File: 7094II.doc Version dated 11 July 2010

7094-II SYSTEM SUPPORT FOR NUMERICAL ANALYSIS

W. Kahan
Department of Computer Science
University of Toronto

Draft of first half: August 1966
With corrections to June, 1968

Retyped in 2010 for this file by S.K. Kahan from
IBM SHARE Secretarial Distribution SSD #159 Item C-4537 (1966)
plus an extract from
"Error in Numerical Computation",
Univ. of Mich. Eng'g Summer Conf'ce #6818,
Numerical Analysis (1968)

7094-II System Support For Numerical Analysis

W. Kahan, University of Toronto

Abstract

This is the first half of a progress report on the author's efforts to improve the performance of IBSYS in the following areas of FORTRAN programming:

1. Error-traces and diagnostic messages to locate and explain flaws found while executing FORTRAN programs.
2. Post-mortem facilities via the FORTRAN IV statement
   IF (KICKED (OFF)) ...
3. A consistent, sane and flexible treatment of over/underflow and related phenomena.
4. Digit manipulation (like rounding) via FORTRAN built-in functions.
5. The eradication of anomalies in the compiler (IBFTC) and the FORTRAN library (IBLIB).
6. The expansion of the FORTRAN library to include reliable and convenient subprograms for the solution of standard numerical problems like systems of linear equations,
   polynomial equations,
   eigenproblems,
   minimax approximation,
   fitting data by least squares,
   systems of ordinary differential equations,
   etc.

Items 1 to 5 are herein regarded as essential prerequisites to the accomplishment of item 6 in such a way that users of these subprograms need not supplement their own competency in mathematics, science, engineering or the humanities by a hyperfine proficiency at both numerical analysis and the debugging of systems programs. Each of the six

areas will be discussed in a correspondingly numbered section of this report, which begins by introducing the motivations for and the constraints upon the author's efforts. Sections 1 to 3 follow; section 4 to 6 will be issued separately later.

Sections 1 to 3 are intended to demonstrate in detail the validity of the author's rationale for treating unscheduled events during a computation. The reader who is unfamiliar with IBSYS and IBM's 7094 is asked to persevere; that rationale would work on his machine too if it were designed right.

Introduction

For as long as electronic computers have been in use (since 1949 at the University of Toronto), there has existed a steadfast policy to widen the range of intellectual disciplines that might benefit from the machine. That policy is partly responsible for a decline in the numerical sophistication of users, a decline which has yet to be compensated by an increased sophistication in the programs they can use. Despite intensive attempts to educate them in the arts of computation, too many new users attribute to the numerical library subprograms the infallibility of a mathematical proof. They shall be disillusioned. To what extent can their disillusionment be written off as part of their education? To what extent can their dissatisfaction be traced to shoddy computing systems? There is room for improvement in both the quality of education and the quality of computer performance. But you cannot teach an old dog new tricks, and you cannot teach a new dog very much. Therefore the bulk of the improvement must and can come in the performance of computer systems.

The performance of IBM's IBSYS on the 7094-II has left a lot of room for improvement. The improvements listed here were motivated almost entirely by the inadequacies uncovered during the author's researches into numerical methods. The object of the researches was to produce working programs about which might be proved something simple and useful to a numerically unsophisticated but otherwise intelligent and educated user. As a by-product of these researches, the following vague generalities have emerged:

-Computation costs most when its result is not known to be right nor wrong, because it costs so much to find out what is wrong and why. Costs can be cut by a small amount of self-doubt applied early.
-Whether or not the purpose of computing be “insight”, its most dependable benefit is hindsight. Programmers dislike forgoing this benefit through lack of foresight.
-Errors, anomalies and arbitrary restrictions hurt most when they are too rare to remember but not rare enough to ignore.

These generalities have influenced the many decisions on questions of detail which arose during the work on the system. A more decisive influence was exerted by three constraints.

First, it was deemed essential that programs be capable of conversion to whatever machine might replace the 7094-II, and so it was decided that all numerical subprograms be written in a language like FORTRAN or ALGOL, except where efficient coding was so obviously machine dependent that the assembly language MAP was used. I chose FORTRAN IV in preference to ALGOL. I would rather fight than switch. I am still fighting with the latest version (13) of the IBFTC compiler to incorporate all the modifications which I had introduced into the previous version, and further modifications to correct newly discovered deficiencies.

Second, since no one had anticipated a need to rewrite IBSYS or IBFTC in its entirety, no resources were allocated for such a task. Therefore, IBSYS and IBFTC have been modified as little as possible, instead of being replaced. The modifications have cost about three man-years of work all told, much of which has been dissipated in the transfer of the modifications from version 12 to version 13 of IBSYS.

Third, but most important, is our decision that the Toronto version of IBSYS remain compatible with the standard IBM IBSYS. Consequently, any FORTRAN IV program, even if it be in the form of a binary object-program deck, which has been designed for and runs correctly on a 7094 under standard IBM IBSYS with a hundred or so storage locations to spare, runs at least as well under our modified system. If the program be recompiled with no other modification, then the user may benefit from our improved diagnostics, especially where division by zero is concerned. Most of the users of our 7094-II are unaware of any departure from standard. But programs which run well on our system sometimes fail mysteriously at other 7094 installations.

In this report an attempt will be made to discriminate between IBM's standard IBSYS and our modified IBSYS by referring to theirs in the past tense whenever it differs from ours. Further details about IBM's IBSYS can be obtained from their manuals:

C28-6248 (IBSYS monitor)
C28-6389 (IBJOB; loader and library)
C28-6390 (IBFTC FORTRAN compiler)

Further details about our modified system can be found in

“The Programmers' Reference Manual” 3rd ed.

obtainable from

The Secretary, Institute of Computer Science,
University of Toronto,
Toronto 5, Ontario,
Canada.

and henceforth referred to as the PRM. Program listings are obtainable too if requested by name.

Acknowledgement

The author is deeply grateful for the patient assistance rendered by several IBM personnel, both in Toronto and elsewhere, who went out of their way, and sometimes out on a limb, to help with this work. Particular thanks go to J. Leppik, G. Howard and J. Bell for their help with the monitor, the compiler and the revised SAVE pseudo-op. Thanks go as well to colleagues in the Department and in the Institute of Computer Science for their encouragement over several years, and for their help with policy decisions about kick-off and diagnostic procedures.

Some of the work reported here was supported by the National Research Council of Canada.

1. Error-traces and Diagnostic Messages

It may seem peculiar that a Numerical Analyst be preoccupied with the System Programmer's traditional responsibility for error-traces, diagnostics and post-mortem information. But let us watch the Numerical Analyst at work. Much of his computer time is dissipated by the diagnostics and post-mortems which he receives while trying to discover why his algorithms do not work as well as he had hoped. From time to time he hands one of his subprograms on to some other user numerically less sophisticated than himself, and in so doing he tacitly shares with the Systems Programmers some responsibility for issuing diagnostics. His program may produce diagnostic messages for different reasons than merely to signal its own collapse. Diagnostics may be the only “correct” answers that the program can deliver in response to problems outside the intended domain of its applicability, especially when the program's domain cannot easily be defined other than by attempting to execute the program. For example, a hopelessly ill conditioned linear system

$$A\underline{x} = \underline{b}$$

is most easily identified when a sound linear-equation-solver fails to solve the system for $\underline{x}$ but exhibits instead a near linear dependence $\underline{d}$ in the left hand side A; i.e.,

$$\| A\,\underline{d} \| / (\| A \| \, \| \underline{d} \|) \approx 0 \quad .$$

The Numerical Analyst's subprogram ought to pass on this kind of diagnostic information in a form easily interpreted either by the user's calling program or by the user personally.

The latter form of diagnostic is usually a message printed amidst the user's output and is often the consequence of an error or oversight. The crucial question is

“Where was this error committed?”

but no computer program can answer this question. The best that can be done automatically is to answer the question

"Where did the program first encounter some anomalous consequence of the error?"

The answer takes the form of an Error-Trace. Under IBM's IBJOB this would be provided by library subprogram .FXEM., the FORTRAN execution Error Monitor. Let us examine an error-trace typical of those produced by IBM's .FXEM. . For example, suppose line 2 of the user's main program MAIN called a subprogram SUB1 in whose line 25 was a call to SUB2 in whose line 17 was a reference to SQRT(-4.0). When this reference was executed, the SQRT program would detect the inappropriately negative argument and call .FXEM. (say in line 31) to produce an error-trace and diagnostic message. IBM's error-trace would look like this:

ERR⊙R-TRACE CALLS IN REVERSE ⊙RDER

CALLING R⊙UTINE	IFN ⊙R LINE N⊙	ABS⊙LUTE L⊙CATI⊙N
SQRT	31	17621
SUB2	17	14513
SUB1	25	07762
MAIN	2	05413

The names in the first column are the deck-names assigned by the user to his subprograms (or else, in our modified system, assigned by default by the system). The line numbers or "Internal Formula Numbers" in the second column refer to numbers printed in the programs' source listings, and can be exploited by the FORTRAN IV programmer without recourse to storage maps. For this reason, the third column of absolute octal core locations is of secondary value to the FORTRAN programmer. It is a great convenience that he can ignore this column and dispense with storage maps most of the time.

The completeness of the error-trace shown above is one of its most valuable features. Complicated programs can contain several references to the SQRT subroutine, and it is vital that the path of control to the invalid reference by laid out explicitly. The complete error-trace is even more valuable when languages which permit recursive procedures are used. If a user were instead provided with only the reference to SQRT (or only to SQRT and SUB2) in the error-trace above, he might waste a lot of time checking through all of his calls to SUB2 in an attempt to uncover the faulty one.

IBM's .FXEM. would print out a two-line diagnostic message and provide a means to exercise options regarding kick-off or continued execution following the diagnostic error-trace. But .FXEM. suffered from two defects.

