A program to calculate the GAMM measure

B. A. Wichmann and J. Du Croz*

Computing Services Unit, National Physical Laboratory, Teddington, Middlesex TW11 0LW

A FORTRAN program is given whose execution time is proportional to the GAMM measure of computer performance. An error analysis is given of the floating point calculation in the program. This analysis permits the program to be used as a confidence check on the accuracy of the floating point facilities of the computer.
(Received December 1978)

Introduction

The GAMM measure of performance is often quoted by manufacturers, mainly in Europe, to indicate the speed of a machine on scientific calculations. The measure is a weighted average of the time taken in five simple numerical loops. For any particular computer, the loops are traditionally programmed in assembly code and timed by hand using the manufacturer's machine manual. Sometimes the timing is checked by running the program, particularly with the faster machines where it is not easy to time programs by hand because of instruction look-ahead and slaving.

It is clearly more satisfactory if the programming can be done in a high level language because this will take into account the efficiency of the compiler, which is important now that scientific programming is rarely performed in assembly language. The note describes a program to calculate the GAMM loops using FORTRAN (although recoding in any other language with floating point operations would be simple).

Definition

The five loops are:

1. Addition of two vectors of order 30.
2. Element by element multiplication of two vectors of order 30.
3. Evaluation of a polynomial of order 10 by Horner's method.
4. Selection of the maximum component of a vector of order 10.
5. Computation of a square root by Newton's method with five iterations.

If the times for the complete loops are T_i microseconds, then the GAMM measure is defined by

$$\text{GAMM} = (T_1/(1*30) + T_2/(2*30) + T_3/(2*10) + T_4/(2*10) + T_5/(3*5))/5$$

The second factor in each divisor is the number of repetitions of the statements in each loop. If the initialisation (and termination code) of the loops is ignored and if t_i is the time for the code within each loop, the figure becomes

$$\text{GAMM} = (t_1 + t_2/2 + t_3/2 + t_4/2 + t_5/3)/5$$

There does not appear to be any justification for the particular weights chosen. The definition is repeated here because the original (Heinhold and Bauer, 1962) does not seem to be easily obtainable.

Construction of the program

The usual method of calculating the GAMM figure requires the timing of five loops which only take a few microseconds. It is impossible to perform this timing in a machine independent manner and, in any case, reliable timing of such short intervals is difficult on many machines (Gentleman and Wichmann, 1973). Hence the method adopted here is to use the total program times only. To do this, each of the five loops is repeated a number of times to give the correct proportion. Using the first formula above 300 GAMM units is $2*T_1+T_2+3*T_3+3*T_4+4*T_5$. Hence the program requires 13 loops: 2 additions, 1 multiplication, 3 polynomials, 3 maxima and 4 square roots. The 13 loops are nested within an outer loop to give a program whose execution time can easily be varied to suit the timing capabilities of any particular machine. This outer loop is repeated N times.

Having determined the overall structure of the program, it is necessary to complete the details to meet a further constraint. The program must not be capable of being optimised unduly by a good compiler such as the IBM FORTRAN H (OPT = 2). For instance, if an optimising compiler could detect that a complete loop could be omitted, then the program would not time the necessary loops. Legitimate optimisation includes good use of registers, using special loop control instructions, removing array address calculation out of the loop or even 'flattening' the loop by repeating copies of the loop code. To ensure that each loop is indeed compiled, all computed values should be used by the program. This requirement is not easily met since the program must not either overflow or underflow (even on machines with 16 bit integers or a small exponent in floating point). The method used is to perform exactly the same computation in each cycle of the 13 loops by permuting the values of the three arrays. Each cycle contributes to an accumulated sum and hence cannot be optimised out. At six points within the cycle, a value Y (or ROOT) is calculated. The statement

$$\text{ACC1} = \text{ACC1} + Y*\text{DIVN}$$

is then executed where DIVN = 1·0/FLOAT(N). At the end of the loop, ACC1 is added to the current value of ACC. The final accumulated value of ACC is printed at the end of the program (together with the number of repetitions of the main cycle) to ensure that the program was executed successfully.

To use the program to measure the GAMM figure for a particular computer, one of two methods can be used. Firstly, the number of repetitions of the outer loop can be set to a large value so that the time for the initialisation code and printing the results can be ignored. Alternatively, the program can be run with just one cycle to determine the overhead and this time subtracted from the time for a large number of cycles. This method is to be preferred since with many computers it is not clear how much of the loading overhead is added to the processor time given by the operating system. The method can also be used with load and go systems such as WATFOR, where the overhead would include compiling.

