/\partial x_j$ = const) the problem, in general, has a unique solution, the calculation of which involves a matrix inversion. Even if the matrix to be inverted is ill-conditioned, an attempt at a direct solution is much to be preferred to the use of an iterative method suitable for non-linear problems, since any such method is likely to converge very slowly if the problem is ill-conditioned.

2. SIMPLE METHODS

The most obvious method of finding a minimum of a function of n variables is what is known as the method of coordinate variation. A minimum is found with respect to each coordinate in turn, repeating with x_1 after x_n. This is illustrated in two dimensions in Fig. 1 which is a plot of the contours of the function to be minimized. Starting from the point A_0, successive minima along lines parallel to the coordinate axes are found at A_1, A_2, etc. This illustrates the fact that this method requires a great many steps to converge when used on functions with large correlation between the variables.

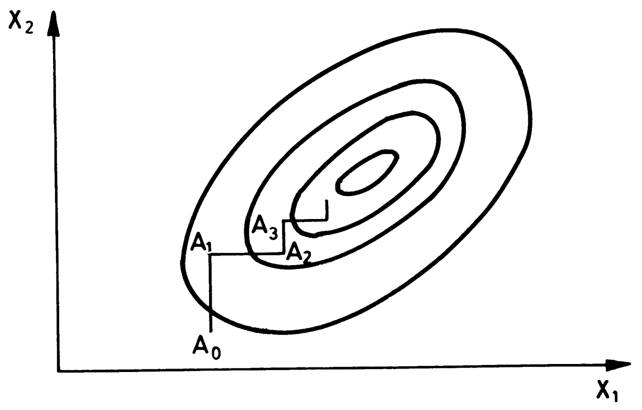


FIG.1

If $\partial F/\partial x_j$, the derivatives of F(x), can be calculated as well as F(x), the method of steepest descent can be used. This method is intrinsically attractive since it chooses the direction from the starting point in which the function decreases most rapidly. The minimum is found in this direction and the process repeated.

SIS/kw/sb

The direction of steepest descent, ζ, is defined by

$$\zeta_i = \frac{\partial F}{\partial x_i} \Big/ \left[\sum_{i=1}^{n} \left\{ \frac{\partial F}{\partial x_i} \right\}^2 \right]^{1/2} .$$

Thus if x is the minimum point along a direction η, and ζ is the direction of steepest descent from x, then

$$\zeta' \eta = \sum_{i=1}^{n} \eta_i \zeta_i = 0$$

since x is the minimum point along η. Hence ζ and η, which are successive directions of search, are perpendicular. This is illustrated in two dimensions in Fig. 2, and it is clear that, in two dimensions, this method is equivalent to the method of coordinate variation apart from a transformation of axes. In many dimensions the two methods are not equivalent, but experience shows that, when there is strong correlation between the variables, the method of steepest descent converges only very slowly.

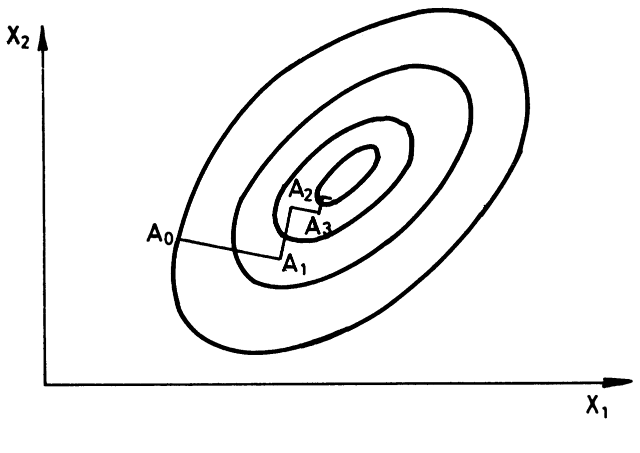


FIG. 2

3. ROSENBROCK'S METHOD

The method to be given here is a slight variation of that given in the paper by Rosenbrock[1)], and is an extension of the method of coordinate variation. It does not require the calculation of derivatives.

Suppose that in Fig. 1 the point A_2 has been reached, then the principle of this method is that the best direction in which to search next is along the line joining A_0 to A_2. A minimum is found along this line, then in the direction perpendicular, and the process is repeated with a new 'best' direction.

In n dimensions the procedure is as follows.

Let $\zeta^{(i)}$ $(i = 1, \ldots, n)$ be n orthogonal directions. Find the minimum of the function in each of these directions in turn and suppose that d_i is the size of the step taken to reach the minimum in the direction $\zeta^{(i)}$.

Let

$$a^{(i)} = \sum_{j=i}^{n} d_j \zeta^{(j)} \qquad (j = 1, \ldots, n),$$

i.e. $a^{(1)}$ is the vector joining the initial and final points, $a^{(2)}$ is the sum of all steps taken except the first, etc.

New directions $\zeta^{(1)}$ are then defined as follows:

$$\zeta^{(1)} = a^{(1)} \Big/ |a^{(1)}|$$

$$\left.\begin{aligned} b^{(i)} &= a^{(i)} - \sum_{j=1}^{i-1} \left(a^{(j)}\zeta^{(j)}\right) \zeta^{(j)} \\ \zeta^{(i)} &= b^{(i)} \Big/ |b^{(i)}| \end{aligned}\right\} \quad i = 2, \ldots, n .$$

The procedure is then repeated with the new set of orthogonal directions $\zeta^{(i)}$.

While this method has no real theoretical background, in practice it has been found very useful.

Finding the minimum along a line

Starting from a point x take a step of length s in the direction ζ, i.e. evaluate $F(x + s\zeta)$.

If $F(x + s\zeta) \leq F(x)$ the trial is a success, replace x by $(x + s\zeta)$, and s by αs, where $\alpha > 1$.

If $F(x + s\zeta) > F(x)$ the trial is a failure, replace s by βs, where $-1 < \beta < 0$.

Repeat this procedure until there is a failure immediately preceded by a success. This will ensure that a parabola fitted through the last three points will have a minimum (not a maximum), and that this minimum will be between the first and third of these points.

4. CONJUGATE DIRECTION METHODS

These methods have a broader theoretical background based on the minimization of a quadratic form. The justification for using this as a basis is that any analytic function will approximate to a quadratic form sufficiently near a minimum. Powell[2] gives a method which does not require the calculation of derivatives of the function to be minimized, and Fletcher and Powell[3] give a method which uses these derivatives.

Suppose that the function to be minimized is the quadratic from

$$F(x) = F_0 + a'x + \frac{1}{2} x'Gx ,$$

where a is a column vector and G the $n \times n$ matrix of second derivatives.

Then p and q are defined to be conjugate directions if $p'Gq = 0$.

Suppose now that $q_1, \dots, q_m$ $(m \leq n)$ are m mutually conjugate directions, then any point in the space defined by the point x_0 and these m directions can be represented by

$$x = x_0 + \sum_{i=1}^{m} \alpha_i q_i$$

where $\alpha_1, \dots, \alpha_m$ are m scalars. Also

$$F(x) = F(x_0) + \sum_{i=1}^{m} \left\{ \alpha_i q_i'(a + Gx_0) + \frac{1}{2} \alpha_i^2 q_i' G q_i \right\}$$

since $q_i' G q_j = 0$ for $i \neq j$.

Now, the term in each α_i in this expression is independent of the other α_j; therefore the minimum in the space defined by x_0 and q_i $(i = 1, \dots, m)$ can be found by the estimation of each α_i, for the minimum point, in turn, i.e. by a search in each of the directions q_i in turn.

It should also be noted that if two different points x_0 and x_1 are both minimum points of $F(x)$ in the same direction q, then

$$q'(a + Gx_0) = 0$$

and

$$q'(a + Gx_1) = 0$$

therefore

$$q'G(x_0 - x_1) = 0 ,$$

i.e. q and $(x_0 - x_1)$ are conjugate directions.

This is the basis of the method given in the paper by Fletcher and Powell[3)] of which an iteration is as follows:

Given n linearly independent directions ζ_i and starting from a point p_0,

for $r = 1, 2, \dots, n$ calculate λ_r so that $F(p_{r-1} + \lambda_r \zeta_r)$ is a minimum and define $p_r = p_{r-1} + \lambda_r \zeta_r$;

for $r = 1, \dots, (n - 1)$ replace ζ_r by ζ_{r+1};

replace ζ_n by $(p_n - p_0)$;

find λ so that $F(p_n + \lambda\zeta_n)$ is a minimum, replace p_0 by $(p_n + \lambda\zeta_n)$ and repeat the iteration.

It can be shown by induction[2)] that after n such iterations the directions ζ_i ($i = 1, \ldots, n$) will be mutually conjugate, and that the point p_n of the last iteration will be the minimum of the quadratic form.

However, in practice the procedure as given can run into trouble as it may choose directions which are very nearly dependent. In any case, the choice of the direction ζ_1 as the one to be dropped is arbitrary, and a modification should be made to the basic procedure. The basis of this modification is the fact that if the directions ζ_i ($i = 1,\ldots,n$) are scaled so that $\zeta_i' G \zeta_i = 1$ ($i = 1,\ldots,n$) then the determinant whose columns are the n vectors ζ_i has its maximum value when these vectors are mutually conjugate (for proof see Ref. 2). Therefore, in the iteration procedure, the new direction $(p_n - p_o)$ is used if it will increase det $(\zeta_1, \ldots, \zeta_n)$, and replaces ζ_i such that the determinant is maximized.

Supposing that $(p_n - p_o) = \alpha_1\zeta_1 + \alpha_2\zeta_2 + \ldots + \alpha_n\zeta_n$ and $(p_n - p_o) = \mu\zeta_p$ where $\zeta_p G \zeta_p = 1$, the replacement of ζ_j by ζ_p will multiply det $(\zeta_1, \ldots, \zeta_n)$ by the factor α_j/μ.

Now

$$\alpha_j = \sqrt{2\left(f(p_{n-1}) - f(p_n)\right)} \qquad \text{(see Ref. 2)}$$

and μ can be calculated by fitting a parabola through the function values $F(p_o)$, $F(p_n)$, and $F(2p_n - p_o)$. The direction ζ_j to be replaced is chosen to make α_j a maximum and is replaced by ζ_p only if $\alpha_j > \mu$.

<u>Use of derivatives</u>

If the derivatives of the function to be minimized can be calculated, the method given in Ref. 3 can be used.