One, the easiest to remedy, was that .FXEM. could be called only from a MAP assembly language program. We fixed this by providing a program called UNCLE; any programmer can kick himself off (and produce an error-trace plus post-mortem debugging output) by executing

```
CALL UNCLE .
```

He can offer users of his program a limited range of kick-off-or-continue options by writing

```
CALL UNCLE (N)
```

with a suitably chosen integer expression N. He can supply one or two diagnostic messages too by writing

```
CALL UNCLE (N,Message)     or
CALL UNCLE (N,Message l, Message 2)   .
```

The messages can be inserted literally as Hollerith strings or they can be referenced as arrays of alphanumerical data. In the latter case, rudimentary binary-to-BCD conversion facilities are available to permit integer valued variables like indices or error-codes to be inserted into the diagnostic without first reserving core storage for the panoply of FORTRAN input/output subprograms. This last is an important considerations when program overlay is required during execution. (For more details about UNCLE, consult the PRM.)

.FXEM.'s second defect was that it could cope only with what I call “scheduled errors” ; these are errors each of which is discovered in a subprogram which, when it calls .FXEM. to produce an error-trace, can supply whatever linking information is needed by .FXEM. to start the error-trace. For example, SQRT(-4.0) is a scheduled error because SQRT is called in a standard way. But when unscheduled errors like over/underflow, division by zero, running overtime, ... , were detected they would “trap”, i.e. cause interrupts which transferred control to appropriate subprograms without carrying the standard linking information that made an error-trace possible. Consequently, the diagnostics for unscheduled errors answered the question “where?” with an absolute octal core location, but could not answer the question

“How did I get there?”

That IBSYS's standard linking sequence contained a partial answer to the last question was widely recognized. The first effort to extract a full answer was made by G. Wiederhold and G.D. Johnson at Berkeley (Univ. of California) in 1963. Their work has appeared in SHARE SSD 121 of May 21/64 and SDA's 3066-7. A similar scheme was devised by J. Leppik, G. Howard and the author at Toronto in 1964. Our scheme differs from theirs mainly in that ours is simpler to use, slightly less flexible, and fully compatible with the standard IBM system.

The first step in both schemes is to revise the standard SAVE pseudo-operation by which subprograms are expected to save and restore index registers, control linkages, etc. When IBM's SAVE was executed upon entry to a subprogram SUB, it used to save in a cell called SYSL⊙C the pointer to the statement

CALL SUB .

but no subsequent use was made of SYSL⊙C . We have added two instructions to SAVE whose effect is to store the same pointer, during the RETURN from SUB to the instructions following

CALL SUB ,

in such a way that the contents of SYSL⊙C show whether SUB has just been entered or has just returned. This modification has no effect upon the way IBM's .FXEM. behaves for scheduled errors.

Next, I rewrote .FXEM. so that it can be called from a trap-handling program. Such a CALL is distinguished from other standard CALLS by the absence of certain otherwise expected linking information, the lack of which forces .FXEM. into a new mode of action which examines SYSL⊙C to produce the first line of the error-trace.

The behaviour of the new .FXEM. is best illustrated by an example. Suppose that SUB2 in the example above contains, besides SQRT(-4.0), a division which, when executed, turns out to be a division of zero by zero. The result is the following diagnostic (in which the contents of the second line depend upon an option selected by the user):

```
                0.0/0.0 ERR⊙R AT     14506
   RESULTS IN 0.0      or      EXECUTI⊙N TERMINATED
ERR⊙R-TRACE WITH CALLS IN REVERSE ⊙RDER          C⊙DE 25
CALL IS IN      AT IFN ⊙R       ABS⊙LUTE
DECK NAMED      LINE N⊙.        L⊙CATI⊙N
  SUB 2           17+            14513
  SUB1            25             07762
  MAIN             2             05413
```

The important change shows up in the + sign after the line no. 17. This means that the announced anomaly was detected during or after (in time) the execution of line no. 17 of SUB2, but before any subsequent CALL was executed. Since SUB2 has a call to SQRT in line 17 at location 14513 (cf. the previous error-trace), and the 0.0/0.0 occurred five words ahead of this location in the program, it seems likely that the program was executing a loop, perhaps a D⊙-loop, which contains the offending division just a line or two in the listing ahead of the square root; and this loop was executed at least once before the divisor vanished.

The detective work in the last sentence is not typical; usually the error can be located by the most superficial inspection. But the need for any detective work at all is an unfortunate consequence of the way IBM's FORTRAN IV compiler works. Instead of identifying every line in the symbolic listing with a line number that .FXEM. could deduce at execution time (for example, by locating a dummy instruction

```
TIX     ID, 0, LKDR
```

at the beginning of the coding emitted by the compiler for line no. ID of the FORTRAN subprogram whose linkage information can be found at LKDR), the compiler assigns a useable line number only when a CALL is generated. Since an implicit CALL is generated for all references to FUNCTION subroutines, as well as for most

exponentiations of the form X**J and X**Y, for input/output, for complex multiplication and division, and for a computed G⊙ T⊙ (n_1 ,n_2,...,n_m), I , there are few programs whose listed line numbers are too sparse for a successful interpretation of the error-trace. And, at worst, the unscheduled error is located to within one subprogram.

The C⊙DE 25 at the head of the error-trace tells the programmer how to exercise his option to define 0.0/0.0 in one of two ways; either

0.0/0.0 = 0.0 and continue execution, or
0.0/0.0 = EXECUTI⊙N TERMINATED.

For example, the first option is the result of executing

CALL KIK⊙PT (25, 1)

while the second results from

CALL KIK⊙PT (25, 0) .

The reader is referred to the PRM for precise details about available options and how to exercise them conveniently. What follows is a condensation.

The PRM contains a table of error codes and messages (cf. Fig. 25 and the section “Subroutine Library Error Messages” in IBM's IBJOB manual, Form C28-6389-1) which describes for each code its error condition, the options available, and which option is assumed by the system in default of a request to the contrary. The default option is usually to provide a message and then continue execution in some reasonable way.

I believe that, taken together with the other diagnostic facilities in our system, our surprisingly simple set of options covers almost all circumstances satisfactorily. For serious errors we assign positive codes, like +25 for 0.0/0.0, to signify that the allowed options are

+1) Give a message and error-trace, and then continue reasonably,
or
+0) Give a message and error-trace, and then terminate execution.

(Some errors, like

G⊙ T⊙ (1, 2, 3), 4

are so serious that option +1 is denied.) For milder errors we assign negative codes, like -13 for SQRT (-4.0), which signify that the allowed options are

-1) Give a message and error=trace, and then continue reasonably,
or
-0) Give no message nor error-trace; just continue reasonably.

The meaning of “continue reasonably” is discussed later in this report. For now it suffices to give a few examples:

Error Condition and "Reasonable" Response	Code
SQRT(-X) = - SQRT(X)	-13
L⊙G(-A) = L⊙G(ABS(A))	-10 *
0.0**0 = 1.0	- 3
0**0 = 1	- 1
0.0**0.0 = 1.0	+ 6
0.0/0.0 = 0.0	+25

*Footnote: We allow programmers to write L⊙G(X) or AL⊙G (X) interchangeably as they please rather than penalize them for the venial sin of omitting the A .

Programmers, particularly writers of library subprograms, can easily provide other kinds of optional responses to error conditions detected by their own subprograms because the status of the option-indicator (a binary digit) associated with any error-code number can be sensed and stored as well as changed via KIK⊙PT. A complicated program may have several error-codes assigned to it, but this causes no problems because 280 codes are available. Programmers are free to use error-codes as flags or flip-flops in a way comparable to the use of sense-switches and sense-lights on the older slower machines.

A comment is required to explain that last .FXEM. option -0 which, in effect, allows .FXEM.'s activity to be suppressed entirely when the error is a mild one with a negative code. Some of these errors are better described as differences of opinion about the most apt definition of a function or an expression, as in the cases of 0**0 = 1 and 0.0**0 = 1.0 (cf. the Taylor series $\sum_0^\infty a_r x^r$ at x = 0.0). In these cases the warning messages serve only to remind the user that my definitions are not universally accepted in the computing world. If he is satisfied to do things my way, he can turn the message off. If he prefers another way, he can easily change the relevant program to his own specifications with the aid of the documentation which we supply.

Other errors with negative codes sometimes represent minor oversights; an example is

L⊙G(-X) = L⊙G(ABS(X)) , code – 10.

For reasons discussed later, our policy is to try not to terminate execution because of such an oversight. Rather, it seems better to continue and find out what else the programmer overlooked. We do not encourage programmers to exploit system side-effects to save the bother of a sign-test or some such simple instruction. We do not regard the -0 option as one which should be employed in production or library programs to correct oversights, except possibly temporarily, because this type of hidden coding is so difficult to remember when late-hatching bugs are being sought.

To implement the new .FXEM. and error-trace required several man-months of work, most of which was spent tracking down anomalies. For example, several input/output programs supplied as part of earlier versions of FORTRAN IV were found to use non-standard subprogram linkages, and these had to be repaired to allow even the old .FXEM. to produce meaningful error-traces before they were further modified to work with the new .FXEM. . Every library program had to be examined; here we reaped an unexpected reward when we discovered that the new .FXEM. makes possible a shorter and faster subprogram linkage to certain library programs like SQRT, C⊙S, L⊙G, EXP, complex multiply, complex divide, A**J, and others.

But one large job remains. The FORTRAN compiler must be modified to generate standard CALLS to Arithmetic Statement Functions which at the present, as compiled by IBM's FORTRAN IV v. 13, use non-standard CALLS in order to save about 7 microseconds per CALL. (One division costs 8.4 microseconds.) Consequently both IBM's .FXEM. and ours produce error-traces which skip, sometimes confusingly, over references to Arithmetic Statement Functions.