The program passes Fosdick's data flow analysis test (a static test that variables are initialised and values used) (Osterweil and Fosdick, 1976) with the exception of the array elements C. Hence all the elements of the arrays A and B and all the simple

*Numerical Algorithms Group, Oxford

Downloaded from https://academic.oup.com/comjnl/article/22/4/317/443277 by guest on 19 April 2024

variables are both initialised and used after being computed. This demonstrates that optimisation of the program by a good compiler is unlikely to produce code which does not follow the FORTRAN source text. The data flow analysis was checked with a facility in the BABEL compiler at NPL (Scowen, 1969).

Differences from the GAMM definition

The program has three differences from the GAMM definition as follows:

1. The time to set up the loops is counted as well as the loops themselves.
2. Initialisation of simple variables, accumulation of the final result and a main loop are added to make a complete program.
3. One loop subtracts two vectors instead of adding them. (This is to ensure that every loop performs the same calculation.)

The following choices are made which are not defined in the GAMM measure:

1. The loop control code for the thirteen loops are chosen to reflect the statistics of ALGOL 60 programs (Wichmann, 1973, page 98). This includes stepping backwards through one array, a process which cannot be handled directly by a FORTRAN DO loop.
2. In finding the maximum element of ten in a vector, the data values are such that four assignments to the maximum are necessary.

The additional computation and loop initialisation give a GAMM figure about 5% higher than the true formula (on KDF9).

Results from KDF9

The GAMM figure results obtained from versions of this program on KDF9 are as follows:

Language	*Type of compiler*	*GAMM figure*
Assembler	optimal coding	30·4
ALGOL 60	Good optimising compiler	41
BABEL (fast option)	Good non-optimising compiler	105
FORTRAN	Compiler with bound checking	276
ALGOL 60	Interpretive system	4616

Since KDF9 has about a 7 microsecond average instruction time, the GAMM figure varies from about 4 to 40 machine instructions for a true compiler. An interpreter can take much longer corresponding to about 700 machine instructions. The GAMM figure from BASIC will typically be this long.

Simple error analysis

It is hoped that the GAMM program may be run on a wide variety of machines. Since the final number printed by the program is subject to considerable rounding error, an error analysis is provided here to allow the program to be used as a confidence check on the floating point hardware.

The two numbers printed by the program are the number of loop repetitions (N) and then ACC.

It is easy to see that in the six places where ACC1 is accumulated, the value to be added is identical for each loop. The values are the currently computed value of Y, ROOT, ROOT, ROOT, Y, and ROOT added after the statements numbered 3, 5, 8, 11, 12 and 14 respectively. The six values are (for $N = 1$)

$X1 = 1{\cdot}18967\ 4523$
$X2 = 4{\cdot}58258\ 19709\ 72577\ 87482\ 60582\ 68143\ 87174\ 14293$
$X3 = 1{\cdot}04822\ 01257\ 84655\ 97785\ 78545\ 20972\ 84151\ 82396$
$X4 = 5{\cdot}56786\ 43331\ 01262\ 00469\ 24209\ 77173\ 90268\ 22309$
$X5 = 1{\cdot}02846\ 6483$
$X6 = 3{\cdot}31662\ 48052\ 31568\ 85947\ 16804\ 46747\ 11540\ 44996$

and their sum is:

TRUEACC = 16·73343 22410 90064 71684 80142 13037 73134 63994 .

In the program, each of the Xi is multiplied by DIVN (= about $1/N$) before being added to ACC1. Hence the value finally accumulated (ACC) is always about TRUEACC, since it is the sum of N terms of size about TRUEACC/N.

The second, fourth and sixth numbers arise from Newton's algorithm but convergence to the square root is not obtained for machines with a sufficiently long mantissa. The error in calculating the six numbers is of the same order as the machine accuracy since the error depends only upon the last floating point operations. This is due to the inherent stability of Horner's method and Newton's algorithm. We shall see that the likely rounding error in ACC is of order N and arises from the accumulation.

For the first few passes of the main loops, the contribution to the error is small. However, consider the case when $8 < \text{ACC} < 16$ (assuming a binary exponent with floating point). This condition will be satisfied for nearly half the values for REP. In adding ACC1 to ACC, the product must be shifted down before adding. Moreover, for these values of REP, the value lost at the bottom is the same, since ACC1 is identical and the shift is by the same amount. The same reasoning applies with the other ranges for ACC, the only cases being excluded are when ACC changes its exponent (of order log (N)).

To obtain a crude estimate of the error distribution, one proceeds as follows. For any particular value of N (and particular machine characteristics) the value of ACC printed is completely determined. This value can deviate from TRUEACC by quite a small amount even for large N, but the expected deviation is proportional to N. If the expected absolute value of the error arising from one cycle with ACC over 8 is A, then when ACC is half this, the shift before addition is one less and the number of cycles is half so that the total error is four times smaller. Hence we have

Range of ACC	*Number of loops*	*Error in one loop*	*Total error in this range*
16 to 16·73	·73*N/16·73	A	·73*N*A/16·73
8 to 16	8*N/16·73	A/2	4*N*A/16·73
4 to 8	4*N/16·73	A/4	N*A/16·73
...	...	...	...