Let g be the vector of derivatives of the quadratic form $F(x)$, then

$$g = a + Gx \text{ .}$$

Therefore if x_m is the minimum point

$$x_m = -G^{-1}a = -G^{-1}g + x \text{ ,}$$

that is

$$x_m - x = -G^{-1}g \text{ ,}$$

i.e. the direction in which to search for a minimum is given by $\zeta = -G^{-1}g$. This method therefore assumes an estimate H of the matrix G^{-1}, and given the derivative vector g it searches for a minimum in the direction $\zeta = -Hg$. Having found the minimum, an improvement is made to the estimate H of G^{-1}.

Suppose that, starting from the point x_i, a minimum is found in the direction ζ_i at $x_{i+1} = x_i + \alpha_i\zeta_i$.

Let the derivative vector at x_i be g_i; let

$$y_i = g_{i+1} - g_i$$

and let

$$\sigma_i = \alpha_i \zeta_i = x_{i+1} - x_i$$

then put

$$H_{i+1} = H_i + \frac{\sigma_i \sigma_i'}{\sigma_i' y_i} - \frac{H_i y_i y_i' H_i}{y_i' H_i y_i} .$$

In Ref. 3 it is proved that after n such steps $H_n = G^{-1}$ and x_n is the minimum point of the quadratic form, σ_i (i = 1, ..., n) being mutually conjugate directions.

Finding a minimum along a line

In the case where derivatives are available, cubic interpolation can be used given function values and derivatives at two points. Details of this are given in Ref. 3.

5. MINIMIZING SUMS OF SQUARES

Let the function to be minimized be given by

$$F(x) = \sum_{k=1}^{m} \left[f^{(k)}(x) \right]^2$$

where x is a column vector of n components and $m \geq n$.

The method given in the paper by Powell[4)] is based on the linearization of the problem. Let

$$g_i^{(k)}(x) = \frac{\partial}{\partial x_i} f^{(k)}(x) \qquad (i = 1, \ldots, n)$$
$$(k = 1, \ldots, m)$$

Then for a small δ_i (i = 1, ..., n)

$$F(x + \delta) \approx \sum_{k=1}^{m} \left[f^{(k)}(x) + \sum_{i=1}^{n} \delta_i g_i^{(k)}(x) \right]^2 ,$$

the minimum of which is given by the solution of

$$\sum_{j=1}^{n} \left\{ \sum_{k=1}^{m} g_i^{(k)}(x) g_j^{(k)}(x) \right\} \delta_j = -\sum_{k=1}^{m} g_i^{(k)}(x) f^{(k)}(x) \qquad (i = 1, \ldots, n) .$$

Thus starting from a point x, given function values and estimates of the derivatives, a correction δ to x is calculated. Then $F(x + \lambda\delta)$ is minimized with respect to λ. At this point the function values calculated during this minimization along a line are used to correct the estimated derivatives, and then the procedure is repeated. Thus the derivatives have to be estimated by finite differences (thus using extra function evaluations) only once at the beginning. Full details of the procedure are given in Ref. 4.

6. COMPARISON

The papers by Box[5] and Fletcher[6] give some comparisons between the methods given here and also some others.

Their main conclusion is that for a least squares problem a method designed for this particular case [e.g. Powell's method[4]] is usually better than a general minimization method. For the general problem it is usually worth while to calculate derivatives and use the method of Fletcher and Powell[3]. If this is not possible Box[5] considers Powell's method[2] superior, but Fletcher[6] says that, with an increasing number of variables, this method is less favourable in comparison with Rosenbrock's[1].

* * *

REFERENCES

1) H.H. Rosenbrock, "An automatic method for finding the greatest or least value of a function", Comput.J. 3, 175 (1960).

2) M.J.D. Powell, "An efficient method for finding the minimum of a function of several variables without calculating derivatives", Comput.J. 7, 155 (1964).

3) R. Fletcher and M.J.D. Powell, "A rapidly convergent descent method for minimization", Comput.J. 6, 163 (1963).

4) M.J.D. Powell, "A method for minimizing a sum of squares of non-linear functions without calculating derivatives", Comput.J. 7, 303 (1965).

5) M.J. Box, "A comparison of several current optimization methods", Comput.J. 9, 67 (1966).

6) R. Fletcher, "Function minimization without evaluating derivatives -- a review", Comput.J. 8, 33 (1965).

MATRIX MANIPULATION TECHNIQUES

by

G.A. Erskine

CONTENTS

1. INTRODUCTION

1.1 Matrix storage conventions in FORTRAN

Let A be a matrix of m rows and n columns:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & & \cdots & \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} .$$

The most natural way to store A is to set a_{IJ} = A(I,J) where A is a two-dimensional FORTRAN array with declared dimensions not less than m and n. If A(1,1) is stored at address α within the computer, we have

$$\{\text{Address of } A(I,J)\} = \alpha + (I-1) + M*(J-1) . \qquad (1)$$

The convention a_{IJ} = A(I,J) is not always the most convenient one. For example, if A is large and symmetric it might be better to store only the upper or lower triangle of A with rows (or columns) packed end-to-end.

Subroutines for manipulating matrices will normally be written so as to accept matrices of arbitrary size. This means that the subroutine must have a DIMENSION statement containing formal parameters. For example

```
SUBROUTINE MATOP(A,B,M,N)
DIMENSION A(M,N),B(M,N)
...
...
```

The subroutine then assumes that it will find a_{IJ} in the address given by formula (1). This will be the case only if the values of M and N at the time when the subroutine is called are equal to the dimensions appearing in the DIMENSION statement of any other program or subroutine referring to the matrices A and B. In order to avoid this limitation we can supply the declared dimensions as additional parameters to the subroutine:

```
  SUBROUTINE MATOP(A,B,M,N,M1,N1)
  DIMENSION A(M1,N1),B(M1,N1)
  DO 1 I=1,M
  DO 1 J=1,N
1 B(I,J)  = FUNCTION(A(I,J))
  RETURN
  END
```

M1 and N1 are here the declared dimension of the arrays A and B in the routine which calls MATOP; M and N are the "mathematical" dimensions of the corresponding matrices. In fact, as can be seen from formula (1), the column dimension is irrelevant, and N1 can be omitted and replaced by 1 (or any other integer) in the DIMENSION statement of the subroutine.

SIS/kw/hsi

We therefore have three possible conventions for transmitting dimensions:

i) transmit mathematical dimensions M and N only (these must then also be the declared dimensions in other programs referring to the same matrix);

ii) transmit the declared row dimension M1 as well as M and N;

iii) transmit the declared row and column dimensions as well as M and N (although the column dimension is redundant).

1.2 Elementary operations

The basic matrix operations of addition, subtraction, multiplication, multiplication by transposed matrix, etc., are available in two packages: the GRIND library package (permanently stored on the CDC 6600 disc), and the CDC MATRIX package. The function calls for both these packages are listed in Tables 1 and 2. Each package provides some operations not provided by the other. All the CDC MATRIX routines are written in machine language (ASCENT) and most of the GRIND routines are also in ASCENT.

2. SIMULTANEOUS LINEAR EQUATIONS, MATRIX INVERSION

If A is a non-singular square matrix of order n, and if $\underline{x}$ and $\underline{b}$ denote column vectors, where $\underline{b}$ is known, the system of simultaneous linear equations

$$A\underline{x} = \underline{b} \qquad (2)$$

has a unique solution. If the solution is required for k different sets of right-hand sides $\underline{b}_1, \underline{b}_2, \ldots, \underline{b}_k$, and if $\underline{x}_1, \underline{x}_2, \ldots, \underline{x}_k$ are the corresponding solution vectors, we have

$$AX = B, \qquad (3)$$

where X is a matrix whose columns are $\underline{x}_1, \underline{x}_2, \ldots, \underline{x}_k$ and B is a matrix whose columns are $\underline{b}_1, \underline{b}_2, \ldots, \underline{b}_k$. If B is set equal to the n^{th} order unit matrix I, the solution of Eq. (3) is $X = A^{-1}$. Therefore any procedure for solving sets of simultaneous linear equations may be used to find the inverse of a matrix. Conversely, if the inverse A^{-1} of A can be calculated, the solution of Eq. (2) is given by $\underline{x} = A^{-1}\,\underline{b}$. However, the calculation of A^{-1} is always slower than the direct solution of Eq. (2).

We assume throughout that A is a non-singular matrix. The probability that a matrix , whose elements are subject to round-off error should have a zero determinant is negligible, but it is fairly common for matrices to have two or more rows or columns which are nearly linearly dependent, resulting in a near-zero determinant. For such matrices, small perturbations in the matrix elements (for example, round-off errors) can cause much larger perturbations in the components of the solution vector $\underline{x}$, and special precautions are needed in the numerical solution. Matrices of this kind are said to be ill-conditioned. There is an extensive literature on the error analysis of methods for the solution of linear equations, with particular reference to ill-conditioned matrices. It should be noted, however, that the solution of an ill-conditioned system of equations with rounded coefficients is inherently poorly determined, and that a big effort to improve the accuracy of the solution (for example, by using double precision arithmetic) may be misplaced.

We describe below only a few of the large number of numerical methods for solving linear equations or inverting matrices.

2.1 Gaussian elimination

One of the simplest methods for solving the system of Eqs. (2) is that of successive elimination of the variables, usually called Gaussian elimination. By subtracting appropriate multiples of the first equation from each of the remaining equations, the coefficient of x_1 may be reduced to zero in the remaining equations. The system of equations is written in full below, with the appropriate multiplier enclosed in parenthesis on the left of each equation:

$$\left.\begin{array}{rl} & a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ (\ell_{21} = a_{21}/a_{11}) & a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ (\ell_{31} = a_{31}/a_{11}) & a_{31}x_1 + a_{32}x_2 + \dots + a_{3n}x_n = b_3 \\ \dots & \dots \quad \dots \quad \dots \\ \dots & \dots \quad \dots \quad \dots \\ (\ell_{n1} = a_{n1}/a_{11}) & a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n = b_n \end{array}\right\} . \qquad (4)$$

After the subtractions have been performed, the first column of the new matrix of coefficients will have zeros in all positions except the first. Thus the second equation is now of the form

$$u_{22}x_2 + u_{23}x_3 + \dots + u_{2n}x_n = c_2 .$$

The next stage is to subtract multiples of this equation from the 3rd, 4th, ..., nth of the new equations so as to reduce the coefficient of x_2 to zero in these equations. The appropriate multipliers are $\ell_{i2} = u_{i2}/u_{22}$ $(i = 3, 4, \dots, n)$. At each stage the new coefficients can be written on top of the old ones, which are no longer required. Continuing in this way we eventually arrive at the following triangular system of equations:

$$\left.\begin{array}{r} u_{11}x_1 + u_{12}x_2 + u_{13}x_3 + \dots + u_{1n}x_n = c_1 \\ u_{22}x_2 + u_{23}x_3 + \dots + u_{2n}x_n = c_2 \\ u_{33}x_3 + \dots + u_{3n}x_n = c_3 \\ \dots \quad \dots \quad \dots \\ \dots \quad \dots \\ u_{nn}x_n = c_n \end{array}\right\} . \qquad (5)$$

These equations are solved by <u>back-substitution</u>: x_n is obtained from the last equation, x_{n-1} from the $(n-1)^{th}$ equation, and so on. The operations which reduce (4) to the triangular form (5) do not change the value of the determinant of A, which is therefore equal to the product of diagonal elements $u_{11}u_{22} \dots u_{nn}$.