2. Post-mortem Facilities

We prefer to think of kick-off as an act of desperation on the part of a subprogram, and therefore try not to terminate execution unless it is overwhelmingly probable that continued execution will be an utter waste. There is little risk that errors like SQRT(-4.0) will be repeated millions of times to no good purpose, because the monitor imposes the user's own limit upon the total number of lines of printed output, thereby protecting him from a million lines of SQRT's diagnostic and error-trace. Furthermore, programmers who are especially sensitive to a waste of their computer time allotment can use statements like

```
IF (CL⊙CK (TSTART)    .GT.   TMAX)     CALL UNCLE
```

to kick themselves off when the elapsed time since

```
TSTART = CL⊙CK (0.0)
```

exceeds TMAX, at a cost of 70 microseconds per execution. (One square root costs 64 microseconds.)

But sometimes kick-off is the only reasonable response to an error. This response gives rise to a breed of programmer who has only one diagnostic and error-trace to show for his several seconds (or minutes) of computer time. It is uncharitable to advise him that he should have exercised enough foresight to provide intermediate output as insurance against such an event. Besides, he may reply

“I thought I had debugged that program.”

We doubt the wisdom of the widespread tendency to inundate every user who is kicked off with a complete dump of storage willy-nilly. This could drown him in octal data which he is unlikely to be able to read. It is a costly way to educate students.

The ideal solution would be to display conveniently just those variables which have figured in the events leading up to the debacle. Our solution is not ideal, but it is simple and flexible. It is an improved version of our PM⊙RT described in Comm. A.C.M. 7 (1964) p. 15. We allow the programmer to write into his FORTRAN IV program a statement of the form

IF (KICKED(⊙FF)) ‹ any executable statement ›
‹ the next executable statement ›

with the expectation that, because the value of the logical function KICKED is always .FALSE. , his program will merely execute ‹the next executable statement› . But if and when his program is kicked off, the monitor will give him the diagnostic and error-trace that he deserves and then, after over-writing ‹the next executable statement› with CALL EXIT, will execute ‹any executable statement› .

e.g. 1: IF(KICKED(⊙FF)) WRITE(...)

causes the desired information to be written out if and only after the program has been kicked off. The programmer can choose a F⊙RMAT to suit himself or, if more convenient, he can use the simple un-formatted output provided by the NAMELIST feature of FORTRAN IV; or he can CALL DUMP and be drowned.

e.g. 2: IF(KICKED(⊙FF)) CALL ... or
G⊙ T⊙ ...

causes the desired transfer of control to take place after kick-off, and thus permits a user to store valuable data on magnetic tapes and ask the operator to save them. Or he can call a complicated diagnostic program of his own, or he can try again to solve his problem by some method other than the one which failed. The monitor will allow, say 20 seconds and 300 printed lines of computer activity after the first kick-off. Of course, any second kick-off is final

despite further IF (KICKED(⊙FF))... requests. Because the user has recourse to KICKED, writers of library and systems programs are under less pressure when they have to decide whether an anomalous condition should terminate execution or just produce a warning.

Programmers are encouraged to use KICKED as often as they like in both FORTRAN and MAP assembly language programs; and they can leave these KICKED statements in production programs as insurance against the remote possibility that an undiscovered bug may terminate execution in a cloud of mystery. Each executed reference to KICKED consumes less than 14 microseconds (less than two division times) so KICKED can be used in fairly tight loops without seriously wasting time. The monitor will respond at kick-off only to the last executed reference to KICKED.

An important limitation upon KICKED was imposed by the absence of any block structure in FORTRAN comparable to that in ALGOL, and by the way that indexing is optimized in FORTRAN. This limitation exists because, whenever kick-off occurs in some subprogram remote from the one containing the KICKED statement and then control is passed to ‹any executable statement› after the IF(KICKED(OFF)), no attempt is made to restore index registers to the state they were in when KICKED was called nor to re-set tapes to their former positions. More important, there is no way to reduce the effect of those instructions which may have been placed in “optimum” positions ahead of the call to KICKED in order to initialize index registers and addresses as efficiently as possible from the point of view of the normal sequence of control. For example, if kick-off occurs during the computation of FCN in the sequence

```
      D⊙  3      J = 1, 10
        A(1,J) = J – 1
        D⊙  3      I = 1, J
          IF (KICKED(⊙FF)) WRITE(...) I, J, B(I), B(J), (A(K,J), K=1,J)
  3       A(I+1, J) = FCN(B(I), B(J), A(I+1, J)) + A (I, J)
```

there is no way at kick-off time to move the numbers I and J from storage into the appropriate cells and index registers for the references to B(I), B(J), A(K, J) and "K = 1, J" following the call to KICKED.

A second limitation shows up when program overlay takes place; there is no simple way to detect whether ‹any executable statement› in the IF (KICKED(⊙FF)) statement has been partially overlaid, or whether it refers to data which has been overlaid. Consequently we inserted an instruction in .L⊙VRY, the overlay handling subprogram, which causes the monitor to forget the last reference to KICKED whenever overlay occurs. We take no pride in this expedient.

Any programmer who is aware of these two limitations can easily code around them. Simple suggestions are contained in the PRM. Indeed, the limitations are so easy to circumvent that programmers sometimes forget to do so, and for this reason we have included a warning message like the one in the following example:

```
0.0/0.0 ERR⊙R AT 14506
EXECUTI⊙N TERMINATED.
ERR⊙R-TRACE WITH CALLS IN REVERSE ⊙RDER     C⊙DE 25
CALL IS IN       AT IFN ⊙R    ABS⊙LUTE
DECK NAMED       LINE N⊙      L⊙CATION
   SUB2            17+        14513
   SUB1            25         07762
   MAIN             2         05413
 EXECUTING IFN/LINE N⊙. 2 OF 'SUB1' AFTER PR⊙GRAM WAS
 KICKED ⊙FF.  FR⊙M N⊙W ⊙N IN 'SUB1, THE VALUE OF A SUB-
 SCRIPTED VARIABLE WITH VARIABLE SUBSCRIPT, OR THE EXE-
 CUTI⊙N OF A C⊙MPUTED 'G⊙ T⊙' OR 'D⊙' STATEMENT WITH
 VARIABLE PARAMETER, MAY BE INC⊙RRECT UNLESS THE RELEVANT
 INDEX IS RESET.  SEE THE PR⊙GRAMMERS' REFERENCE MANUAL.
```

This message is more formidable than necessary. It would be unnecessary altogether if the IF(KICKED(⊙FF)) statement were implemented in a language, like ALGOL, with a block structure. Then kick-off within a block would cause control to be transferred to the last KICKED reference, if any, executed in the same block but not in a deeper sub-block.

One other complication would arise were the IF(KICKED(⊙FF)) statement to be implemented within a compiler which contained a M⊙NIT⊙R statement. Such a statement is exemplified by

M⊙NIT⊙R X, Y(*), Z(*,3), PR⊙G, n

which would cause output of the following kind to be generated:

Whenever the variable X is changed, write out its new value;

X = 14.271434 .

Whenever the variable Y is changed, indicate which element too;

Y (2) = .74131042 E –18 .

Whenever the third column of array Z is changed, say so;

Z(13,3) = 0.0 .

Whenever the subprogram PR⊙G is called, write out its arguments;

CALL PR⊙G (13, 27.421493, Y) WITH

Y(1) = 1.4012362

Y(2) = .74131042 E -18

Y(3) = 0.0 .

If PR⊙G is a function, write out its value too;

PR⊙G (13, 27.421493, Y) = 1.7014 E38 WITH

Y(1) = etc.

Whenever statement n is executed, say so. If this is a logical IF statement, tell what happened.

The M⊙NIT⊙R facility as described above has been implemented at least partially in several compilers*; unfortunately, ours is not one of them. The problem is to deal with the statement

IF (KICKED(⊙FF)) M⊙NIT⊙R ......,

for which the nicest solution would be a retroactive display of, say, the last 300 lines of output which would have been produced if that M⊙NIT⊙R statement had not been bypassed. Some compilers already have a feature of this kind; the author envies their users.

Now is a good time to compare the error-options needed by the programmer with those available to him. He may want to assign to a specified anomaly, like 0.0**0 , one of the following four consequences:

-0) Re-interpret the request in a way judged to be appropriate for the majority of users (say 0.0**0 = 1.0) and continue with no message nor error-trace.

1) Re-interpret the request as above, and put out a message and error-trace to tell the programmer what happened and where, and then continue execution.

+0) Put out a message and error-trace to explain where and why execution was terminated, and then grant any post-mortem request that may have been made via

IF (KICKED(⊙FF)... .

2) Transfer control to a location designated in advance by the programmer where he may cope with the anomaly as he pleases, provided the necessary information is easily accessible to him.

---

* R. Bayer, D.Gries, M. Paul, H.R. Wiehle [1967] "The ALCOR Illinois 7090/7094 Post Mortem Dump" *Comm. ACM* 10 #12 pp. 804-8

Our system offers at least two of the first three options for most error conditions. The last option is dangerous in FORTRAN for the reasons cited while discussing the limitations of KICKED, unless it is handled carefully. The following discussion explains how some of our library programs offer option 2).

Consider for example our least squares library subroutine LSTSQ which, given a rectangular $M \times N$ matrix X and a column vector $\underline{y}$, attempts to find that coefficient vector $\underline{c}$ which minimizes the sum of squares

$$S = (\underline{y} - X\underline{c})^T (\underline{y} - X\underline{c}) = \Sigma_i (y_i - \Sigma_j x_{ij} c_j)^2 \ .$$

A solution $\underline{c}$ always exists and satisfies the normal equations

$$X^T X \, \underline{c} = X^T \underline{y} \qquad .$$

LSTSQ tries to solve these equations (in double precision, because that is the fastest adequate method on a 7094) for $\underline{c}$ and the corresponding minimum value of S and, if requested, the inverse matrix

$$V = (X^T X)^{-1} \qquad .$$

But if the columns of X are nearly linearly dependent, in the sense that there exists a perturbation ΔX of the order of a few units in the last place of X such that the columns of $(X+\Delta X)$ are linearly dependent, then the solution $\underline{c}$ is not well defined and LSTSQ produces one of two things instead of $\underline{c}$:

0) If the user wrote

```
CALL LSTSQ (X, M, N, Y, C, S)       or
CALL LSTSQ (X, M, N, Y, C, S, V)
```

then he has made no provision for the possibility that X be nearly singular, so he receives a suitable diagnostic and error-trace and is kicked off.