Hence the total error from all the ranges is

N*A*(·73 + 4 + 1 + 1/4 . . .)/16·73
= 6·067*N*A/16·73 = ·3626*N*A

This analysis shows that the error is proportional to N. To obtain the actual distribution requires more care since the assumption made above that the various contributions could be summed together is not valid.

Error analysis and distribution

Consider the problem of computing $N*X$ be repeated addition, i.e.:

$S := 0{\cdot}0;$
for $i := 1$ **step** 1 **until** N **do**
  $S := S + X;$

Assume that the computer uses a standard floating point representation, with base 'b' and 't' b-ary digits in the mantissa. Let X lie in the range $[b\uparrow(q-1), b\uparrow q)$ and let the b-ary digits in the mantissa of X be:

$$x_1 x_2 x_3 \ldots x_t, \quad 0 \leqslant x_i < b .$$

Downloaded from https://academic.oup.com/comjnl/article/22/4/317/343277 by guest on 19 April 2024

At a typical stage in the computation, suppose S lies in the range $[b\uparrow(q+i-1), b\uparrow(q+i))$: then the mantissa of X must be shifted i digits to the right, so provided $S + X < b\uparrow(q+i)$, the rounding error with truncation is simply

$$D_i = .\, x_{t-i+1}x_{t-i+2}\ldots x_t * b\uparrow(q-t+i)$$

Let ACC be the final computed result: ACC is about $N*X$, and suppose ACC lies in the range $(b\uparrow(r-1), b\uparrow r)$. Then, since the number of values of X contributing to ACC while S lies between $b\uparrow(q+i-1)$ and $b\uparrow(q+i)$ is about $(b\uparrow(q+i)-b\uparrow(q+i-1)/X$, the total error due to truncation is (to a close approximation):

$$(\mathrm{ACC} - b\uparrow(r-1)) * D_{r-q}/X + \sum_{i=1}^{r-q-1}(b\uparrow(q+i) - b\uparrow(q+i-1))*D_i/X$$
$$= b\uparrow(-t)*N/\mathrm{ACC} \sum_{i=1}^{r-q} x_{t-i+1}*b\uparrow(q+i-1)*(\mathrm{ACC} - b\uparrow(q+i-1))$$

If we can assume that the low order digits of X are uniformly distributed over the range $[0, b-1]$, then each has a mean of $(b-1)/2$, and so the mean total rounding error is:

$$b\uparrow(-t)*(b-1)*N/(2*\mathrm{ACC})\,(\mathrm{ACC}*(b\uparrow(r-1) + b\uparrow(r-2) + \ldots b\uparrow q) - (b\uparrow(2*r-2) + b\uparrow(2*r-4) + \ldots + b\uparrow(2q)))$$

if one assumes that $b\uparrow(r-q)$ is negligible, then this can be approximated by

$$b\uparrow(r-t)*(\mathrm{ACC} - b\uparrow r/(b+1))*N/(2*\mathrm{ACC})\,.$$

Moreover, each digit has a variance of $(b\uparrow 2 - 1)/12$, and if we assume that the digits are independently distributed, we can sum the variances, and so the variance of the total rounding error can be calculated, and is approximately (again neglecting terms of order $b\uparrow(r-q)$):

$$((b\uparrow(r-t)*N/2*\mathrm{ACC})\uparrow 2*(b\uparrow 2 - 1)/3*(\mathrm{ACC}\uparrow 2/(b\uparrow 2 - 1) - 2*\mathrm{ACC}*b\uparrow r/(b\uparrow 3 - 1) + b\uparrow(2*r)/(b\uparrow 4 - 1))$$

The calculation of the error with rounding is more difficult. The rounding error on each addition is:

$$D_i - A_i*b\uparrow(q-t+i)\,,$$

where the D_i is defined as for truncation, and

$$A_i = 0 \text{ if } D_i < \cdot 5*b\uparrow(q-t+i)$$
$$A_i = 1 \text{ if } D_i \geqslant \cdot 5*b\uparrow(q-t+i)$$

(assuming the usual method of rounding, where 1/2 is always rounded up). The total rounding error is therefore:

$$b\uparrow(-t)*N/\mathrm{ACC}\{ \sum_{i=1}^{r-q} x_{t-i+1}*b\uparrow(q+i-1)*(\mathrm{ACC} - b\uparrow(q+i-1)) - \sum_{i=1}^{r-q-1} A_i*b\uparrow(q+i)*(b\uparrow(q+i) - b\uparrow(q+i-1)) - A_{r-q}*b\uparrow r*(\mathrm{ACC} - b\uparrow(r-1))$$