2.2 Pivoting strategies

The diagonal elements u_{ii} by which we must divide in order to form the multipliers ℓ_{ij} are called the <u>pivots</u>, and the equation from which the pivot is chosen (and which remains unchanged in subsequent stages of the reduction) is called the pivotal equation. The procedure described above, in which the pivots are chosen in the order u_{11}, u_{22}, ..., is called sequential pivoting.

Sequential pivoting breaks down if any of the u_{ii} is zero. This can happen for perfectly well-behaved matrices; for example for

$$A = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.$$

However, at any stage in the reduction, it is not possible for all the coefficients of the variable being eliminated to become zero, since this would mean that the matrix was singular. At the i^{th} stage of the reduction we may therefore select some equation for which a_{ri} $(r \geq i)$ is not zero, and interchange this equation with the i^{th} equation in the store of the computer so that it becomes the pivotal equation. It is natural to choose as the equation for interchange the equation in which the coefficient of x_i has the largest modulus. This not only avoids the possibility of having a zero pivot, but also makes it less likely that the selected pivot is one which has lost a large number of significant digits through cancellation. This pivoting strategy, in which the i^{th} pivot is chosen from the leading column of the current reduced matrix, is called partial pivoting.

It is also possible to choose as pivot at the i^{th} stage of the reduction the coefficient of largest modulus in the whole of the square sub-matrix which remains to be reduced, instead of only its first column. This procedure is called complete pivoting. When the pivot has been chosen, the row containing it must be interchanged with the i^{th} row, and the column containing it must be interchanged with the i^{th} column. The column number r of each pivot must be recorded, so that when the back-substitution is complete, x_i and x_r can be interchanged. Experience has shown that total pivoting does not usually produce any large increase in accuracy as compared with partial pivoting. It takes appreciably longer than partial pivoting. We therefore make the following recommendation: when a program allows a choice between complete pivoting and partial pivoting, choose partial pivoting.

2.3 Equilibration

Some computer programs which employ complete or partial pivoting carry out a preliminary scaling of the rows and columns of the matrix so as to ensure that the largest element in each row and column has a given order of magnitude. On a binary computer, multiplication by powers of two can be done without introducing any round-off error, and the rows and columns can be scaled so that the element of largest modulus in any row or column lies between $\frac{1}{2}$ and 1. This process is called equilibration. Equilibration makes it more likely that coefficients which have lost many significant digits during the elimination will be small in magnitude and will not therefore be accepted as pivots. After the final back-substitution it is necessary to divide x_i by the multiplier used in scaling the i^{th} column.

2.4 Jordan elimination

In Gaussian elimination the pivotal equations are not used in subsequent stages of the elimination. The program therefore operates on successively smaller square sub-matrices, and the final result is a triangular system of equations consisting of the pivotal equations used at successive stages. An alternative procedure is to eliminate x_1 from all equations except the first as in Gaussian elimination, but then to eliminate x_2 from <u>all</u> equations except the second, including the equation which has just been used as a pivotal equation. In the same way x_3, x_4, ..., are eliminated from all equations except the 3rd, 4th, ..., equations respectively. This procedure is called Jordan elimination. It can be thought of as a version of Gaussian elimination in which the equations above the current pivotal equation are no longer exempt from manipulation. The final result is a diagonal system of equations of the form $v_{ii}x_i = c_i$ $(i = 1, 2, \ldots, n)$ from which each variable x_i can be calculated by a single division. As with Gaussian elimination, the modified coefficients can be written on top of the original coefficients in the store of the computer, and no additional storage space is required.

Sequential pivoting, partial pivoting, and complete pivoting may all be used. Since no back-substitution is required in Jordan elimination, the computer program is slightly simpler than for Gaussian elimination.

2.5 <u>Triangular factorization</u>

Suppose that a system of linear equations with matrix A has been reduced to the triangular form (5) by Gaussian elimination with sequential pivoting. If the multipliers used during the elimination are ℓ_{ij} and the coefficients in the final triangular system are u_{ij}, it can be shown that for any subscript pair (p,q)

$$a_{pq} = \ell_{p1}u_{1q} + \ell_{p2}u_{2q} + \cdots + \ell_{p,p-1}u_{p-1,q} + u_{pq} \,. \tag{6}$$

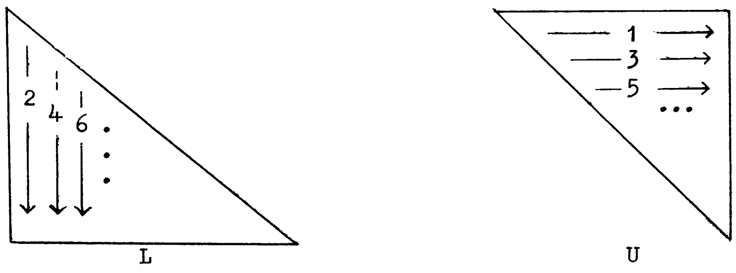
This is equivalent to

$$A = LU \,,$$

where

$$L = \begin{bmatrix} 1 & & & & 0 \\ \ell_{21} & 1 & & & \\ \ell_{31} & \ell_{32} & 1 & & \\ \cdots & & \cdots & 1 & \\ \ell_{n1} & \ell_{n2} & \ell_{n3} & \cdots & \ell_{n,n-1} \end{bmatrix}, \quad U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ & u_{22} & u_{23} & \cdots & u_{2n} \\ & & u_{33} & \cdots & u_{3n} \\ & & & \cdots & \\ 0 & & & & u_{nn} \end{bmatrix}.$$

Gaussian elimination therefore produces a triangular factorization of A. The left factor L is a lower triangle with units along the diagonal, and the right factor U is an upper triangle. It is possible to perform the factorization directly, using only the relation (6). If the rows of U and the columns of L are calculated alternately in the order shown in the diagram below, equation (6) determines the value of exactly one of the ℓ_{ij} or u_{ij}



As soon as each element of L or U is calculated it can be written on top of the element a_{pq} of A from which it was derived using formula (6). If suitably programmed, this method is not only mathematically equivalent to Gaussian elimination, but is arithmetically identical with it.

It can be seen from formula (6) that the currently unknown element of L or U is calculated from a sum of products of previously calculated elements. If this sum of products can be accumulated to double precision accuracy, round-off errors can be significantly reduced without having to provide double precision storage for n^2 numbers, as would be the case if Gaussian elimination were used.

To solve the equations $A\underline{x} = \underline{b}$, the vector $\underline{c}$ defined by $\underline{b} = L\underline{c}$ can be calculated at the same time as the calculation of the factorization A = LU. The equation $A\underline{x} = \underline{b}$ are then equivalent to $LU\underline{x} = L\underline{c}$, or $U\underline{x} = \underline{c}$ which is the matrix form of Eq. (5).

Partial pivoting can be used with triangular factorization, but there does not seem to be any way of carrying out complete pivoting. However, experience has shown that provided the sums of products are accumulated to double precision accuracy, triangular factorization with partial pivoting is usually more accurate than single precision Gaussian elimination with complete pivoting.

To calculate A^{-1} we first invert L. The inverse L^{-1} is also a lower triangle, and the calculation can be done on top of L itself. We can then find $X = A^{-1}$ by solving $UX = L^{-1}$.

2.6 Choleski factorization

If the matrix A is symmetric, the triangular factorization A = LU can be carried out by setting U equal to the transpose of L. We have, therefore

$$A = LL^T .$$

Partial or complete pivoting of the kind described above cannot be used. Also, as can be seen from formula (6) by setting $u_{ij} = \ell_{ji}$, the diagonal elements ℓ_{ii} must be calculated by taking a square root. The fact that A is symmetric does not guarantee that the diagonal elements are real and non-zero. This can, however, be guaranteed whenever A is positive definite, and the process is usually used only for positive definite matrices.

To calculate A^{-1} we can first invert L and then calculate $X = A^{-1}$ by solving $L^TX = L^{-1}$.

2.7 Operational counts

An approximate measure of the amount of work required by each of the above methods is given by the number of multiplications required to solve AX = B for a matrix B consisting of k right-hand sides.

For comparison, it is interesting to consider the number of multiplications involved in solving the system AX = B by first calculating A^{-1} and then $X = A^{-1}B$. We see that for a non-symmetric matrix A^{-1} requires at least n^3 multiplications. Since $A^{-1}B$ requires kn^2 multiplications, the total is at least $n^3 + kn^2$. Therefore, do not form A^{-1} when only the solution of simultaneous equations is required.

Method	Approximate No. of multiplications	
	$AX = B$	$AX = I$
Gauss (or Triangular Factorization)	$\frac{1}{3}n^3 + kn^2$	$\frac{4}{3}n^3$ *)
Jordan	$\frac{1}{2}n^3 + kn^2$	$\frac{3}{2}n^3$ *)
Choleski	$\frac{1}{6}n^3 + kn^2$	$\frac{1}{2}n^3$

*) The asterisk indicates that in these cases the number of multiplications can be reduced to n^3 if the program, at each stage of the reduction, by-passes the zeros which remain on the right-hand side.

The time in seconds taken by several CDC 6600 computer routines available at CERN is shown in Table 3. The source program language is indicated by F (FORTRAN) or A (ASCENT). FORTRAN compilations were carried out using SCOPE (version dated 29.1.1968).