1) If the user wrote

CALL LSTSQ (X, M, N, Y, C, S, $n) or

CALL LSTSQ (X, M, N, Y, C, S, V, $n)

where n is an integer standing for a statement number, LSTSQ returns control to statement number n in the user's calling program, and diagnostic information is made available in V (or elsewhere if V was not requested) which permits the calling program to identify the linear dependence relatively easily and change X appropriately. (Usually the calling program just decreases N.) LSTSQ does not put out any messages in this case.

The foregoing description is somewhat simplified; details can be found in the PRM. The interesting feature is not so much the use of a FORTRAN IV error return $n as the fact that this error return is optional. The option is available because one of the first statements executed within LSTSQ is

CALL ARGCNT (I,J)

which counts the arguments supplied in the CALL to LSTSQ. I is the number of arguments exclusive of error returns, and J is the number of error returns. The error options described above are numbered 0 and 1 according to the value of J. Similarly, LSTSQ determines whether the user wants $V = (X^TX)^{-1}$ or not according as I = 7 or 6 respectively. Any other values of I or J indicate an error, like a period between the integers M and N instead of a comma, which is serious enough to terminate execution with an appropriate diagnostic.

The use of variable-length argument lists lends a certain elegant simplicity to several of our library programs, and we hope that this feature will be incorporated in the programming languages of the future.

The simplicity with which the error return scheme can be implemented makes it efficient and satisfactory for a wide range of applications, but there are two important areas where the scheme is unsatisfactory. One consists of those difficulties caused by a small lack of foresight and recognized immediately with the slight assistance to hindsight provided by a diagnostic. Many of the error conditions mentioned above, like L⊙G(X) when L⊙G(ABS(X)) was intended, fall into this category. So do many input/output problems. It suffices here to say that a lot more could be said for the desirability and convenience of subprograms like KIK⊙PT which allow the programmer to revise temporarily the execution of his program at each of several spots without having to insert a small explicit change at each spot.

The second area where error returns have proved unsatisfactory covers Over/Underflow, a ubiquitous phenomenon to which the next section of this report is devoted.

3. Over-Underflow

Overflow and Underflow are what take place in the arithmetic registers of a computer whenever an attempt is made to calculate numbers outside the normal range. On the 7094, overflow occurs whenever the magnitude of the result of a floating point arithmetic operation equals or exceeds

$$2^{127} \approx 1.70141183 \times 10^{38} ;$$

underflow occurs whenever the magnitude is not exactly zero and is smaller than

$$2^{-129} \approx .146936794 \times 10^{-38} .$$

Special provision must be made to cope with over/underflow in a way which does not produce misleading results.

It is sometimes argued that overflow is an error for which the penalty should be

EXECUTI⊙N TERMINATED

but this penalty would place an intolerable burden upon even the most expert numerical analyst. He is often unable to predict in advance what the range of numbers will be in complicated calculations, especially where exponentials, polynomials, and rational functions of high degree, or spaces of high dimensionality are concerned. For example, if $P(x,y)$ is a polynomial in x of degree 10 whose coefficients are wild functions of y, then the desired solution $x = X(y)$ of the equation $P(x,y) = 0$ may be well-defined and reasonable even though it is inaccessible unless the polynomial-zero-finding subprogram is allowed to pursue a flexible scaling strategy in response to over/underflows, if any, which occur during the computation of $P(x,y)$. Overflows should not force kick-off; if worse comes to worst, a program can kick itself off by executing, say,

```
IF(⊙VFL⊙W)  CALL UNCLE(0,22H INESCAPABLE ⊙VERFL⊙W.)    .
```

An opposite attitude of laissez-faire is reflected in the designs of those machines whose hardware automatically replace an overflowed magnitude by a special digit pattern representing ∞ and then plunge on. Such a scheme might well include, say, Θ to replace an underflowed magnitude and % to indicate an indeterminate value. These symbols might obey rules like the following:

i) Whenever an arithmetic operation generates $\pm\infty$, Θ or %, a corresponding flag is raised to indicate to the program that overflow, underflow or lost significance respectively has occurred. If requested by the programmer in advance, a message can be printed out for his information.

ii) Any arithmetic operation with % as an operand generates % as a result. % is also generated by the following expressions $\infty-\infty$, ∞/∞, $0/0$, $0/\Theta$, $\Theta/0$, $\infty*0$, $\infty*\Theta$ and x/Θ.

iii) If $x \geq$ (1 unit in the last place of the overflow threshold) then $\infty-x = \%$; otherwise $\infty \pm x = \infty$.

If (1 unit in the last place of x) $\leq$ (the underflow threshold) then $x \pm \Theta = \%$; otherwise $x \pm \Theta = x \pm 0 = x$.

If $x \geq 1$ then $x * \infty = \infty * \text{sign}(x)$; otherwise $x * \infty = \%$.

Similar rules hold for x/∞, ∞/x, $x*\Theta$ and Θ/x.

$x/0 = \infty*\text{sign}(x)$ unless $x = 0$ or Θ.

iv) The number 0 can be generated only by direct assignment or as the result of $x-x$ with $x \neq \Theta$ nor ∞. The symbol Θ, which stands for the set of all numbers smaller in magnitude than the underflow threshold, can be generated only by direct assignment or by an underflow as indicated above.

During comparisons the symbol Θ simultaneously falsifies

$$\Theta > 0 \quad , \quad \Theta = 0 \quad , \quad \Theta < 0 \quad ;$$

and $x > \Theta$ if and only if $x > 0$ too.

Rules like the foregoing are formidable, and have not been implemented in any hardware known to the author (who would not expect to find them in any machine except possibly one with interval-arithmetic built into the hardware). But no other less elaborate rules are known to be foolproof.* For example, the CDC 6600's hardware follows similar rules whose most obvious difference is the lack of any distinction whatever between underflow to Θ and the number 0 . A comparable deficiency is to be found at those IBM installations where, to escape a plethora of insignificant underflow messages, all underflow messages are suppressed by many users most of the time. The following segment of FORTRAN coding shows what can happen when this is done. Here A, B, C, D and X are all positive normalized floating point numbers (not special symbols nor zero).

```
              Y = (A*X+B)/(C*X+D)
              Z = (A+B/X)/(C+D/X)
              W = Y/Z
              WRITE (...) W
              ...........
         Output:  W = 1.98
```

In the absence of any indications of over/underflow, how is this phenomenon to be explained? The only thing unnatural about this example is the WRITE statement; W is more likely to have remained "out of sight, out of mind" .

The replacement of underflowed numbers by zero with no indication to program nor programmer is a clearly unsatisfactory practice. And even when an indication of over/underflow is given,

---

* Experience since this was first written has found Θ to be useless.

there is ample reason to protest against the destruction by hardware (as on the IBM 360 and CDC 6600) rather than software of information which could otherwise be of significance to the programmer; this is discussed in more detail below in connection with the Unnormalized Mode and the Counting Mode of treating over/underflow. But, to be fair, it must be acknowledged that most programmers would be satisfied most of the time by the provision of representations for $+\infty$, $-\infty$, $\ominus$ and $\%$ obeying rules like i) to iv) above.*

What more might a numerical analyst demand? From time to time time he will want to generate and use numbers which lie beyond the over/underflow thresholds. And certainly no programmer wants to be forced to check for over/underflow after (much less before) the execution of each arithmetic instruction in his program, and to decide each time upon an appropriate course of action. He will prefer to choose one of the several modes of execution provided for him by the system, with the understanding that while the program is being executed in his chosen mode each over/underflow will be treated according to the rules tabulated for that mode. Rules i) to iv) above could define one such mode. The programmer should be allowed to change modes between one line of his program and the next. Ideally, he should be allowed, if he wants, to define his own mode by specifying in detail just what rules are to be obeyed for each type of arithmetic operation. Finally, although the programmer who is ignorant of the problems of over/underflow must be warned when they occur, care must be taken not to drown him in a cascade of over/underflow messages, especially when they are obviously irrelevant. (An example of an obviously irrelevant underflow is remainder underflow after a floating point division in a FORTRAN program, which always discards the remainder.)

---

* Except for $\ominus$.

An attempt has been made to serve as many of these needs as can be served in a FORTRAN context by means of a substantial extension of the service supplied by IBM via their subprogram .FPTRP in IBJOB . This program exploits the fact that whenever a floating point over/underflow occurs the 7094 “traps”; it interrupts itself and transfers control to a designated core location after setting up an indicator word (cell 0) to describe what caused the trap and where. This floating point trap, FPT, takes precedence over all others in the machine, and when it occurs the registers in the machine contain the over/underflowed result unaltered, so that no significant information is lost. A hardware option can be purchased (RPQ 880291) which includes improper divisions like 1/0 in the scope of the FPT .

I rewrote .FPTRP in a way which, while maintaining compatibility, increased its speed and augmented its capabilities so that programs can easily choose and change to any one of five modes of execution. The Standard Modes treat over/underflow very much as IBM did, the main difference being that now underflow sets up an indicator the same way as does overflow. The Unnormalized Modes exploit unnormalized arithmetic to permit underflow to occur “gently” without setting up distracting indicators or messages. The Silent Modes set indicators to indicate over/underflow to the program but put out almost no messages for the programmer; cascades of over/underflows in the Silent Modes do not slow programs down appreciably. The Printing Modes set indicators for the program and also report each indicated over/underflow, as it occurs, in a printed message for the programmer, thus helping him to debug his program. The Counting Mode allows

certain kinds of computations to be carried out with no risk of over/underflow because the allowed range of magnitudes is extended to include numbers like

$$2^{\pm 2^{42}} .$$

These five modes are discussed below in appropriately titled subsections of this report. The last two subsections discuss improper divisions and simulated over/underflows.