If $b = 2$, $A_i = x_{t-i+1}$ this expression simplifies to:

$$(2\uparrow(-t)*N/\mathrm{ACC})\{ \sum_{i=1}^{r-q-1} x_{t-i+1}*2\uparrow(q+i-1)*(\mathrm{ACC} - 3*2\uparrow(q+i-1))\} + x_{t-r+q+1}*2\uparrow(r-1)*(2\uparrow(r-1) - \mathrm{ACC}))$$

The mean total rounding error is now exactly $b\uparrow(-t)*b\uparrow q*(b\uparrow q - \mathrm{ACC})*N/2*\mathrm{ACC}$, i.e. of order $b\uparrow(q-r)$. To calculate the variance, we cannot assume that the A_i are independent of the x_j; however if the base b is even, the A_i depends only on x_{t-i+1} and the variance of each term of the form $x_{t-i+1}*H_i - A_i*G_i$ is:

$$(b\uparrow 2 - 1)*H_i\uparrow 2/12 - b/4*H_i*G_i + 1/4*G_i\uparrow 2$$

Hence we find that the variance with rounding is less than the variance under truncation by:

$$(b\uparrow(r-p)*N/2*\mathrm{ACC})\uparrow 2*(b-1)*b\uparrow(r-1)*(\mathrm{ACC}/(b\uparrow 3 - 1) - b\uparrow r/(b\uparrow 4 - 1))$$

approximately, and this is always positive since $\mathrm{ACC} \geqslant b\uparrow(r-1)$.

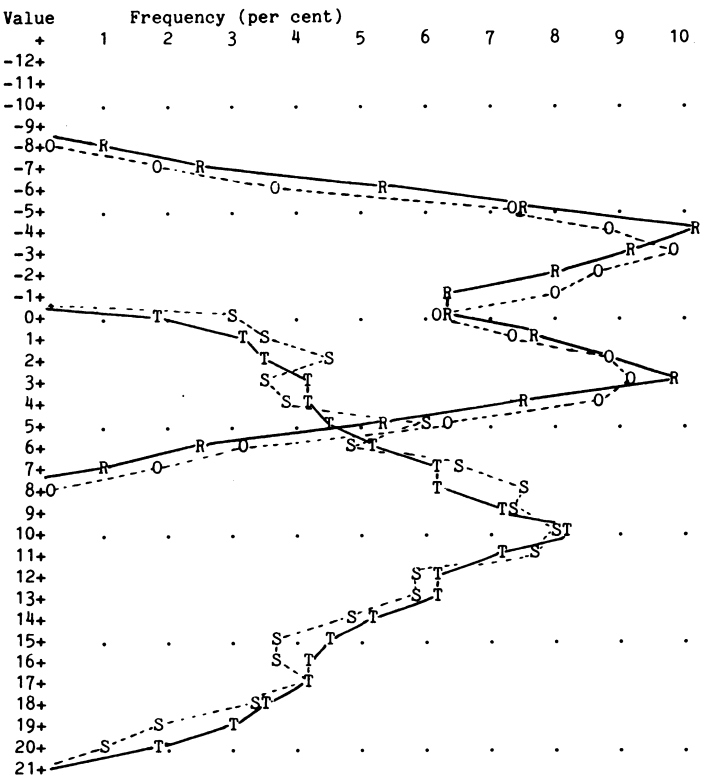

Fig. 1 Graph of error distribution for base 2 arithmetic
Value plotted is $(\mathrm{TRUEACC} - \mathrm{ACC})*2\uparrow t*1{\cdot}81/N$.
R = Rounding, theoretical
O = Rounding, observed (KDF9)
T = Truncation, theoretical
S = Truncation, observed (CDC 7600).

Observed and calculated distribution function

A program has been written to calculate the distribution of the errors with truncation. The program calculates the first few terms of the error assuming N is large. The truncation error simplifies to

$$b\uparrow(-t)*N*b\uparrow r/\mathrm{ACC} \times \{\sum_{i=1}^{-\infty} x'*b\uparrow i*(\mathrm{ACC} - b\uparrow i)\}$$

where the x' are independently distributed digits to base b.

Similarly the formula for the mean becomes

$$b\uparrow(-t)*N*b\uparrow r*(\mathrm{ACC} - b\uparrow r/(b+1))/2*\mathrm{ACC}\,.$$

and the formula for the standard deviation becomes

$$b\uparrow(-t)*N*b\uparrow r*\mathrm{sqrt}((b\uparrow 2 - 1)*(\mathrm{ACC}\uparrow 2/(b\uparrow 2 - 1) - 2*\mathrm{ACC}* b\uparrow r/(b\uparrow 3 - 1)b\uparrow(2*r)/(b\uparrow 4 - 1))/3)/2*\mathrm{ACC}$$

Three runs were done with the program taking 4096 values in each case. This is 12 terms for base 2, 4 terms for base 8 and 3 terms for base 16. A histogram was produced for each case by dividing the maximum range into 21 parts (to give a symmetric output). A little care is necessary to ensure that the histogram handles the end points correctly. This was done by reducing the interval by a small amount, which resulted in very small deviations from the expected symmetry. The output is illustrated in the graphs given for each base (**Figs. 1, 2 and 3**)