2.8 Improvement of the solution

If $\underline{x}_0$ is an approximation to the exact solution $\underline{x}$ of $A\underline{x} = \underline{b}$, we shall have

$$\underline{b} - A\underline{x}_0 = \underline{r}_0$$

where the <u>residual vector $\underline{r}_0$</u> is unlikely to be zero. Hence, the error vector $\underline{x} - \underline{x}_0$ satisfies the equation

$$A(\underline{x} - \underline{x}_0) = \underline{b} - (\underline{b} - \underline{r}_0) = \underline{r}_0 \ .$$

We would therefore expect that the vector $\underline{x}_1 = \underline{x}_0 + \underline{\delta}_0$, where $\underline{\delta}_0$ is obtained by solving $A\underline{\delta}_0 = \underline{r}_0$, would be an improved approximation to $\underline{x}$. If necessary the process could be repeated. In practice it is necessary to use double precision accumulation of sums of products in forming the residual vector $\underline{r}_0$, but only single precision working is needed in solving $A\underline{\delta}_0 = \underline{r}_0$. Usually the triangular factors L and U of $A = LU$ are available from the calculation of $\underline{x}_0$. Therefore only n^2 multiplications are needed to solve for $\underline{\delta}_0$.

2.9 Iterative methods

The method of the preceding section leads to the following iteration

$$\left.\begin{aligned} \underline{r}_s &= \underline{b} - A\underline{x}_s \\ A\underline{\delta}_s &= \underline{r}_s \\ \underline{x}_{s+1} &= \underline{x}_s + \underline{\delta}_s \end{aligned}\right\} \quad (s = 0, 1, 2, \dots) \ .$$

Other iterative methods for solving $A\underline{x} = \underline{b}$ are known, and conditions for convergence can be established. The disadvantage of these methods is that convergence cannot usually be guaranteed for a general matrix A and an arbitrary starting vector. Their advantage is that, since they usually involve only multiplications by the original matrix A, full advantage can be taken of any zeros in A (which are destroyed by elimination methods). This is particularly important for the large but "sparse" matrices which occur as the finite difference analogues of partial differential equations. In addition to having many zeros, these matrices often have a repetitive structure which can be exploited in devising iteration methods, and there is an extensive literature on the iterative solution of systems of equations of this kind.

One iterative method which converges for any positive definite matrix A is the method of Conjugate Gradients. If there were no round-off errors this method would converge to the true solution of $A\underline{x} = \underline{b}$ in exactly n steps, each step requiring approximately $n^3 + 5n^2$ multiplications.

2.10 Complex matrices

The system of equations $A\underline{z} = \underline{b}$ where

$$A = B + iC, \quad \underline{b} = \underline{c} + i\underline{d}, \quad \underline{z} = \underline{x} + i\underline{y}$$

is equivalent to the two sets of equations

$$\left.\begin{aligned} B\underline{x} - C\underline{y} &= \underline{c} \\ C\underline{x} + B\underline{y} &= \underline{d} \end{aligned}\right\} .$$

These 2n equations involving real numbers may be solved by any of the methods described above.

3. EIGENVALUES AND EIGENVECTORS

Given a square nth order matrix A, the problem here is to find those numbers λ for which there exists one or more column vectors $\underline{u}$ such that

$$A\underline{u} = \lambda\underline{u} \ . \tag{7}$$

A number λ which satisfies this condition is an eigenvalue of A, and the associated vector $\underline{u}$ is an eigenvector corresponding to λ. Equation (7) is equivalent to $(A - \lambda I)\underline{u} = 0$. This system of homogeneous equations can have a solution only if the determinant of $(A - \lambda I)$ is zero. Therefore λ must satisfy the equation

$$\begin{aligned} \det(A - \lambda I) &= \alpha_0 + \alpha_1\lambda + \ldots + \alpha_{n-1}\lambda^{n-1} + (-1)^n\lambda^n \\ &= p_n(\lambda) \\ &= 0 \ . \end{aligned} \tag{8}$$

Equation (8) is the characteristic equation of the matrix A. It follows from Eq. (8) that the matrix A has n eigenvalues, some of which may be coincident. Also some of the λ's and $\underline{u}$'s may be complex even when A is real. For a general (non-symmetric) matrix A, the number of linearly independent eigenvectors may be less than n.

If X is an nth order non-singular square matrix, and if B is defined by

$$B = X^{-1}AX \ , \tag{9}$$

then B is similar to A. The matrices A and B have the same eigenvalues, and if $\underline{u}$ is an eigenvector of A then $X^{-1}\underline{u}$ is an eigenvector of B. Many of the numerical methods for calculating the eigenvalues of a matrix depend on reducing the matrix to a simpler form by means of similarity transformations. A particularly important subclass of similarity transformations are those for which the real transformation matrix U is <u>orthogonal</u>, defined by

$$U^TU = I \ .$$

Equation (9) then becomes

$$B = U^TAU \ .$$

The most direct approach to finding the eigenvalues of A is to calculate in some way the coefficients α_0, α_1, ... of the characteristic equation (8), and then to use one of the standard equation-solving techniques to find the roots λ. In this procedure there is a serious danger of losing significant digits when calculating the coefficients α_i from the elements of A. Further, even when the coefficients in Eq. (8) are accurately known, the problem is often ill-conditioned in that small perturbation in the coefficients α_0, α_1, ... can result in large changes to one or more of the roots λ. For this reason the direct calculation of the characteristic equation is usually recommended only for very small matrices or for certain classes of matrices for which this procedure is found to be numerically stable.

In the following we describe in brief outline a few of the large number of numerical methods which have been used for solving the eigenproblem (7).

3.1 <u>The power method</u>

Suppose that the eigenvalues of A have been arranged in order of decreasing modulus: $|\lambda_1| \geq |\lambda_2| \geq \ldots \geq |\lambda_n|$, and that there exists a corresponding set of linearly independent eigenvectors $\underline{u}_1$, $\underline{u}_2$, ..., $\underline{u}_n$ (this is always true for symmetric A, and is usually true for non-symmetric A). Then if $\underline{x}_0$ is an arbitrary vector, we can write

$$\underline{x}_0 = c_1\underline{u}_1 + c_2u_2 + \ldots + c_n\underline{u}_n$$

On multiplying $\underline{x}_0$ repeatedly by A we obtain the sequence of vectors

$$\begin{aligned} \underline{x}_r &= A\,\underline{x}_{r-1} = A^r\underline{x}_0 \\ &= c_1\lambda_1{}^r\underline{u}_1 + c_2\lambda_2{}^r u_2 + \ldots + c_n\lambda_n{}^r\underline{u}_n \\ &= \lambda_1{}^r\left[c_1\underline{u}_1 + c_2\left(\frac{\lambda_2}{\lambda_1}\right)^r \underline{u}_2 + \ldots + c_n\left(\frac{\lambda_n}{\lambda_1}\right)^r \underline{u}_n\right] \ . \end{aligned}$$

Therefore, if $|\lambda_1| > |\lambda_2|$, $\underline{x}_r$ approaches a multiple of $\underline{u}_1$ as r tends to infinity, and the ratio of corresponding components of $\underline{x}_r$ and $\underline{x}_{r-1}$ tends to the dominant eigenvalue λ_1. In practice it is convenient to divide each $\underline{x}_r$ by its component of largest modulus before multiplying by A. The remaining components of $\underline{x}_r$ and $\underline{x}_{r-1}$ can then be easily compared.

The rate of convergence of this process depends upon the ratio $|\lambda_2/\lambda_1|$, and will be slow if $|\lambda_2|$ is close to $|\lambda_1|$. The simplest way to increase the rate of convergence is to iterate with a matrix (A - pI) instead of A, where p is some number for which the ratio $|(\lambda_1 - p)/(\lambda_2 - p)|$ is not too close to unity. There are other devices for accelerating convergence, and the method can be adapted to handle equal or complex eigenvalues and complex eigenvectors. However, these special cases must be tested for during the course of the iteration, and such tests may be troublesome to programme.

Once an eigenvalue λ_1 and the corresponding vector $\underline{u}_1$ have been found, there exist methods of constructing a matrix of order $(n-1)$ which does not have λ_1 as an eigenvalue. This process is called deflation, and may be used in conjunction with any numerical technique which yields the eigenvalues and eigenvectors one at a time.

3.2 The real symmetric matrix

When the matrix A is real and symmetric the number of methods available for the solution of the eigenproblem is larger than for a general A, and it is easier to obtain accurate numerical results. The important properties of the symmetric matrix are:

i) all eigenvalues and eigenvectors are real;

ii) irrespective of whether or not the eigenvalues are all distinct, it is always possible to find n mutually orthogonal eigenvectors $\underline{u}_1, \underline{u}_2, \ldots, \underline{u}_n$ (i.e. $\underline{u}_i^T\underline{u}_j = 0$ if $i \neq j$). These vectors form a basis for the n-dimensional space.

If the eigenvectors $\underline{u}_i$ are normalized so that $\underline{u}_i^T\underline{u}_i = 1$, and if U is defined to be the nth order matrix whose columns are $\underline{u}_1, \underline{u}_2, \ldots, \underline{u}_n$, we have $U^TU = I$ (i.e. U is orthogonal) and

$$U^TAU = \begin{bmatrix} \lambda_1 & & & & 0 \\ & \lambda_2 & & & \\ & & \cdot & & \\ & & & \cdot & \\ 0 & & & & \lambda_n \end{bmatrix}$$

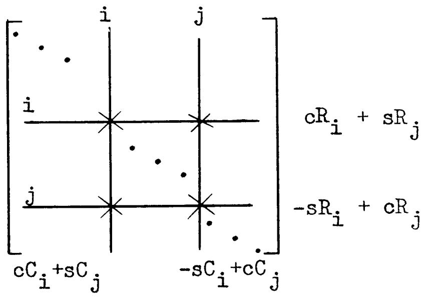
3.3 Jacobi iteration

Given an angle ϑ we may define a rotation matrix R(i,j) by

$$R(i,j)^T = \begin{array}{cccccccccc}
 & & & \text{Column } i & & & \text{Column } j & & & \\
 & & & \downarrow & & & \downarrow & & & \\
1 & & & & & & & & & \\
 & \ddots & & & & & & & & \\
 & & 1 & & & & & & & \\
 & & & \cos\vartheta & & & \sin\vartheta & & & \leftarrow \text{Row } i \\
 & & & & 1 & & & & & \\
 & & & & & \ddots & & & & \\
 & & & & & 1 & & & & \\
 & & & -\sin\vartheta & & & \cos\vartheta & & & \leftarrow \text{Row } j \\
 & & & & & & & 1 & & \\
 & & & & & & & & \ddots & \\
 & & & & & & & & & 1
\end{array}$$

This matrix differs from a unit matrix only in the rows and columns identified by i and j, and is orthogonal. To simplify the notation we write $c = \cos\vartheta$, $s = \sin\vartheta$, and we denote the ith row of A by R_i and the ith column by C_i. If A′ is obtained from A by using R(i,j) as a transformation matrix we have

$$A' = R(i,j)^T A\, R(i,j) = \qquad (10)$$



Rows and columns of A other than the ith and jth remain unchanged. Except for the four elements marked with a cross, the ith and jth rows and columns of A′ are simple linear combinations of the same two rows and columns of A as shown in formula (10). If A is symmetric, the two off-diagonal elements marked with a cross are given by

$$a_{ij}' = a_{ji}' = sc(a_{jj} - a_{ii}) + (c^2 - s^2)a_{ij} .$$

These elements can be made to vanish by choosing ϑ such that

$$\tan 2\vartheta = \frac{2sc}{c^2 - s^2} = \frac{2a_{ij}}{a_{ii} - a_{jj}}$$

with $|\vartheta| < \pi/4$.