The Standard Silent Mode

This is the mode in which the system operates by default in the absence of requests for some other mode. Whenever a floating point arithmetic operation overflows, its result is replaced by the largest possible magnitude (1.7014×10^{38}) with the same sign , and this event is recorded by setting ⊙VFL⊙W = .TRUE. . Whenever a result underflows it is replaced by zero with the same sign, and this event is recorded by setting UNFL⊙W = .TRUE. . The indicators ⊙VFL⊙W and UNFL⊙W are logical variables which can easily be sensed, stored and/or reset to .FALSE. in several ways described in the PRM. In particular, the declarations

```
L⊙GICAL ⊙VFL⊙W
C⊙MM⊙N/⊙VFL⊙W/⊙VFL⊙W
```

permit statements like

```
IF (⊙VFL⊙W)....          and
⊙VFL⊙W = .FALSE.
```

to be executed without wasting time on subprogram linkages in short loops.

This mode is called Silent because each over/underflow sets its indicator without disturbing the programmer's output with any diagnostic message. However, just after his program's execution is terminated (either normally or by kick-off) a message is produced to draw the programmer's attention to any over/underflows that escaped the attention of his program; more about this later. In the Standard Silent Mode, each over/underflow costs 15 to 30 microseconds; i.e. two to four division times.

The Standard Printing Mode

This mode differs from the previous mode only in that each over/underflow, as it occurs, inserts a message into the programmer's output to answer the following questions:

What happened, overflow or underflow?

Which machine registers are involved; AC, MQ or both?

What arithmetic operation was attempted: + , - , * , / , double-precision, ...,? (An octal operation-code is given here.)

What change was made in the affected register(s)?

Where is the instruction whose execution caused this over-underflow? (An octal core address is given here.)

Where in the source-program did all this happen? (An error-trace is given here by our version of .FXEM. .)

We also considered writing out the operands whose sum, product or quotient had over/underflowed, but the cost of doing so seemed more than the information was worth. This point deserves reconsideration. Anyway, the error-trace usually points to within a few lines of the site of the over-underflow in a FORTRAN program.

The over/underflow handling subprogram .FPTRP can be switched in 40 microseconds from a Silent Mode to the corresponding Printing Mode via the statement

CALL NFPTST(M)

with a positive integer expression M . When this statement is executed, an internal counter N is set to M and .FPTRP is caused to operate in a Printing Mode until M over-underflow

messages have been put out. N is decreased by 1 each time a message is put out, and when N becomes 0 an extra message

N⊙W ⊙VER/UNDERFL⊙W MESSAGES ARE IN ABEYANCE

is produced and the Mode is switched back to Silent.

CALL NFPTST(0)

switches the Mode back to Silent without any extra message.

In accordance with current good practice, the FORTRAN programmer is allowed easily to sense, save, set and/or reset the message-counter N as well as the indicators ⊙VFL⊙W and UNFL⊙W . Details may be found in the PRM. But programmers are advised not to set the latter two logical variables to .TRUE. directly in a FORTRAN program; instead they are advised to force an over/underflow like

```
DUMMY = (1.7E38)**2    .
```

This is done because, whenever over/underflow occurs, .FPTRP stores the current contents of SYSL⊙C into the appropriate indicator to make it .TRUE. . Later, when the program's execution is finished, the monitor looks at each indicator to see whether it is .TRUE. , and if so then that indicator is interpreted as a pointer in roughly the same fashion as .FXEM. interprets SYSL⊙C when providing the first line of the error-trace immediately after an over/underflow in the Printing Mode. Consequently, the programmer's output finishes, whenever appropriate and possible, with a message like

```
LAST UNREQUITED ⊙VERFL⊙W WAS IN ⊙R AFTER
LINE 17 ⊙F DECK SUB2  .

LAST UNREQUITED UNDERFL⊙W WAS IN A SUBPR⊙GRAM CALLED IN
LINE 24 ⊙F DECK SUB1  .
```

Often the programmer can deduce from the information given here that the over/underflow did no harm; then, since the messages have

not tainted his formatted output, he is free to cut them off and publish the rest.

If program overlay has intervened between the last unnoticed over/underflow and program termination, or if the indicators ⊙VFL⊙W and UNFL⊙W were set to .TRUE. in a naïve way, then the post-execution message may describe the desired deck-name and line number as UNKN⊙WN .

It is especially important to understand that the word "UNREQUITED" signifies that the *program* has not reset the indicators to .FALSE. , presumably because it has not responded to the over/underflows. The *system* may already have printed several messages for the programmer, notifying him each time his program ignored an over/underflow while the system was in the Printing Mode.

I see now that we could have supplied, at little extra cost, post-execution warnings more like this:

```
3943 ⊙VERFL⊙WS WENT UNREQUITED BY THE PR⊙GRAM BETWEEN
LINE 17 ⊙F DECK SUB2
AND A SUBPR⊙GRAM CALLED IN LINE 64 ⊙F DECK SUB1   .
```

Such a message can be more useful in deciding whether or not to ignore the over-underflows. Also, the counts of overflows and underflows could be used by any programmer who, for reasons unclear to me, wished to terminate his program's execution after a specified number of overflows had occurred. Another improvement would be to allow a negative value for M in

```
CALL NFPTST(M)
```

to signify that -M overflow messages are to be allowed while all underflow messages are to be suppressed. Most of these improvements have been incorporated into the adaptation of our scheme for the Burroughs B5500 written by Mr. Michael D. Green at Stanford University in 1966, and I expect to put them into our system soon.

The Treatment of Underflow

Some programmers have good reasons to want to be informed about underflow. They may want to avoid consequent loss of precision or subsequent division by zero. But most programmers whom I asked said they preferred that underflowed numbers be replaced by zero without their attention being distracted by the event. This attitude was justified at a time when most over/underflow messages reported "MQ UNDERFL⊙W" during an addition, subtraction, multiplication or double precision division. This message signified that the double-length result of those operations in the AC-MQ register was small enough to cause the characteristic of the less significant word in the MQ to underflow even though the more significant word was correct. Since the less significant word is entirely ignored in single-precision FORTRAN expressions, and since the double-precision hardware of the 7094 ignores the characteristic of the less significant word in double-precision expressions, I decided that .FPTRP. should simply ignore MQ underflow after those operations where it was obviously irrelevant.* This decision's first consequence was a welcome reduction in the number of messages and complaints, especially where iterative calculations with residuals tending to zero were concerned. The second consequence was that certain old 7090 programs which had performed double-precision arithmetic by simulating the 7094's double-precision hardware, ran into spurious overflow troubles and required revision so that they would use instead of simulate our machine's hardware. Fortunately, any user who insists upon running a 7090 program unchanged upon our 7094 can do so in safety by merely changing two well-marked instructions in .FPTRP . The second instruction

---

* The 27 significant bits in the MQ are not ignored nor cleared when the characteristic of the MQ underflows, so no accuracy is lost.

is needed to force appropriate actions when remainders underflow after division; otherwise they would be ignored too.

It is not good enough that the system ignores obviously irrelevant underflows. Many irrelevant underflows are not obviously irrelevant. Consider, for example, a segment of a typical matrix handling program which computes

$$r = b - \sum_i a_i x_i \quad .$$

The usual rule, which replaces each underflowed sum or product by zero, is satisfactory except when b and all the products $a_i x_i$ are so close to the underflow threshold that the usual rule produces a significantly wrong value for r. If all underflows are reported, how can the rare significant reports be distinguished from the common ignorable ones? If no underflows are reported, how can the rare incorrect values of r be distinguished from the common correct ones? The easiest way I know to cope with these questions is to use our Unnormalized Modes.

The Unnormalized Silent Mode and the Unnormalized Printing Mode

These two modes differ from one another in just one respect; the Printing Mode reports overflows in the way described under the Standard Printing Mode above. The two Unnormalized Modes differ from their corresponding Standard Modes only in the way they treat underflow. A number, which in a Standard Mode would have underflowed to zero and set UNFL⊙W = .TRUE. , is in an Unnormalized Mode replaced by its closest unnormalized approximation and UNFL⊙W is unchanged. For example, consider a decimal machine whose underflow threshold is $.10000000 \times 10^{-38}$. In a Standard Mode, $.15743219 \times 10^{-40}$ would underflow to zero, but in an Unnormalized Mode it is replaced by $.00157432 \times 10^{-38}$. A number must now drop below $.00000001 \times 10^{-38}$ before it is silently replaced by zero.

In the Unnormalized Modes the range of non zero floating point numbers representable in the 7094 is extended downward from 2^{-129} to 2^{-155} in single-precision and 2^{-182} in double-precision. This allows underflow to take place more gently, and improves the accuracy of certain results. But these benefits are secondary; the primary justification for the Unnormalized Modes is that they ease the task of deciding, in certain cases, whether a result is right or wrong.