Another program has been written to examine experimentally the distribution of the rounding error. The program can run quickly if the redundant calculation is removed from the loop (i.e. the repeated calculation of ACC1). Runs have been made for 4096 different consecutive values of N with base 2 truncation (CDC 7600), base 2 rounding (KDF9), base 8 truncation and

Downloaded from https://academic.oup.com/comjnl/article/22/4/317/343277 by guest on 19 April 2024

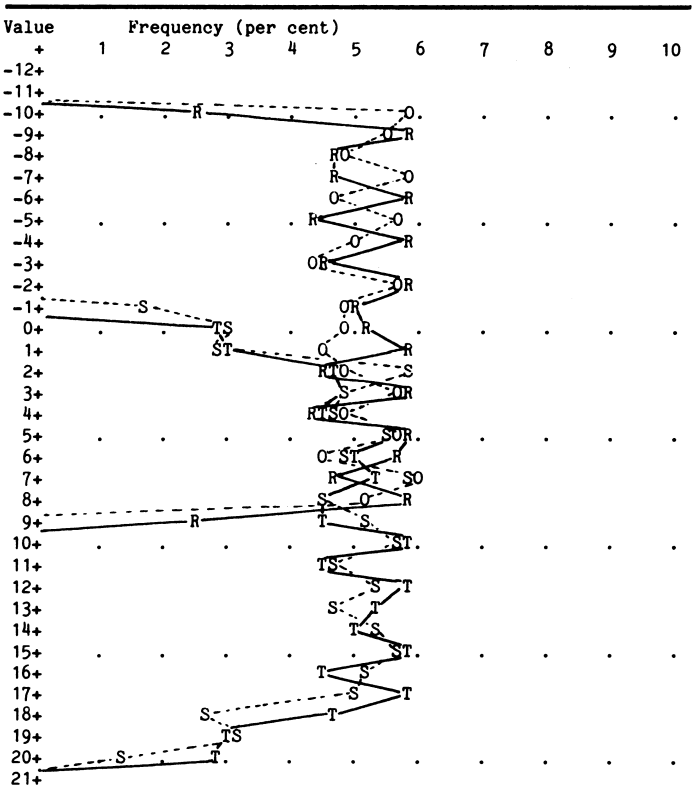

Fig. 2 Graph of error distribution for base 8 arithmetic
Value plotted is (TRUEACC − ACC)*8↑t*·571/N.
R = Rounding, theoretical
O = Rounding, observed (B5500, single precision)
T = Truncation, theoretical
S = Truncation, observed (B5500, double precision)

rounding (B5500) and base 16 truncation (370). These results also appear on the graphs. Standard FORTRAN programs are available from NPL to produce these graphs for both single and double precision. A charge is made for these programs.

The visual agreement between the actual and theoretical distributions is seen to be good although a chi-squared test fails. This is thought to be due to the correlation in the low order digits for ACC1 for consecutive N. The binomial expansion for $N + 1$ in terms of that for N will converge in 4 or 5 terms on most machines.

It can be seen that the distributions are by no means normal, and indeed some are almost rectangular. Hence the maximum range for the distribution gives a very good acceptance test; results obtained outside this are extremely likely to indicate an error (hardware or software). The different shape is caused by the different magnitude of the first term—which is in turn due to where 16·7 (TRUEACC) is placed relative to the powers of b.

A similar program was written to calculate the distribution with rounding. The output from these calculations is incorporated in the graphs for each base.

Use of the program as a confidence check

The error analysis above allows the program to be used as a confidence check. For any value of N, the deviation between the computed value of ACC and its true value will vary in a random manner. Hence any one computer run with a large value of N (>200) cannot reveal much about the machine accuracy. A single run could suggest an error in either software or hardware only if the deviation is much greater than the expected amount.

To check a single value one needs to know the machine characteristics. The three important factors are rounding/truncation, the mantissa length (t) and the base of the exponent (b). Some typical machine characteristics are:

Machine	*Options*	*t*	*b*	*Round/truncate*
IBM 370	Single	6	16	truncate
	Double	14	16	truncate
CDC7600	Single, round	47	2	round
	Single, truncate	48	2	truncate
	Double	96	2	truncate
ICL1906A	Single	37	2	round
	Double	74	2	round
KDF9	Single	39	2	round
	Double	78	2	truncate
B5500	Single	13	8	round
	Double	26	8	truncate

To perform the check one proceeds as follows:
The quantity B = (TRUEACC − ACC)*b↑t/N is calculated. This is an estimate of the error which is independent of the machine accuracy and the value of N used.