It can be verified that the sum of the squares of the off-diagonal elements other than (i,j) and (j,i) is invariant under the transformation (10). Therefore if ϑ has been chosen so as to ensure that $a_{ij}' = a_{ji}' = 0$ we have

$$\sum_{p\neq q}\sum (a_{pq}')^2 - 0 = \sum_{p\neq q}\sum a_{pq}{}^2 - 2a_{ij}{}^2 . \qquad (11)$$

Thus the sum of squares of <u>all</u> off-diagonal elements is decreased by any transformation of type (10) which annihilates the (i,j) and (j,i) elements of the matrix.

The Jacobi method consists of repeatedly applying transformations of type (10) until all the off-diagonal elements have become negligible. We are then left with a diagonal matrix D obtained from A by an orthogonal transformation of the type

$$D = (R_k^T \ldots R_2^T R_1^T)\, A\, (R_1 R_2 \ldots R_k) .$$

The diagonal elements of D are the eigenvalues of A and the columns of the product matrix $R_k, \ldots, R_2R_1$ are eigenvectors of A. After each transformation, the elements which were set to zero by the preceding transformation become non-zero again. However, Eq. (11) shows that the sum of squares of the off-diagonal elements must decrease in spite of this, whereas the sum of squares of <u>all</u> the elements remains constant under the transformation (10). Therefore it is only necessary to prove that the limit of the decreasing sum (11) is in fact zero. This can be done for the following commonly used versions of the Jacobi process:

i) repeatedly annihilate a_{ij} over the upper triangle in cyclic order: (1,2), (1,3), ..., (1,n); (2,3), (2,4), ..., (2,n), etc;

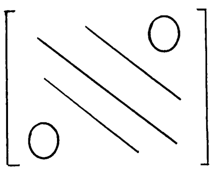
ii) at each step annihilate the off-diagonal element of largest modulus;

iii) annihilate only those elements which are larger than some fixed threshold value ϵ. Then reduce ϵ and repeat. Go on doing this until ϵ is small enough.

If full advantage is taken of symmetry, a complete sweep through all the off-diagonal elements of A requires approximately $2n^3$ multiplications and n^2 square roots.

3.4 Givens' method

This method uses the same transformations as the Jacobi method, but in such a way as to ensure that the zeros introduced by one transformation are not destroyed by the next. It is not possible to reduce a symmetric matrix A to diagonal form in this way, but A can be reduced to the tridiagonal form shown below:



If we consider the transformation of A by R(i,j) with $i < j$, we see from the transformation (10) that for elements of the j^{th} row other than those marked with a cross we have

$$a_{jr}' = -\sin\vartheta\, a_{ir} + \cos\vartheta\, a_{jr} \qquad (r \neq i,j) .$$

If A is symmetric, a'_{rj} has the same value as a'_{jr}, and these two elements can be annihilated by setting

$$\tan\vartheta = a_{jr}/a_{ir} ,$$

giving

$$\cos\vartheta = \frac{a_{ir}}{\sqrt{a_{ir}^2 + a_{jr}^2}} , \qquad \sin\vartheta = \frac{a_{jr}}{\sqrt{a_{ir}^2 + a_{jr}^2}} .$$

By using the transformation matrices R(2,3), R(2,4), R(2,5), ... , we can annihilate a_{13}, a_{14}, a_{15}, ... and (by symmetry) a_{31}, a_{41}, a_{51}, ... Thus all except the first two elements of the first row and the first column have been reduced to zero. Then, by using R(3,4), R(3,5), R(3,6), ... we can annihilate a_{24}, a_{25}, a_{26}, ... in the first row and a_{42}, a_{52}, a_{62}, ... in the second column. The important point is that this second sequence of transformations leaves unaltered the zeros introduced by the earlier transformations. (Each zero is replaced by a linear combination of two zeros.) By proceeding in this way we eventually arrive at a tridiagonal matrix.

If full advantage is taken of symmetry, the reduction to tridiagonal form requires approximately $(4/3)n^3$ multiplications and $(1/2)n^2$ square roots.

The calculation of the eigenvalues and eigenvectors of the tridiagonal form will be considered after Householder's method has been described in the next section. Once the eigenvectors of the tridiagonal form have been found, the eigenvectors of the original matrix can be calculated as in the Jacobi method.

3.5 Householder's method

This method is similar to that of Givens in that it reduces the symmetric matrix A to tridiagonal form by a finite sequence of orthogonal transformations. However, the transformation matrices are not the rotation matrices R(i,j) used in the Jacobi and Givens methods, but are symmetric matrices of the form

$$P = I - 2\underline{w}\underline{w}^T = \begin{bmatrix} 1-2w_1^2 & -2w_1w_2 & -2w_1w_3 & \cdots \\ -2w_2w_1 & 1-2w_2^2 & -2w_2w_3 & \cdots \\ -2w_3w_1 & -2w_3w_2 & 1-2w_3^2 & \\ \cdots & \cdots & & \ddots \\ & & & & 1-2w_n^2 \end{bmatrix}$$

For P to be orthogonal we require $P^TP = I$, and this is the case if $\underline{w}^T\underline{w} = w_1^2 + w_2^2 + \ldots + w_n^2 = 1$. Each step of the method consists of a transformation of the form $A_r = P_r^T A_{r-1} P_r$ where the numbers $w_1, w_2, \ldots, w_n$ which define P_r are chosen in such a way that the necessary set of zeros are introduced into one row and column of A_{r+1}. Thus, at the first step, setting $A_1 = A$, we require:

$$A_2 = P_2^T A_1 P_2 = \begin{bmatrix} a_{11}' & a_{21}' & 0 \ldots 0 \\ a_{12}' & \multicolumn{2}{c}{\text{▨}} \\ 0 & & \\ \vdots & & \\ 0 & & \end{bmatrix} \qquad (12)$$

This result can be achieved if the first component of $\underline{w}$ is set equal to zero, so that $\underline{w}^T$ is of the form $(0, w_2, w_3, \ldots, w_n)$. Equation (12) then determines $w_2, w_3, \ldots, w_n$. In fact $w_3, \ldots, w_n$ (but not w_2) are merely multiples of $a_{13}, \ldots, a_{1n}$, respectively. The next transformation $A_3 = P_3^T A_2 P_3$ is chosen so as to perform a similar transformation on the shaded sub-matrix in (12); the appropriate $\underline{w}^T$ is of the form $(0, 0, w_3, \ldots, w_n)$, which ensures that the first row and column of A_2 are not affected by the transformation. After a total of (n-2) transformations we obtain a tridiagonal matrix.

If full advantage is taken of symmetry, the reduction of A to tridiagonal form requires approximately $(2/3)n^3$ multiplications [cf. $(4/3)n^3$ for Givens' method] and n square roots [cf. $(1/2)n^2$ for Givens' method]. Both Givens' method and Householder's method are numerically stable if suitably programmed.

3.6 Calculating the eigenvalues of a tridiagonal matrix

Let us suppose that a matrix A has been reduced by a sequence of similarity transformations to some special form C, not necessarily the tridiagonal form. The eigenvalues of A are then the zeros of the nth degree polynomial

$$p_n(\lambda) = \det(C - \lambda I).$$

If there is a method by which the value of $p_n(\lambda)$ can be easily calculated for any given value of λ, the zeros of p_n can be obtained by means of one of the iterative root-finding methods which uses only functional values (for example, the Rule of False Position or the quadratic interpolation method of Muller). If the derivative p_n' can also be calculated, Newton's method can be used; if p_n'' can be calculated, Laguerre's method is available. The problem is therefore to find a convenient way of calculating $p_n(\lambda)$, and possibly one or more derivatives, for an arbitrary value of λ.

If C is a tridiagonal form (not necessarily symmetric) we have

$$p_n(\lambda) = \begin{vmatrix} \alpha_1-\lambda & \beta_2 & & & \\ \gamma_2 & \alpha_2-\lambda & \beta_3 & & \\ & \gamma_3 & \alpha_3-\lambda & \ddots & \\ & & \ddots & \ddots & \beta_n \\ & & & \gamma_n & \alpha_n-\lambda \end{vmatrix}$$

If $p_r(\lambda)$ denotes the leading principal minor of order r in this determinant, the expansion of $p_r(\lambda)$ by its last row yields the recurrence relation

$$\left.\begin{aligned} p_0(\lambda) &= 1 \\ p_1(\lambda) &= \alpha_1 - \lambda \\ p_r(\lambda) &= (\alpha_r - \lambda)\, p_{r-1}(\lambda) - \beta_r\gamma_r\, p_{r-2}(\lambda) \\ & \qquad\qquad (r = 2, 3, \ldots, n) \end{aligned}\right\} . \qquad (13)$$

Differentiating relation (13) yields recurrence relations of similar type for the derivatives of p_r. These relations permit the calculation of the quantities required by the various root-finding algorithms.