For example, consider the following FORTRAN program to compute

$$r = b - \sum_1^N a_i x_i \quad .$$

(In accordance with good computing practice, and because it costs almost nothing extra to do so on our 7094-11, the products of the single-precision numbers a_i and x_i are accumulated to double precision before r is rounded (not truncated) to single-precision.)

```
      D⊙UBLE PRECISI⊙N  D
      DIMENSI⊙N  A(...), X(...)
      D = -B
 C    ENTER THE UNN⊙RMALIZED M⊙DE.    (14 MICR⊙SEC.)
               CALL  FPTUN
      D⊙ 1  I=1,N
   1   D = A(I)*X(I) + D
 C    REST⊙RE THE STANDARD M⊙DE,      (13 MICR⊙SEC.)
               CALL  FPTST
      R = 0.0 – RND(D)
```

The last statement rounds D to single precision, changes sign, and adds zero before storing the result in R . If the rounded value of D is a nonzero unnormalized number, then the normalization that always follows addition will cause an underflow which, in the Standard Mode, will set R = 0.0 and UNFL⊙W = .TRUE. . But if RND(D) is a normalized number then adding zero will not change anything. Consequently, R is correct as it stands, despite the possible underflows of intermediate results, with the following exceptions:

- If ⊙VFLOW or UNFL⊙W is .TRUE. , R is wrong.
- If severe cancellation has taken place in statement 1, R may be badly contaminated by double-precision truncation errors. This possibility is independent of over/underflow, and is irrelevant if B, A and X are each uncertain by a unit in their respective last places.
- If R = 0.0 then it may be further contaminated by an error of 2^{-156} , although this is irrelevant if B is non zero and uncertain by a unit in its last place. But

if B = 0.0 then all the products A(I)*X(I) might have underflowed to zero silently.

There are very few applications where any but the first exception is relevant, and that one is caught by the system. The absence of over/underflow tests in the inner loop permits calculations in the normal range to proceed with no noticeable loss of speed.

The Unnormalized Modes may be used in single precision, double precision and complex arithmetic at the cost of 42 micro-seconds per underflow. These modes would be much more useful on a 7094 but for a quirk in the hardware which forces the “normalized” product of two nonzero unnormalized numbers to be zero on certain occasions. The Unnormalized Modes are best suited to those machines, like the Burroughs B5500, which handle unnormalized operands without serious anomalies. But, because of the peculiar behaviour of our machine, the Unnormalized Modes are so beset by restrictions (for which see the PRM) that the author and a few of his students are perhaps the only programmers who use them. We find them valuable for computations with matrices, power series, and numerical quadrature.

The Counting Mode

This mode deals with over/underflow in a way which permits programmers to save all the significant digits which are lost by the other modes, and is specially useful for evaluating expressions like

$$q = \prod_1^N (a_i+b_i)/(c_i+d_i)$$

when q is likely to be a reasonable number even though its partial products and quotients are afflicted with over/underflow. The execution of

CALL FPTCT(J) ,

where J is the name of an integer variable, switches .FPTRP in 14 microseconds to the Counting Mode and designates cell J to act as a leftward extension for the 8-bit characteristics of the AC and MQ registers. Henceforth, over/underflows are counted in J . Whenever an arithmetic operation overflows, its result is divided by 2^{256} and J is increased by 1 . Whenever an arithmetic operation underflows, its result is multiplied by 2^{256} and J is decreased by 1 .

For example, the FORTRAN statements

```
CALL  FPTCT(J)
J = 0
X = (A+B)*(C+D)*(E/F)/G
```

produce a pair (J,X) whose values really satisfy

$$(A+B)(C+D)(E/F)/G = 2^{256J} X \quad .$$

In effect, the missing binary digits in X's characteristic have been added to J while X's other significant binary digits have remained unchanged.

FORTRAN programmers who use the Counting Mode must be reasonably familiar with the workings of the compiler so that they will not try to evaluate expressions like

A/(B+C) nor A*B+C nor A**B

in one FORTRAN statement.

The following example shows how the Counting Mode is used to evaluate

$$q = \prod_1^N (a_i+b_i)/(c_i+d_i)$$

for large N with no over/underflow tests inside the D⊙ loops, although each over/underflow does cost 35 microseconds.

```
   J = 0               Initialize Over/Underflow Counter
   PAB = 1.                       Numerator, and
   PCD = 1.                       Denominator.
   CALL  FPTCT(J)      Switch to Counting Mode.
    D⊙ 1  I = 1,N                        Compute Denominator using
1    PCD=RND(PCD*RND(C(I)+D(I)))         Rounded Arithmetic
    IF(PCD .EQ. 0.0) G⊙ T⊙ 3           ...because Numerator vanished.
    J = -J                              Reverse meaning of Counter  .
    D⊙ 2  I=1,N
2    PAB=RND(PAB*RND(A(I)+B(I)))        Compute Numerator
    Q = PAB/PCD
    CALL  FPTST       Switch back to Standard Mode  .
  IF (Q .EQ. 0.0)     J=0              ...because Numerator vanished.
  IF (J)  4, 5, 3
3  ...Q has Overflowed, because  J > 0 or Denominator = 0   .
        ...
4  ...Q has Underflowed, because J < 0  .
        ...
5  ...Q is correct as it stands, because  J = 0  .
        ...
```

Whatever value J may have, and provided the denominator PCD is non zero, the stored value Q is related to the desired value q by

$$q = 2^{256J} Q \quad .$$

The Counting Mode works for both single and double precision arithmetic, and is indispensable for computing determinants and certain ratios of factorials, but I have not yet figured out how to make a Complex Counting Mode work with comparable elegance on our machine. However, the next example is one where our Counting Mode is useful in a complex arithmetic calculation.

Suppose the complex array Z(I) is given and we seek K such that

$$\text{CABS}(Z(K)) = \max_{1 \le I \le N} \text{CABS}(Z(I)) \quad .$$

(Here CABS(Z) = |Z| in FORTRAN IV.) To avoid the square roots, we may prefer to calculate only squared magnitudes, thereby exploiting the equivalence between the statements

$$|a + ib| > |u + iv| \qquad \text{(i)}$$

and

$$a^2 + b^2 > u^2 + v^2 \qquad \text{(ii)}$$

But the squared magnitudes may over/underflow despite that the magnitudes |a + ib| and |u + iv| are well within the machine's range. The following program exploits the equivalence between (ii) above and

$$(a-u)(a+u) > (v-b)(v+b) \qquad \text{(iii)}$$

and then copes with over/underflows via the Counting Mode. N is assumed to exceed 1 .

```
      C⊙MPLEX  Z(...), C, W
       DIMENSI⊙N  ABC(2), UVW(2)
       EQUIVALENCE (C,ABC,A), (B,ABC(2)), (W,UVW,U), (V,UVW(2))
C      This EQUIVALENCE makes  c=a+ib  and  w=u+iv   .
      CALL FPTCT(J)
      K=1                          Initialize current maximum.
      C=Z(1)
      D⊙ 5  I=2,N
       J=0
       W=Z(1)
       XL = (A-U)*(A+U)
       J= -J
       XR = (V-B)*(V+B)
       IF(XR .EQ. 0. .⊙R. XL .EQ. 0.)  G⊙ T⊙ 3
       IF(J) 2, 3, 1
C        J>0 means |XR|  should exceed |XL|, so ignore  XL .
  1     IF(XR) 5, 5, 4
C        J<0 means |XL| should exceed  |XR|, so ignore  XR .
  2     IF(XL) 4, 5, 5
C      J=0 means  XL  and XR  are directly comparable.
  3   IF(XL .GE. XR) G⊙ T⊙ 5
  4    K=I                  Update current maximum whenever
       C=W                         W > C  .
  5   C⊙NTINUE
      CALL FPTST
```

Now C = Z(K) is the largest in magnitude of the values Z(I) . Some minor refinements can be introduced to reduce the influence of roundoff in critical cases of near equality, but they do not change the relative speed and simplicity exhibited by this program when compared with alternatives. (For more details, see our library program CMAXA in the PRM.)

An attempt was made to extend the idea of FPTCT to cope with integer overflows; i.e. we wanted to allow the FORTRAN programmer to designate a cell which would act as a leftward extension of the

integer accumulator in the same way as J in FPTCT(J) acts as a leftward extension of the floating point accumulator's characteristic. However, this scheme would first have required certain modifications to the 7094 to permit trapping on fixed point overflow, and then the FORTRAN IV compiler would have had to be extensively rewritten. A frustrating feature of the present compiler is that it renders certain integer overflows undetectable! Consequently, FORTRAN programs which manipulate large integers are very much complicated by the need for frequent overflow tests in advance of arithmetic operations.* The same complication afflicts ALGOL and any other programming language I know; it is the price we must pay for a lapse in communication among the architects, implementers and users of a programming language.

A similar lapse has frustrated attempts so far to implement the Unnormalized and Counting Modes upon some other machines. The B5500 discards one of the digits in the characteristic of an over/under-flowed result, thereby preventing any analysis from determining whether the result over/underflowed by a little or by a lot. The IBM 360 series wantonly destroys everything, including the sign of an overflowed result.‡ The CDC 6600 has its own fixed ideas about over/underflow. The tendency of other high-performance machines, like the IBM 360/91, to suffer from imprecise interrupts implies that those machines will have to deal with over/underflow entirely in their hardware. <u>This in turn implies that their treatment of over/underflow will be intolerable unless numerical analysts act soon to lay down reasonable guidelines for machine designers to follow.</u>

---

* These overflows can cause embarrassment if they are ignored; see R. Korfhage, Bulletin Amer. Math. Soc. <u>70</u> (1964), pp.341-2, and the retraction on p.747.

‡ In Feb.1967, IBM undertook to remedy these and other of the less attractive aspects of the 360's floating point hardware. There have been significant improvements. See IBM's Form A22-6821-7, and an article by A. Padegs in IBM's System Journal <u>7</u> (1968) pp. 22-29.

Improper Divisions

On a 7094 with divide-check-trap hardware, improper divisions do not turn on the divide-check indicator. Instead they trap to .FPTRP which, in our system, responds as illustrated below.

1.0/0.0 = 1.7014×10^{38} and Overflow occurs.