The value of B must now be compared with the relevant graphs. Each graph uses a different scale to accommodate the varying spread. If a value falls outside the bounds shown on the graph by more than one unit, there is strong reason to believe an error in the floating point hardware or software.

If the above check indicates a strong possibility of an error, then the program should be rerun with $N = 1$. Any large deviation for $N = 1$ is clearly serious, but a small variation may occur in the lowest digits merely due to the accuracy of the FORTRAN output package.

An important cross-check that should be performed whenever possible is to run the program with different compiling options

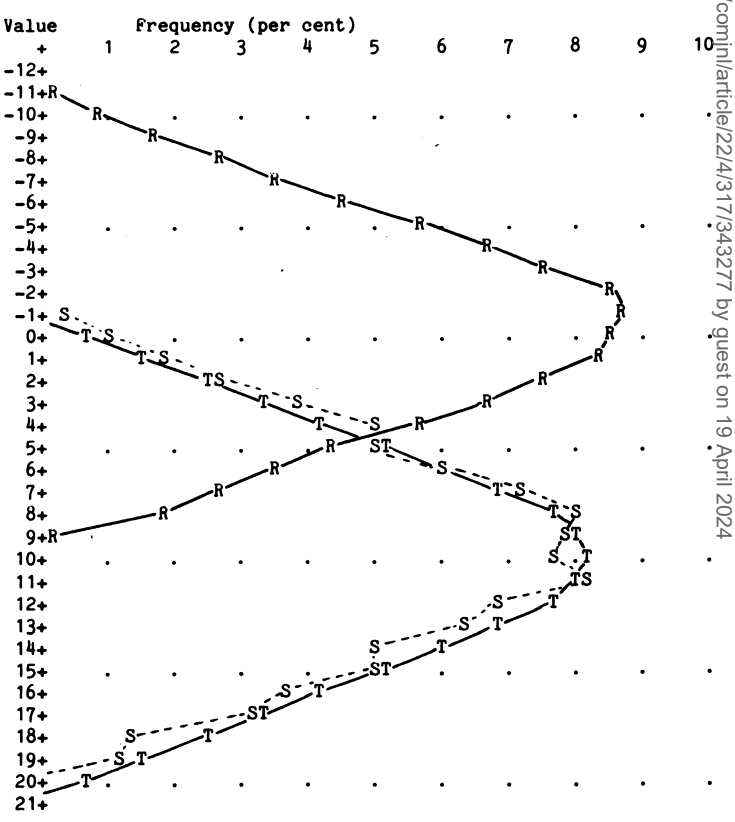

Fig. 3 Graph of error distribution for base 16 arithmetic
Value plotted is (TRUEACC − ACC)*16↑t*·820/N.
R = Rounding, theoretical
T = Truncation, theoretical
S = Truncation, observed (IBM 370/168, single precision)

Downloaded from https://academic.oup.com/comjnl/article/22/4/317/343277 by guest on 19 April 2024

(diagnostics, etc.). Such options should make no difference at all to the output printed. The same is true of versions of the program coded in different languages but run on the same underlying hardware. If different results are obtained, a check with $N = 1$ should be made and the relevant manuals consulted to see if any deviations have a simple explanation. A diagnostic FORTRAN compiler could calculate all real expressions to double precision which would clearly give marginally more accurate results. If the results are very accurate (say always on the zero position on the graph), then it is possible that 't' has been given the wrong value or that the precision is more accurate than the manual states.

If a fault cannot be located but this program gives results repeatedly outside the bounds shown on the graph, the vendor of the compiler and hardware should be consulted. The computer runs should be supplied together with a copy of this report.

Acknowledgements

We are grateful to Dr L. Thomas of the Burroughs Corporation for performing the calculation of ACC to 40 digits on a B1700. Dr M. G. Cox from NPL provided considerable assistance with the error analysis. Many helpful comments were made on a draft of this paper by Dr B. Smith of the Argonne Laboratory. Mr C. Harman produced the FORTRAN program which was used to give the histograms (Figs. 1, 2, 3). Mr R. Harvey of the Post Office was kind enough to run the programs on the B5500 and 370/168 to give the distributions for an octal and hexadecimal machine.

Appendix 1 Program listing

The single or double precision version of this program can be constructed from the other version by means of the following editing interchanges:

single precision	*double precision*
REAL	DOUBLE PRECISION
E + 0	D + 0
E30·22	D40·30