If the tridiagonal form C is symmetric, $\gamma_i = \beta_i$, and hence:

$$p_r(\lambda) = (\alpha_r - \lambda)p_{r-1}(\lambda) - \beta_r^2\, p_{r-2}(\lambda) \quad .$$

If none of the β_i is zero, it can be shown that there is exactly one zero of p_r between each of the necessarily distinct zeros of p_{r+1}. From this the following theorem can be proved: "For any number μ, the number of agreements in sign of consecutive members of the sequence $p_0(\mu)$, $p_1(\mu)$, ..., $p_n(\mu)$ is the number of eigenvalues of C which are strictly greater than μ".

This forms the basis of an effective method for localizing the eigenvalues of C by repeated bisection of an initial interval which is known to contain all the eigenvalues.

3.7 Calculating the eigenvectors of a tridiagonal matrix

Having obtained a close approximation λ to one of the eigenvalues of a symmetric tridiagonal matrix C, we wish to find a solution $\underline{u}$ to the homogeneous system of linear equations

$$(C - \lambda I)\underline{u} = 0 \ .$$

If $\underline{u}^T = (x_1, x_2, \ldots, x_n)$, these equations are

$$\left.\begin{array}{ll}(\alpha_1 - \lambda)x_1 + \beta_2 x_2 & = 0 \\ \beta_2 x_1 + (\alpha_2 - \lambda)x_2 + \beta_3 x_3 & = 0 \\ \quad\ddots\quad\ddots & \\ \beta_{n-1}x_{n-2} + (\alpha_{n-1} - \lambda)x_{n-1} + \beta_n x_n & = 0 \\ \beta_n x_{n-1} + (\alpha_n - \lambda)x_n & = 0\end{array}\right\}.$$

In principle a solution could be obtained by rejecting one of these equations -- say the i^{th} -- and solving the remaining (n-1) equations with x_i set equal to one. On substituting the solution so obtained back into the i^{th} equation we shall obtain some non-zero number δ, owing to round-off errors. The numbers $x_1, x_2, \ldots, x_n$ so obtained therefore satisfy the following system of equations:

$$\beta_j x_{j-1} + (\alpha_j - \lambda)x_j + \beta_{j+1}x_{j+1} = \begin{cases} 0 & \text{if } j \neq i \\ \delta & \text{if } j = i \end{cases} .$$

Since the eigenvector is independent of any arbitrary multiplier, we can consider this solution as being equivalent to the solution of the system of equations

$$(C - \lambda I)\underline{u} = \underline{e}_i \ , \tag{14}$$

where $\underline{e}_i$ is the i^{th} unit vector. Wilkinson has shown that, even when λ is very close to a true eigenvalue of c, the solution of Eq. (14) may be a poor approximation to the corresponding eigenvector. It is better to replace Eq. (14) by

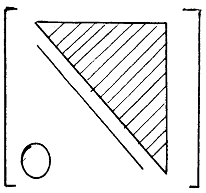
$$(C - \lambda I)\underline{u} = \underline{b} \ , \tag{15}$$

where $\underline{b}$ is an arbitrary vector, and to iterate one or more times using Eq. (15). Thus on the first iteration the solution $\underline{u}$ of Eq. (15) is used on the right-hand side instead of $\underline{b}$. Alternatively, it is possible to calculate the initial vector $\underline{b}$ in such a way as to make iterations beyond the first unnecessary in the majority of cases. This procedure is usually called <u>inverse iteration</u>.

3.8 <u>The eigenvalue problem for non-symmetric matrices</u>

The numerical calculation of eigenvalues and eigenvectors of non-symmetric matrices is more difficult than for symmetric matrices. Many mathematically elegant procedures lead either to disastrous numerical instability or to excessive amounts of calculation, or both. Even when this is not the case, it is often necessary to use double precision arithmetic at some stage of the calculation. The eigenvalues and eigenvectors may be complex, and it is no longer possible to reduce the amount of calculation by exploiting symmetry. However, a more serious difficulty in devising satisfactory procedures arises from the fact that one can no longer be sure that there exists a set of eigenvectors spanning the whole of the n-dimensional space.

The power method and its refinements may be used with non-symmetric matrices. An alternative approach is to use the orthogonal transformations which reduce a symmetric matrix to tridiagonal form, such as the methods of Givens and Householder. If applied to a non-symmetric real matrix these transformations produce a Hessenberg matrix, consisting of an upper or lower triangle bordered by a single diagonal; for example,



Other methods besides those of Givens and Householder can be used to produce a Hessenberg matrix. To calculate the eigenvalues of the Hessenberg matrix one can proceed in two ways. One can either perform a further reduction (preferably in double precision arithmetic) to produce a tridiagonal matrix, and then use the methods described earlier (possibly using complex arithmetic), or one can make use of recurrence relations analogous to formula (13) to evaluate $p_n(\lambda)$ in a standard root-finding technique. When the eigenvalues of a Hessenberg matrix are known the eigenvectors can be calculated by inverse iteration.

Two methods which have been developed in recent years for application to non-symmetric matrices are the LR algorithm of Rutishauser and the QR algorithm of Francis.

The LR method is based upon the triangular factorization algorithm mentioned above in connection with the solving of linear equations. Writing $A = LR$, where L is lower triangular and R is upper triangular, we deduce immediately that $L^{-1}AL = RL$. Thus the matrix RL formed by multiplying the triangular factors in reverse order is similar to A. The principle of the LR algorithm is to generate a sequence of matrices similar to $A = A_1$ by means of the iteration

$$\left.\begin{aligned} A_{r-1} &= L_{r-1}R_{r-1} \text{ (defining } L_{r-1} \text{ and } R_{r-1}\text{)} \\ A_r &= R_{r-1}L_{r-1} \end{aligned}\right\} \quad (r = 2,3, \dots) .$$

It can be shown that under certain conditions the matrices A_r tend to an upper triangular matrix whose diagonal elements are the eigenvalues of A.

The QR method uses a different factorization of A, namely,

$$A = QR ,$$

where R is upper triangular and Q is orthogonal. Either the Jacobi rotation matrices or the Householder transformation matrices may be used for Q. In either case it can be shown that for a non-singular A the factorization is unique if the diagonal elements of R are taken to be real and positive. We see that $Q^TAQ = RQ$, so that RQ is similar to A. Once again we form a sequence of matrices similar to $A = A_1$ by means of an iterative procedure:

$$\left.\begin{aligned} A_{r-1} &= Q_{r-1}R_{r-1} \text{ (defining } Q_{r-1} \text{ and } R_{r-1}\text{)} \\ A_r &= R_{r-1}Q_{r-1} \end{aligned}\right\} (r = 2,3, \dots) .$$

The A_r will tend (under more general conditions than for the LR algorithm) to an upper triangular matrix whose diagonal elements are the eigenvalues of A.

SUGGESTED READING

L. Fox, An Introduction to Numerical Linear Algebra (Oxford University Press, London, 1964) (General Survey).

G.E. Forsythe and C.B. Moler, Computer Solution of Linear Algebraic Systems (Prentice-Hall, Englewood Cliffs, 1967) (Analysis of Gaussian elimination and its variants, with computer programs).

J.H. Wilkinson, The Algebraic Eigenvalue Problem (Oxford University Press, London, 1965) (Treatise with detailed error analyses and examples illustrating difficult cases).

A.S. Householder, The Theory of Matrices in Numerical Analysis (Blaisdell, New York, 1964).

Table 1

FORTRAN arguments for TC matrix routines

OPERATION	ENTRY	DIMENSIONS A	B	C
$X \rightarrow B$ where $A^TX = B$	MXEQU (A,B,I,J)	I,I	I,J	
	MXEQU1(A,B,I,J)			

The system of equations $A^TX = B$ is solved without pivoting. (This is valid when A is positive definite.) The matrix A is replaced by a matrix the product of whose principal diagonal elements is equal to the determinant of A.

MXEQU1 has the same effect as MXEQU, but assumes that CALL MXEQU has already been executed with the same A:

OPERATION	ENTRY	A	B	C
$AB \rightarrow C$	MXMPY (B,A,C,K,J,I)	I,J	J,K	I,K
$A^TB \rightarrow C$	MXMPY1(B,A,C,K,J,I)	J,I	J,K	I,K
$AB^T \rightarrow C$	MXMPY2(B,A,C,K,J,I)	I,J	K,J	I,K
$A^TB^T \rightarrow C$	MXMPY3(B,A,C,K,J,I)	J,I	K,J	I,K

A and B may be the same. If J is zero, C is filled with zeros:

OPERATION	ENTRY	A	B	C
$AB + C \rightarrow C$	MXMAD (B,A,C,K,J,I)	I,J	J,K	I,K
$A^TB + C \rightarrow C$	MXMAD1(B,A,C,K,J,I)	J,I	J,K	I,K
$AB^T + C \rightarrow C$	MXMAD2(B,A,C,K,J,I)	I,J	K,J	I,K
$A^TB^T + C \rightarrow C$	MXMAD3(B,A,C,K,J,I)	J,I	K,J	I,K
$AB - C \rightarrow C$	MXMUB (B,A,C,K,J,I)	I,J	J,K	I,K
$A^TB - C \rightarrow C$	MXMUB1(B,A,C,K,J,I)	J,I	J,K	I,K
$AB^T - C \rightarrow C$	MXMUB2(B,A,C,K,J,I)	I,J	K,J	I,K
$A^TB^T - C \rightarrow C$	MXMUB3(B,A,C,K,J,I)	J,I	K,J	I,K

Any of A,B,C may be the same. If J is zero the products AB, etc., are assumed to be zero, and the effect is $\pm C \rightarrow C$:

OPERATION	ENTRY	DIMENSIONS A	DIMENSIONS B	DIMENSIONS C
$A + B \rightarrow C$	MXADD (A,B,C,I,J)	I,J	I,J	I,J
$A - B \rightarrow C$	MXSUB (A,B,C,I,J)	I,J	I,J	I,J
$A \rightarrow C$	MXTRA (A,0,C,I,J)	I,J	-	I,J
$\beta A \rightarrow C$ (β = scalar)	MXMTR (A,β,C,I,J)	I,J	-	I,J
$-A \rightarrow C$	MXNTR (A,0,C,I,J)	I,J	-	I,J
$A^T \rightarrow B$	MXTRP (A,B,I,J)	I,J	J,I	-
$E \rightarrow A$ (E = unit matrix)	MXUTY (A,I)	I,I	-	-

Any of A,B,C may be the same.