Any floating point division (single precision, double precision, or complex) of a non zero number by zero is treated as a quotient overflow and sets ⊙VFL⊙W = .TRUE. . No provision has been made to distinguish such divisions by zero from other quotient overflows (except in the Counting Mode, where a message can be produced) because both events almost always have the same causes and consequences. Besides, the programmer can easily (and should) test directly whether a divisor is zero or not. Each division by zero consumes more than thrice as much time as any other overflow.

1/0 = Kickoff unless otherwise has been requested.

Fixed point integer division by zero is almost certainly a drastic error in a FORTRAN program. In ALGOL the issue might not be so clear.

0.0/0.0 = Kickoff unless otherwise has been requested.

Floating point division of zero by zero is a symptom of a serious flaw in the analysis behind a program.

Unnormalized Division may kick off unless otherwise has been requested.

Floating point division by an unnormalized number causes a trap (unless the quotient produced by the hardware happens to be correct). This is a symptom of certain programming errors like

reference to a variable whose value has not previously been set,
AL⊙G(3) instead of AL⊙G(3.0) ,
a forgotten EQUIVALENCE (A,I) ,
reference to A(13) when DIMENSI⊙N A(6) , or
a significant underflow in an Unnormalized Mode.

After the new .FPTRP was installed, failures began to show up in programs which had previously been allowed to proceed silently with a zero quotient for each improper division. A few programmers protested that they liked the old ways better, but they seem to represent a lunatic fringe among programmers as a whole. The author is under the impression that the new .FPTRP's treatment of improper divisions is more widely appreciated than all his other works put together; actually the credit should be shared with R. Jones and J. Bell, who found a way to simulate the divide-check-trap hardware on a 7094 without that equipment. (The equipment is soon to be installed, and with it will come some system simplification.)

However, the most important contribution made by the new .FPTRP is that a programmer who has to cope with a complicated numerical problem can still write whatever program first comes into his mind, just as he did before. And now he will rest assured that, should his algorithm be frustrated by over/underflow, he will find out what happened and, perhaps, be able to cope with his difficulty by simply re-coding a small part of his program instead of laboriously devising a deeper mathematical analysis of his problem. The new .FPTRP strengthens the programmer's most valuable tool, hindsight.

Simulated Over/Underflow in Library Programs

The concept of over/underflow is normally associated with the elementary arithmetic operations, but it takes no imagination to extend the concept from simple functions of X like

A+X , A*X , A/X , X**2

to more complicated functions like

L⊙G(X) , EXP(X) , C⊙T(X) , ...

In general, it seems reasonable to associate overflow with the following behaviour:

as $x \to x_o$ (x_o may be $\pm \infty$), $f(x) \to \pm \infty$.

e.g. as $x \to 0+$, $\log(x) \to -\infty$;

as $x \to +\infty$, $\exp(x) \to +\infty$.

And underflow might just as reasonably be associated with this behaviour:

as $x \to \pm \infty$, $f(x) \to 0$.

e.g. as $x \to -\infty$, $\exp(x) \to 0$.

But we should not like to associate underflow with the value $\log(1) = 0$. In other words, underflow occurs only when the value of the function f(x) is not zero though closer to zero than the underflow threshold.

Here are some examples to illustrate how our functions behave in FORTRAN:

L⊙G(0.0)	≈	-1.7014 E38 and	⊙VFL⊙W is set
C⊙T(±0.0)	≈	±1.7014	⊙VFL⊙W
EXP(3000.)	≈	1.7014 E38	⊙VFL⊙W
EXP(-3000.)	=	0.0	UNFL⊙W
(±0.0)**(-3.0)	≈	±1.7014 E38	⊙VFL⊙W
0.0**(-3.0)	≈	1.7014 E38	⊙VFL⊙W
(-100.)**(-25)	=	-0.0	UNFL⊙W

The last example is interesting because the IBM program signals overflow during the computation; we avoid overflow by computing (1./100)**25 instead of 1./(100.**25) . The previous two examples should not be confused with (integer)

0**(-3) = Kickoff , code 25 ;

the distinction is consistent with the rules for improper divisions. Finally, no underflows occur when L⊙G(1.0) = 0.0 or when SINPI(X) = sin πX vanishes for integer values of X .

I have rewritten several of the elementary function subprograms in the IBLIB library to ensure that their over/underflow behaviour is consonant with the foregoing. When necessary, over/underflow is simulated; this merely means that a transfer to .FPTRP is forced in such a way that the FPT indicator word (cell 0) contains just the information needed for the desired message from .FPTRP . The simplest way to do this in a FORTRAN program is to square a very large or very small number. Of course, .FPTRP must be operating in one of its Standard Modes to allow such simulated over/underflows to produce their intended effects. If the Printing Mode is in use, as it should be while a program is being debugged, then the error-trace points to the function which caused the apparent over/underflow; otherwise the post-execution message may sometimes identify that function. As far as I can see, no vital information is lost by thus failing to discriminate between the simulated over/underflows and the others. The user's view of the library programs becomes less cluttered by their various demands for valid arguments. And the system gains several storage locations vacated by superfluous messages.

However, some programmers claim that one desirable capability has been lost. For example, they would prefer to be able to write

```
CALL KIK⊙PT (9,0)
```

in their main program whenever they want references to `L⊙G(X)` in all their subprograms to cause kickoff when `X = 0.0` . My scheme requires that each appearance of `L⊙G(X)` be preceded by something like

```
IF (X .EQ. 0.0) CALL UNCLE(9,18H L⊙G(X=0.0) ERR⊙R )
```

I think that programs written the second way are easier to read and to debug; but anyone who wants to live dangerously can easily change the library programs to suit himself because their listings are usually amply supplied with comments.

A more penetrating criticism of my scheme is that it denies too many users the valuable education obtained by reading certain IBM diagnostics. For example, increasingly many of our users have too little familiarity with the rate of growth of $\exp(x)$ to appreciate that $\exp(88.0297)$ exceeds the overflow threshold. Our university used to include a professor whose first assignment to freshman physics students was to plot a graph of $\exp(x)$ for $0 \le x \le 10$. His attitude might well serve as an example for the socially acceptable computer systems of the near future.

The extension of a comprehensive treatment of over/underflow over the entire library of numerical subprograms is an enormous task prodigiously demanding of attention to detail. Here is a simple example of a typical detail. The `CABS` function computes the absolute value of a complex variable using the formulae

$$|a + ib| = |a|\sqrt{(1 + (b/a)^2)} \quad \text{if} \quad |a| > |b|$$

$$= |b|\sqrt{(1 + (a/b)^2)} \quad \text{if} \quad |b| \ge |a| \quad .$$

For simplicity assume the former case. Then underflow will occur during the computation of $1 + (b/a)^2$ whenever $(b/a)^2$ is nonzero but smaller than the underflow threshold. This underflow is irrelevant, so our CABS program suppresses it. Had the program been written in FORTRAN the suppression would have been accomplished by computing $1 + (b/a)^2$ in the Unnormalized Mode. Similar but more complicated considerations affect the division of one complex number by another.

The task of taming over/underflow in the library is not yet completed; there are several relatively rarely used programs that remain to be revised. Is this project worth its price? Who should say? Our users can no longer offer a qualified opinion because so few of them are now aware of the issues, and even those few hardly ever have trouble dealing with over/underflow nowadays.

Addendum (June 1968)

Currently machines are being produced which exploit parallelism and pipeline principles to achieve extremely high processing speeds, but at the cost of what are called “imprecise interrupts”. The problem is illustrated by the following sequence of FORTRAN code.

```
       ...
 7     A = ...
 8     B = C*D/E
 9     IF (⊙VFL⊙W) G⊙ T⊙  999
10     I = I + 1
11     F = (A+G)/B
       ...
```

A typical sequence of events in the computer's central processing units will be described on the assumption that none of the variables A, B, C, D, E, F, G, I or ⊙VFL⊙W share storage by virtue of an EQUIVALENCE statement.

After instructions for statement 8 have been fetched, and
while the value intended for A is being computed,
C is fetched (from storage),
D is fetched,
the value of A is delivered ready for storing,
instructions for statement 9 are fetched,
multiplication of C*D is initiated,
A is stored,
E is fetched,
⊙VFL⊙W is fetched,
⊙VFL⊙W is tested and found to be .FALSE. ,
instructions for statement 10 are fetched,
I is fetched,
the product C*D is delivered ready for use.

If C*D has overflowed, a flag is set now to recoed the event; if the overflow is going to be allowed to interrupt the system, another flag is set to inhibit any further fetches of instructions. The value of C*D is replaced by something else and processing contimues.

The division (C*D)/E is initiated,
1 is fetched,
the sum I+1 is formed in a fast integer adder,
instructions for statement 11 are fetched (unless a flag is set),
I = I+1 is stored,
A is fetched (unless ...),
G is fetched (unless ...),
the floating addition of (A+G) is initiated (unless ...)
fetching B is inhibited by instructions held over from statement 8 ,
the quotient (C*D)/E becomes ready for storage into B
and use in statement 11 .

This is the earliest point at which overflow in C*D can suspend the normal sequence of execution without leaving fragments of partially executed instructions circulating in the central processor; but the time is too late because instruction 9 has been passed.

The foregoing sequence gives only a rough illustration of the problem because details of machine design vary considerably from model to model. Among the machines which suffer from some form of imprecise interrupt (at this date) are the CDC 6600 and 7600, the IBM 360/91, and the Burroughs B 8500. Over/underflows on these machines are dealt with by their hardware in a manner similar to our Standard Silent Mode. A program's every attempt to deal with over/underflow more flexibly is frustrated by the hardware. For example, there is no easy way to tell whether a computation has overflowed only slightly or by a lot; there is no easy way to distinguish between important and unimportant underflows as we do in the Unnormalized Modes; division by zero is always treated as a disaster.

There seems to be no way to improve these machines' treatment of arithmetic exceptions that does not involve substantial changes to the hardware. We shall offer here two suggestions which confine the changes to the floating point part of the central processor.