Also, to give a comparable length of time, the initial assignment should be changed to $N = 3500$ for the double precision version. The initial comments should also be altered. Both programs can be provided on paper tape.

```
C  GAMM IN FORTRAN  - SINGLE PRECISION VERSION, MARK 3
C NATIONAL PHYSICAL LABORATORY BENCHMARK GAMM F
C THIS PROGRAM HAS A SINGLE PARAMETER N
C OUTPUT IS BY ONE WRITE STATEMENT TO DEVICE 6
C SET N = 10000 FOR ABOUT ONE MINUTE ON MACHINE LIKE 360/65
      INTEGER FIVE, I, J, N, REP, TEN, THIRTY
      REAL ACC, ACC1, DIVN, RN, ROOT, X, Y
      REAL A(30), B(30), C(30)
      N = 10000
      FIVE = 5
      TEN = 10
      THIRTY = 30
      RN = N
      DIVN = 1.0E+0 / RN
      X = .1E+0
      ACC = 0.0E+0
C         INITIALISE A AND B
      Y = 1.0E+0
      DO 1 I= 1, 30
      A(I) = I
      B(I) = - Y
      Y = - Y
1     CONTINUE
C     ONE PASS OF THIS LOOP CORRESPONDS TO 300 GAMM UNITS
      DO 15 REP = 1, N
C        FIRST ADDITION/SUBTRACTION LOOP
         I = 30
         DO 2 J = 1, THIRTY
            C(I) = A(I) + B(I)
         I = I - 1
2        CONTINUE
C        FIRST POLYNOMIAL LOOP
         Y = 0.0E+0
         DO 3 I = 1, TEN
            Y = (Y + C(I)) * X
3        CONTINUE
         ACC1 =  Y * DIVN
C        FIRST MAXIMUM ELEMENT LOOP
         Y = C(11)
         DO 4 I = 12, 20
            IF (C(I) .GT. Y)  Y = C(I)
4        CONTINUE
C        FIRST SQUARE ROOT LOOP
         ROOT = 1.0E+0
         DO 5 I = 1, 5
            ROOT = 0.5E+0 * (ROOT + Y/ROOT)
5        CONTINUE
         ACC1 = ACC1 + ROOT * DIVN
C        SECOND ADDITION/SUBTRACTION LOOP
         DO 6 I = 1, THIRTY
            A(I) = C(I) - B(I)
6        CONTINUE
C        SECOND POLYNOMIAL LOOP
         Y = 0.0E+0
         DO 7 I = 1, TEN
            Y = (Y + A(I)) * X
7        CONTINUE
C        SECOND SQUARE ROOT LOOP
         ROOT = 1.0E+0
         DO 8 I = 1, FIVE
            ROOT = 0.5E+0 * (ROOT + Y/ROOT)
8        CONTINUE
         ACC1 = ACC1 + ROOT * DIVN
C        FIRST MULTIPLICATION LOOP
         DO 9 I = 1, THIRTY
            C(I) = C(I) * B(I)
9        CONTINUE
C        SECOND MAXIMUM ELEMENT LOOP
         Y = C(20)
         DO 10 I = 21, THIRTY
            IF (C(I) .GT. Y)  Y = C(I)
10       CONTINUE
C        THIRD SQUARE ROOT LOOP
         ROOT = 1.0E+0
         DO 11 I = 1, 5
            ROOT = 0.5E+0 * (ROOT + Y/ROOT)
11       CONTINUE
         ACC1 = ACC1 + ROOT * DIVN
C        THIRD POLYNOMIAL LOOP
         Y = 0.0E+0
          DO 12 I = 1, TEN
            Y = (Y + C(I)) * X
12       CONTINUE
         ACC1 = ACC1 + Y * DIVN
C        THIRD MAXIMUM ELEMENT LOOP
```

Downloaded from https://academic.oup.com/comjnl/article/22/4/317/343277 by guest on 19 April 2024

```
      Y = C(1)
      DO 13 I = 2, TEN
         IF (C(I) .GT. Y)  Y = C(I)
13    CONTINUE
C     FOURTH SQUARE ROOT LOOP
      ROOT = 1.0E+0
      DO 14 I = 1, FIVE
         ROOT = 0.5E+0 * (ROOT + Y/ROOT)
14    CONTINUE
      ACC1 = ACC1 + ROOT * DIVN
      ACC = ACC + ACC1
15    CONTINUE
      WRITE( 6, 100) N, ACC, ACC1
C SHOULD PRINT N THEN 16.73343 22410 90064 71684 80142
C                     13037 73134 63994
C           AND THEN 16.73 ... / N
100   FORMAT( I10, 2E30.22 )
C FORMAT SHOULD BE ADJUSTED TO PRINT TO MAXIMUM PRECISION
STOP
END
```


References

GENTLEMAN, W. M. and WICHMANN, B. A. (1973). Timing on Computers, *SIGARCH*, (ACM) Vol. 2, pp. 20-23.

HEINHOLD, J. and BAUER, F. L. (Eds.) (1962). *Fachbegriffe der Programmierungstechnik*. Ansgearbeitet vom Fachausschutz Programmieren der Gesellschaften fuer Angewandte Mathematik und Mechanik (GAMM) Muechen, Oldenbourg-Verlag.