In the following operations the diagonal matrix D is assumed to be stored as a one-dimensional FORTRAN array containing only the principal diagonal. A and C may be the same:

$DA \rightarrow C$	MXDMR (A,D,C,J,I)	I,J	I,I	I,J
$DA + C \rightarrow C$	MXDMAR(A,D,C,J,I)	I,J	I,I	I,J
$AD \rightarrow C$	MXDML (D,A,C,J,I)	I,J	J,J	I,J
$AD + C \rightarrow C$	MXDMAL(D,A,C,J,I)	I,J	J,J	I,J
$A + \beta D \rightarrow C$ (β = scalar)	MXDMA (A,β,D,C,I)	I,I	I,I	I,I

Table 2

Elementary matrix operations in CDC matrix package

Note that $A^{(p)}$ denotes a symmetric matrix stored in packed form (upper triangle stored by rows).

In all cases the declared row dimension of A, B, C are M1, M2, M3, respectively.

OPERATION	ENTRY	DIMENSIONS | A |	| B |	| C |
$A \rightarrow B$	MATRIX(1,I,J,0,A,M1,B,M2,0,0)	I,J	I,J	-
$A^T \rightarrow B$	MATRIX(0,I,J,0,A,M1,B,M2,0,0)	I,J	J,I	-
$A + B \rightarrow C$	MATRIX(21,I,J,0,A,M1,B,M2,C,M3)	I,J	I,J	I,J
$A - B \rightarrow C$	MATRIX(22,I,J,0,A,M1,B,M2,C,M3)	I,J	I,J	I,J
$AB \rightarrow C$	MATRIX(20,I,J,K,A,M1,B,M2,C,M3)	I,J	J,K	I,K
$A^TB \rightarrow C$	MATRIX(23,I,J,K,A,M1,B,M2,C,M3)	I,J	I,K	J,K
$A^{(p)} \rightarrow B$	MATRIX(5,1,0,0,A,0,B,M2,0,0)	-	I,I	-
$A \rightarrow B^{(p)}$	MATRIX(4,I,0,0,A,M1,B,0,0,0)	I,I	-	-
$A^TA \rightarrow B^{(p)}$	MATRIX(2,I,J,0,A,M1,B,0,0,0)	I,J	-	-

Table 3

6600 Execution times for some simultaneous equation and matrix inversion routines

Routine	Language	Action	Method	Pivoting	Time in seconds n = 10	n = 100
LINEQ1	F	$X=A^{-1}B$	Gauss	Complete	0.008	4.9
MATRIX	A	$X=A^{-1}B$	Jordan	Complete	0.007	4.4
				Partial	0.004	1.7
				Sequential	0.003	1.4
MXEQU	A	$X=A^{-1}B$	Gauss	Sequential	0.004	2.5
MATIN1	F	A^{-1}	Jordan	Complete	0.016	11.9
MATRIX	A	A^{-1}	Jordan	Complete	0.008	5.6
				Partial	0.006	3.0
				Sequential	0.005	2.8

MONTE CARLO METHODS

by

R. Keyser

CONTENTS

1. HISTORICAL INTRODUCTION

Before scientists had available electronic computers, all probabilistic problems were expressed and solved in a deterministic manner. However, there are one or two instances where experiments were performed which contained the ideas of Monte Carlo methods -- perhaps the best known one being that of Captain Fox who threw a needle at a board ruled with straight lines, and inferred the value of π from observations of the number of intersections between needle and lines. It was with the advent of computers, and in particular with the work of von Neumann and Ulam on the atomic bomb, that Monte Carlo methods were used as a research tool. With advancing techniques, Monte Carlo methods were used to solve deterministic problems such as simultaneous equations, eigenvalue problems, etc. However, in a lot of these cases Monte Carlo methods are inefficient, and their application to unsuitable problems and also their use in a crude manner led to their discredit.

With improved variance-reducing techniques, the growing emphasis on problems that involve detailed practical complications, the simulation of probabilistic situations and a better recognition of which problems can efficiently be solved by Monte Carlo methods, such methods have established themselves as a useful tool. In their breaking of new ground they have allowed problems to be seen from a different angle; this has permitted new techniques, such as those using quasi-random numbers, to be developed.

2. CHARACTERISTICS OF THE MONTE CARLO METHOD

One constructs for each problem a random process with parameters equal to the required quantities of the problem, determining estimates of those parameters approximately by carrying out the random process and then computing statistically the required parameters and also setting some form of confidence interval on the results.

For example in evaluating

$$I = \int_0^1 f(x)\,dx$$

if ξ is a random number with a rectangular distribution between 0 and 1, then $f(\xi)$ is an unbiased estimator of I, i.e.

$$E\,[f(\xi)] = \frac{1}{N}\sum_{i}^{N} f(\xi_i) = \bar{f}$$

$$= I$$

and its variance can be shown to be σ^2/N where

$$\int_0^1 [f(x) - I]^2 = \sigma^2 \ .$$

Further we know by the Central Limit Theorem that $\bar{f}$ is normally distributed.

Thus by taking a series of random numbers $\xi_1, \xi_2 \ldots \xi_n$ and forming $\bar{f}$ we can say that

$$I = \bar{f} \pm \frac{2\sigma}{\sqrt{N}} \qquad \text{with 95\% confidence}$$

or

$$I = \bar{f} \pm \frac{2.6\sigma}{\sqrt{N}} \qquad \text{with 99\% confidence.}$$

Now we do not in general know σ but may estimate it from the formula

$$s^2 = \frac{1}{N-1} \sum_i^N \left(f_i - \bar{f}\right)^2 .$$

The most important fact above is that the error decreases as $N^{-1/2}$ -- it is unfortunate that to gain one more significant decimal digit one must calculate 100 times the number of points.

3. APPLICATION OF MONTE CARLO METHODS

The generation of random numbers ξ_i and the tests of randomness are left outside the scope of this lecture. They are generated by the functions

RANF on the CDC 3800

RNDM on the CDC 6600.

The routine on the CDC 3800 is statistically superior to RNDM which is unfortunate since the CDC 6600 is considerably better suited to Monte Carlo calculations than is the CDC 3800.

Monte Carlo methods have been applied to various classical problems including eigenvalue problems, simultaneous equations, partial differential equations, and integrals. However, if a classical method of solution for these problems exists, the classical method is invariably superior.

To obtain some idea as to when the Monte Carlo method may be a reasonable attack, let us note that in the above example the method is

i) independent of the dimension of the integration;
ii) independent of the nature of f other than it was specified to be finite;
iii) easily adapted to the interval of integration if it should be sectional or has other such difficulties.

Thus Monte Carlo methods may be considered for

i) multidimensional problems;
ii) "fussy" problems, e.g. awkward regions or discontinuous functions;
iii) problems intractable in terms of classical mathematics;

iv) simulation of probabilistic processes;

v) large problems, especially isolated solutions, for example, under certain conditions an isolated unknown in a system of 100 × 100 simultaneous equations.

If we wish to improve the error estimate $k\sigma/\sqrt{N}$ without increasing the labour involved, we must design methods which either reduce σ or improve on the power of N in the denominator. I will now state four principles in Monte Carlo work which should be followed:

I) use an exact value rather than an estimate whenever possible;

II) do as much analytically as possible;

III) reduce the variance of the method;

IV) choose your random numbers.

Let us now turn to these principles. (I) and (II) are actually particular cases of (III) but are worth stating separately, for in general the greatest gains in variance reduction arise from exploiting specific peculiarities of the problem.

4. VARIANCE-REDUCING TECHNIQUES

We shall now consider some variance-reducing techniques as applied to simple integration as a model. A simple integral is chosen since it lends itself best to the various ideas. In particular, the methods have been applied to the two integrals

$$I_1 = \int_0^1 \frac{e^x - 1}{e - 1}\, dx \ , \quad I_2 = \int_0^1 \sin \pi x \, dx \ ,$$

and the standard deviations of the various methods when applied to these two integrals are tabulated in the following table.

	I_1	I_2
1) Hit or miss	0.5	0.48
2) Crude	0.286	0.308
3) Central variate	0.037	0.063
4) Importance sampling	0.052	0.08
5) Stratified sampling	0.08	0.1
6) Antithetic variate	0.051	0.09
7) Regression analysis	0.0034	0.013

4.1 "Hit or miss" Monte Carlo

This was one of the earliest ways of applying Monte Carlo techniques. Let us illustrate it.

Suppose that

$$0 \leq f(x) \leq 1 \quad \text{when} \quad 0 \leq x \leq 1 .$$

Then I is the proportion of area of the square beneath the curve. Taking two random numbers ξ_{2i-1}, ξ_{2i} we call it a hit whenever $f(\xi_{2i-1}) \geq \xi_{2i}$.

Then if n^* is the number of hits out of n tries, $n^*/n \to I$. Since the distribution is binomial

$$\sigma^2_{n^*/n} = I(1 - I)/n .$$

This is a terrible method but is easily described pictorially. This method was one of the main causes for the bad reputation that Monte Carlo methods acquired. It is included here for completeness and as a warning.

For the two standard integrals we have

$$\sigma^2_{n^*/n} = \frac{0.418 \times 0.582}{n} = \frac{0.243}{n} \tag{1}$$

$$\sigma^2_{n^*/n} = \frac{\frac{2}{\pi}\left(1 - \frac{2}{\pi}\right)}{n} = \frac{0.231}{n} . \tag{2}$$

4.2 Crude Monte Carlo

This method has been described above in Section 2.

4.3 Control variate method

Suppose we know an answer to a simpler yet similar problem.

Then we break the problem up into two parts, thus:

$$\begin{aligned} I &= \int f(x)\, dx \\ &= \int \varphi(x)\, dx + \int_0^1 \{f(x) - \varphi(x)\}\, dx . \end{aligned}$$

Then we obtain an estimate of the second integral, which if $\varphi(x)$ is well chosen will have a smaller variance than the original, by crude Monte Carlo.

Taking $\varphi(x) = x$ for integral 1, and

$$\begin{aligned} \varphi(x) &= 2x \quad , \quad 0 < x \leq \tfrac{1}{2} \\ &= 2 - 2x, \quad \tfrac{1}{2} \leq x \leq 1 \end{aligned}$$

for integral 2, we obtain the figures given in the table.

Thus we may consider this as an example of principle (II) in the preceding section regarding $f(x)$ as a perturbation of $\varphi(x)$.