One possibility is to micro-program facilities comparable to our five Modes into the hardware. Such a micro-program does not have to run at the same high speed as the rest of the hardware because a modest loss of speed on rare occasions is inconsequential.

A second possibility is to lengthen the central processor's registers so that they may hold numbers lying beyond the range normally held in storage, thereby permitting expressions of modest complexity to be evaluated correctly despite what might otherwise be over/underflow in sub-expressions. Consequently, over/underflow need occur only when information is lost by an attempt either to store a number that cannot be fitted into storage, or to push the contents of a register beyond its extended range.

Both possibilities are complicated, but not as complicated as the lengths to which programmers will occasionally be forced to go to deal with arithmetic exceptions on those machines.

Samples of Library Program Write-Ups from

The PROGRAMMERS' REFERENCE MANUAL,

3rd edition, Aug. 1967,

Univ. of Toronto, Institute of Computer Science.

(Extracted for a Summer Course at the Univ. of Michigan, June 17-21, l968.}

The following FORTRAN functions are described herein:
A**B , DP**DQ , CABS , COS/SIN , COSPI/SINPI ,
Complex arithmetic , CSQRT , DQBRT , DSQRT , EXP ,
TWOXP , (Variable)**(Integer) , LOG , LOG 10 ,
LOG 2 , Max/min over arrays, QBRT , SQRT .

These have been coded to run under an ICS-modified version of IBM's IBSYS v.13 on the 7094-II. They differ from programs supplied by IBM mainly in two respects:

i) All programs conform to the ICS conventions concerning over/underflow, contentious values like 0**0 , and diagnostic options and messages.

ii) All claims to accuracy have been proved mathematically by the programmer; this provides no guarantee of accuracy since proofs are as vulnerable to error as are programs. Also every program has been tested for accuracy and speed on tens of thousands of sample arguments, including critical values appropriate to the function and to the program under test. No claim has been refuted by any test. For five years, every user of IBM's Fortran IV on the ICS's machine has used these functions instead of IBM's;

nobody has complained yet (June 1968).

The programs described herein are some of the latest versions of programs written in 1962-3 for a 7090 at the Univ. of Toronto to replace the appalling FORTRAN functions supplied by IBM at that time. Meanwhile IBM's elementary FORTRAN functions have improved considerably, and now two excellent collections of programs are distributed with IBSYS v.13 for the 7090/7094 and with FORTRAN IV (E.G.H.) for System/360. These programs were produced at the University of Chicago chiefly by Mr. Hirondo Kuki, whose work is described in

- "MATHEMATICAL FUNCTIONS", a description of the Univ. of Chicago Computation Center's 7094 Math. Function Library, by H. Kuki, with a foreward by C.C.J. Roothan, Feb. 1966.

IBM 7090/7094 IBSYS v.13 IBJOB Processor manual, appendix H; Form C28-6389 (Mar. 1966).

IBM System/360 FORTRAN IV Library Subprogram; Form C28-6596-2 (1966).

"Performance Statistics of the FORTRAN IV (H) Library for the IBM System/360" by N.A. Clark, W.J. Cody, K.E. Hillstrom and E.A. Thieleker, Argonne Nat'l Lab. Report ANL-7321, May 1967.

The last report, plus additional work by W.J. Cody at Argonne and by L.R. Turner at NASA's Lewis Res. Center, showed that Kuki's programs for /360 were not as accurate as the best comparable programs on some other machines. Most of the trouble was attributable to oversights in the design of /360's floating point hardware which Kuki had recognized in 1964. He worked self-effacingly on a SHARE committee which, in 1966-67, persuaded IBM to remedy most of these oversights; the fruits of that effort appear among the differences between releases -6 and -7 of "IBM System/360 Principles of Operation" Form A22-6821 (1967-68). He has now improved his /360 programs, taking advantage of better hardware and better algorithms described in forthcoming releases of C28-6596, to the point where the rest of the computing industry will do well if it can match his example.

The ICS programs described herein perform better (though seldom by much) than comparable programs for the 7094 distributed by IBM; this is as it should be because the ICS versions were produced for a purpose that cannot be justified commercially – to approach perfection. The design priorities were these:

i) Performance will be judged solely by what has been proved mathematically taking roundoff, over/underflow and all other aspects of the 7094-II into account. The tests are intended to check the proof, not the program; the proof is wrong except perhaps when the program's tests turn out just slightly better than predicted

ii) Freedom from exceptions is valued most highly, Accuracy second, Speed third, Storage economy fourth.

A mathematical model exists which is worth keeping in mind when appraising any program. Let $F(X)$ be the numerical value stored by a program intended to compute $f(x)$. In general, the best that can be proved is a relation

$$F(X) = (1+\varepsilon)\, f((1+\delta)X)$$

in which ε and δ represent errors for which we seek the smallest possible bounds. Some trade-off is possible between ε and δ insofar as the bound upon one may be reduced at the expense of increasing the other's. The simplest and most desirable case is that when $\delta \equiv 0$, so that ε can be regarded as “the” error in $F(X) = (1+\varepsilon)\, f(X)$. However, there are occasions when δ cannot be suppressed; see the write-ups for COS/SIN. Occasionally δ can be suppressed only at an intolerable cost; see DP**DQ .

We have attempted to keep $\delta = 0$ and to keep ε well below 1 ulp (unit in the last place stored). We have also attempted to preserve familiar properties of *f(x)* like monotonicity, symmetry, simple identities and well-known special values as far as possible in the computed approximation *F(X)* . Examples are

SQRT(*X***2) = ABS(*X*) ,

SIN(*X*)**2 + COS(*X*)**2 = 1.0 within 3 ulp ,

LOG(*X*)/(*X*-1.) $\rightarrow$ 1.0 within 3 ulp as *X* $\rightarrow$ 1.0 ,

SIN(*X*)/*X* $\leq$ 1.0 , ...

Our motives for these attempts have been echoed recently by H. Kuki in a memorandum "Comments on the ANL evaluation [ANL-7321 by Clark, Cody et al.] of OS/360 FORTRAN Math Library" wherein he says on p.4

> "a. It is the strictest accuracy requirement for subroutines one can conceive.
>
> b. Therefore it gives the simplest goal for programmers to aim at so far as accuracy is concerned.
>
> c. In some computations (e.g. integral arguments, assuming all prior computations went meticulously well) where there is no error in the argument, the benefit is real.
>
> d. It is simpler to explain to the users.
>
> Of these reasons, it seems to me the last is most important. After all, coding a subroutine is only half the work, and the remaining half consists of informing the users what exactly the subroutine accomplishes.
>
> ...
>
> ... but it may cost diamond where mere glass may do..."

To help appraise the costs, here are some characteristics of the IBM 7094-II:

Storage: 32678 words, 36 bits each, 1.4 μsec cycle, 2-way interleave.

Speed: Most instructions take one word and 2.8 μsec. Single precision floating point operations take about 4.2 μsec. to add, 5.6 to multiply, 8.4 to divide; double precision takes about twice as long.

Single precision: 27 significant bits; 1 ulp is a relative error between $.75_{10}^{-8}$ and 1.5_{10}^{-8} .

Double precision: 54 bits, of which the last two or three are smeared by double precision multiplication and division.

To prove mathematically that our programs perform at least as well as is claimed may at first appear to be a formidable task, especially when the error is claimed to be so small; e.g.

< .50000163 ulp	for	SQRT and complex arithmetic
.52		LOG and QBRT
.77		EXP
.854		CABS
1.0		COSPI, SINPI, DSQRT, DQBRT.

The proofs were carried out in 1962-5 via lengthy computations with both decimal and octal desk calculations. Nowadays most of the proofs would be regarded as routine applications of high-precision Interval Arithmetic. Here is an example.

Nominally, if $2^{-14} \le |f| \le \frac{1}{2}$,

$$\mathrm{SINPI}(f/2) \equiv (((s_3 f^2 + s_2)f^2 + s_1)f^2 + \pi/2)f$$

for certain constants s_i ; actually the machine generates

$$f_2 \equiv f^2(1+\varepsilon_1) \quad ,$$

$$\mathrm{SINPI}(f/2) \equiv \Big(\big((s_3 f_2(1+\varepsilon_2) + s_2)(1+\varepsilon_3) f_2(1+\varepsilon_4) + s_1 \big)(1+\varepsilon_5) f_2(1+\varepsilon_6) + (1-\delta)\pi/2 \Big)(1+\varepsilon_7) f (1+\varepsilon_8)$$

where each ε_i represents a rounding error committed after an addition or multiplication, and δ is the error committed by truncating $\pi/2$ to 27 bits. We wish to compare $\mathrm{SINPI}(f/2)$ with $\sin(\pi f/2) = f(\pi/2 + \sum_1^6 \sigma_i f^{2i} + \varepsilon_9 \sigma_7 f^{14})$,

where σ_i = (suitable constant) and $\varepsilon_9 = \varepsilon_9(f) \in]0, 1[$,

taking into account the fact that each ε_i and δ is bounded in a way that can be inferred rigorously from the published characteristics of the 7094.

The comparison is effected first by rearranging the symbolic expression for

$$\big(\mathrm{SINPI}(f/2) - \sin(\pi f/2)\big)/\sin(\pi f/2)$$

in a way which achieves as much symbolic cancellation as possible, second by computing appropriately precise interval approximations for the constants σ_i and $\pi/2$ (the s_i are taken precisely out of the program SINPI), and third by using interval arithmetic to overestimate the range of values taken by the expression as all ε_i and δ vary independently over their ranges.

In general, given any function $f(x)$, ingenuity may be needed to choose a good formula for approximating $f(x)$, find best values for constants, write an efficient program $F(X)$, and to rearrange $F(X) - f(X)$ symbolically in a way suitable for Interval Analysis, but the rest is routine.

W. Kahan
Univ. of Toronto
June 1968