OSTERWEIL, L. J. and FOSDICK, L. D. (1976). DAVE—A validation error detection and documentation system for FORTRAN programs, *Software—Practice and Experience*, Vol. 6, pp. 473-486.

SCOWEN, R. S. (1969). BABEL, A new programming language, *National Physical Laboratory Report CCU7*.

WICHMANN, B. A. (1973). *ALGOL 60 Compilation and Assessment*, Academic Press, London.

Downloaded from https://academic.oup.com/comjnl/article/22/4/317/343277 by guest on 19 April 2024

CAD meets AI

Artificial Intelligence and Pattern Recognition in Computer Aided Design, edited by J-C. Latombe, 1978; 510 pages. (*North-Holland*, $60.00)

This book presents the 20 papers and as many discussions of a conference convened in March 1978 by the IFIP working group on Computer Aided Design (CAD). Its purpose was 'to study the impact of Artificial Intelligence (AI) and Pattern Recognition (PR) on CAD'.

Of the three invited papers on CAD, AI and PR, Sandewall's account of AI stands out. It brought home the central message 'the current body of knowledge in AI is mostly a way of thinking' and stressed his own view of AI's deep almost philosophical commitment to advanced programming technology. Warman's paper on CAD failed to give me the picture I needed of CAD practice and problems, though Fenves' paper left me in no doubt about the sheer size of design specifications that are created in building designs and other fields of structural engineering. Nagao's workmanlike review of pattern recognition failed to arouse the audience and PR was clearly a non-issue throughout the conference.

One of the issues that PR might be thought to address was taken up in a number of papers directed towards the man-machine interface. Thus the paper by Mohr and Masini presents the syntax-directed approach to recognising drawings; in contrast Liardets excellent paper shows how knowing the semantics of the pictorial forms being drawn can be mobilised to 'tidy up' and recognise the input. Other presentations concerned with making sense of drawings on paper were consistently attacked by Negroponte (who stands out in many of the discussions) for being the wrong way to go about it: graphical displays will be 'flat portable transparent waterproof' but not like paper capable of being crumpled?

While there can be little doubt that the communication interface can and should require intelligent computation the main emphasis of the conference concerned problem solving. Several papers were directly concerned with the presentation of systems employing problem solving techniques for necessarily limited areas of design (Pereira, Henrion, Perkowski, Tyugu, McDermott), but the significance of the topic lay not so much in particular solutions as in its power to serve as a focus for questions about the nature of designing. Akin examined the behaviour of designers with just such a perspective based upon Newell and Simon's conceptions of human information processing. He believes that many cognitively distinct mechanisms contribute to the emerging solution, an account echoed in another Carnegie Mellon contribution from Eastman who sought to integrate the contribution of different components of a CAD system in a single data base reminiscent of the organisation of the speech-understanding systems built by Reddy and his colleagues at Carnegie Mellon University. Akin notes that 'it is widely accepted that designers use parts of buildings' (as a basis for generating solutions) and it is this notion that is at the centre of what was perhaps the most important contribution to the conference: Sussman's idea of 'problem solving by debugging almost right plans' (PSBDARP). These almost-right-plans or 'answers' provide the basis for organising the design work, specifically in debugging and patching the bits of the answer that don't fit the problem specification. While Sussman's presentation concerned the implementation of his PSBDARP approach in the field of circuit design, it has a wider significance. It represents in part a concept widespread in AI that as intelligent beings we bring to every situation large hunks of organised experience that determine our perceptual expectations, subserve our understanding and guide our thinking. Variously known as schemata, scripts or frames they figure prominently in AI work on vision and natural language. Sussman's distinctive contribution to AI is to introduce his own version the 'plan' into problem solving. Sussman's argument was not just 'get some answers into your systems' but also, 'look at what you are doing as a designer in these terms'. Leaving the 'answers' with the designer of course maintains the balance of design responsibility as CAD has largely conceived it—with the user—but Sussman (supported by Sandewall's account of LISP style programming in AI) sees the prospect of the designer more explicitly formulating his almost-right-answers in computational terms by giving him powerful programming tools rather than problem oriented packages. Programmers are for Sussman designers, and AI programmers write programs to explore the problem often discarding the program to write a better more informed and informative version. The programming technology that supports such an approach to design is perhaps one of the things AI has to offer CAD, and if designing in the new computer age is to become synonymous with debugging then we'll surely need it.

M. CLOWES (Brighton)

An Introduction to Programming and Applications with Fortran by T. E. Hull and D. D. F. Day, 1978; 254 pages. (*Addison-Wesley*, £8·25)

This is really two separate and indifferent volumes stuck together to make one poor but expensive book offering terrible value for money.

The first half is a vague and feeble introduction to FORTRAN (it does not even touch upon COMMON). The second half, which has no need of the first, has its brief moments of value but attempts too much with the result that it is little more than a very superficial survey of various numerical aspects of computation.

D. L. FISHER (Leicester)