4.4 Stratified sampling

Break the range of integration into several pieces $\alpha_{j-1} < x \leq \alpha_j$ where $0 = \alpha_0 < \alpha_1, \ldots, \alpha_n = 1$ and apply crude Monte Carlo sampling to each of them. Then take as an estimator for I

$$t = \sum_{j=1}^{k} \sum_{i=1}^{n_j} (\alpha_j - \alpha_{j-1}) \frac{1}{n_j} f(\alpha_{j-1} + [\alpha_j - \alpha_{j-1}] \xi_{ij})$$

which will have variance

$$\sigma_t^2 = \sum_{j=1}^{k} \frac{\alpha_j - \alpha_{j-1}}{n_j} \int_{\alpha_{j-1}}^{\alpha_j} f(x)^2 \, dx - \sum_{j=1}^{k} \frac{1}{n_j} \left\{ \int_{\alpha_{j-1}}^{\alpha_j} f(x) \, dx \right\}^2 ,$$

where n_j is the number of sample points in the j^{th} piece. In practice,

$$s_t^2 = \sum_{j=1}^{k} \frac{(\alpha_j - \alpha_{j-1})^2}{n_j(n_{j-1})} \sum_{i=1}^{n_j} (f_{ij} - f_j)^2 ,$$

$$f_{ij} = f(\alpha_{j-1} + [\alpha_j - \alpha_{j-1}] \xi_{ij})$$

$$f_j = \frac{1}{n_j} \sum_{i=1}^{n_j} f_{ij} .$$

If the stratification is well chosen so that the variations of f within the pieces are less than the differences of the mean value of f in the various pieces, then s_t^2 will be less than the corresponding crude Monte Carlo process with Σn_j evaluations. In practice, one divides the original interval into k equal pieces (as has been done taking four intervals for the integrals of the table) or so that the variation of f is the same in each piece.

In the examples for the table, an equal number of evaluations have been taken in each interval but a further simple refinement is to take n_j proportional to the variance of f within the interval.

4.5 Importance sampling

We can write

$$I = \int_0^1 f(x) \, dx = \int_0^1 \frac{f(x)}{g(x)} \, dG(x)$$

where

$$G(x) = \int_0^x g(y)\,dy \ .$$

We restrict G to be a positive-valued function and (without further loss of generality) G(1) = 1. Then G(x) is a distribution function and if η is a random variable sampled from G then

$$\xi\left(\frac{f(\eta)}{g(\eta)}\right) = I$$

and with variance

$$\sigma^2_{f/g} = \int_0^1 \left(\frac{f}{g} - I\right)^2 dG(x) \ .$$

G(x) should be chosen so that f/g is as constant as possible. The transformation is especially valuable for unbounded integrals.

For the purpose of the table we have taken for the first integral

$$g(x) = 2 \ ,$$

and for the second integral

$$\begin{aligned} g(x) &= 4x \quad , & x < \tfrac{1}{2} \\ &= 4 - 4x, & x > \tfrac{1}{2} \ . \end{aligned}$$

5. ANTITHETIC VARIATES

Suppose t is an estimator for I.

In the control variate method we sought another estimator t′ with known expectation I′, and we then sampled t - t′ + I′ as the estimator of I. Since

$$\text{var } (t - t' + I') = \text{var } t + \text{var } t' - 2 \text{ cov } (t,t')$$

we will have gained if 2 cov (t,t′) > var t′.

An alternative approach is to seek an estimator t″ having the same unknown expectation as t and a strong negative correlation with t. Then $\alpha t + Bt''$, where $\alpha + \beta = 1$, will be an unbiased estimator of I, and

$$\text{var } \{\alpha t + \beta t''\} = \alpha^2 \text{ var } t + \beta^2 \text{ var } t'' + 2\, \alpha\beta \text{ cov } (t,t'') \ .$$

Thus we will gain if only cov (t,t″) < 0. Any estimators which mutually compensate each others' variations are termed antithetic variates.

For example we may consider a system of antithetic variates by stratification.

a) If the function is monotonically increasing we may take the estimator

$$S_\alpha = \alpha f(\alpha\xi_i) + (1 - \alpha)\, f(\alpha + [1 - \alpha]\, \xi_i) \quad .$$

b) If the function has a single maximum or minimum

$$T_\alpha = \alpha f(\alpha\xi_i) + (1 - \alpha)\, f(1 - [1 - \alpha]\, \xi_i)$$

may be taken with advantage.

It is difficult to locate the value of α exactly but $\alpha = 1/2$ is usually a fair guess. For T_α a more sophisticated estimate would be α such that

$$f(\alpha) = (1 - \alpha)\, f(1) + \alpha f(0) \; .$$

One can continue with this procedure and subdivide the intervals further by first using T_α and then S_β which requires four evaluations of f per estimate. Further variations are unlimited.

6. REGRESSION ANALYSIS

This is a method of general application that does, however, introduce a slight bias into the result.

In its generality we start off with several unknown estimands $I_1, \ldots, I_p$ and a set of estimators $t_1, \ldots, t_n$ where

$$E(\underline{t}) = \underline{X} \cdot \underline{I} \; ,$$

the x_{ij} being known constants.

Let $\underline{V}$ be the n x n variance-covariance matrix of the t_i's. Then it is known that the minimum variance unbiased linear estimator of $\underline{I}$ will be

$$\underline{t}^* = (\underline{X}' \, V^{-1} \, \underline{X})^{-1} \, \underline{X}' \, \underline{V}^{-1} \, \underline{t} \; ,$$

with

$$\text{var}\,(\underline{t}^*) = (\underline{X}' \, \underline{V}^{-1} \, \underline{X})^{-1} \; .$$

If V_0 is some other variance-covariance matrix we have

$$E(\underline{t}_0{}^*) = E\{(\underline{X}' \, \underline{V}_0{}^{-1} \, \underline{X})^{-1} \, \underline{X}' \, \underline{V}_0{}^{-1} \, \underline{t}\}$$

$$= E(\underline{X}' \, \underline{V}_0{}^{-1} \, X)^{-1} \, \underline{X}' \, \underline{V}_0{}^{-1} \, \underline{X} \, \underline{I} = \underline{I} \; .$$

If $\underline{V}_0$ is a reasonable approximation to $\underline{V}$, then $\underline{t}_0$* will be very nearly a minimum variance, unbiased estimator of I.

In practice, we obtain N independent sets of estimates $\underline{t}_1$, $\underline{t}_2$, ..., $\underline{t}_N$. An approximate variance-covariance matrix may be calculated as follows. Denote

$$\underline{t}_k = \{\underline{t}_{ik}\} \qquad (n \times 1)$$

and form

$$\bar{t}_i = \frac{1}{N} \sum_{k=1}^{N} t_{ik} \quad .$$

Then we take as our (i,j) element of V_0 the quantity

$$\frac{1}{N-1} \sum_{k=1}^{N} (t_{ik} - \bar{t}_i)(t_{jk} - \bar{t}_j) \quad .$$

With these values we calculate

$$(\underline{X}' \, \underline{V}_0^{-1} \, \underline{X})^{-1} \, \underline{X}' \, \underline{V}_0^{-1} \, \underline{\bar{t}}$$

as our estimator of I. It is nearly unbiased with covariance matrix approximately

$$(\underline{X}' \, \underline{V}_0^{-1} \, \underline{X})^{-1}/N \quad .$$

Suppose, for example, we estimate an integral I by

$$t_1 = f(\xi)$$

$$t_2 = f(1 - \xi)$$

and obtain N such estimates. Thus

$$\xi(t_1) = I \quad , \quad \xi(t_2) = I$$

i.e.

$$\underline{X}' = \{1, 1\} \quad .$$

Next we form $\bar{t}_1$ and $\bar{t}_2$ and $\underline{V}_0$ (2 × 2) as above. Then if

$$\underline{V}_0^{-1} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

$$t_0^* = \{(a + c) \, \bar{t}_1 + (b + d) \, t_2\}/(a + b + c + d) \quad .$$

7. QUASI-RANDOM NUMBERS

The above principles can be carried over for multidimensional integrals. In fact, Monte Carlo methods are more accurate than the trapezoidal rule in three or more dimensions and better than second order rules in five dimensions, since for such methods of order v in k dimensions the error is of the order

$$N^{(-2v+1)/k}$$

Thus until such time as better methods are devised, the Monte Carlo one is a practical approach.

However, so far we have not restricted f whatsoever. If f is well behaved (in the sense that it can be expanded in a Fourier series)

$$\sum_{n_1} \cdots \sum_{n_k} a(n_1, \ldots, n_k)\, e^{2\pi i \underline{n} \cdot \underline{x}}$$

and

$$\sum_{n_1} \cdots \sum_{n_k} |a| = B < \infty ,$$

then we can choose our random numbers and obtain estimates with errors $O(N^{-1})$ and $O(N^{-2})$.

In particular, to integrate

$$\int f(x_1, \ldots, x_k)\, dx_1, \ldots, dx_k$$

we choose $\alpha_1, \ldots, \alpha_k$ to be independent irrational numbers belonging to a real algebraic field $\delta(\geq k + 1)$, i.e. $\alpha_1, \ldots, \alpha_m$ are real roots of a polynomial of degree δ with integer coefficients, and no non-zero set of integers $n_1, \ldots, n_k$ exist such that $n_1\alpha_1 + \ldots n_k\alpha_k = 0$. Then

$$S = \sum_{n=1}^{N} f([n\alpha_1], [n\alpha_2] \ldots [n\alpha_k]) ,$$

where $[\cdot]$ denotes the fractional part of the enclosed number, is such that $S \to I$ with error $O(N^{-1})$.

Thus, for example, taking $\alpha_1 = \sqrt{2}$ and considering the integral

$$I = \int_0^1 \sin \pi x \, dx$$

it was found that the values of the maxima and minima of the error in I closely obeyed the law 1/N.

Although irrational numbers cannot be represented exactly in a computer, rounding errors can be neglected providing the number of digits in N^2 is less than the number of significant digits in α.

The generality of the Monte Carlo method in the sense that the restrictions on f are minimal gave rise to methods the efficiencies of which were initially questionable when applied to many analytic problems. With the marriage of Monte Carlo methods to other disciplines, efficient methods have been developed for problems which obey slightly more restrictive conditions.

* * *

BIBLIOGRAPHY

J.M. Hammersley and D.C. Handscomb, Monte Carlo methods (Methuen Monograph, London, 1964). Contains excellent bibliography.

Method of statistical testing - Monte Carlo method (Edited by Yu A. Shreider) (Elsevier Publishing Co., Amsterdam, 1964).

A. Hall, On an experimental determination of π, Messeng. Math. 2, 113-4 (1873).