INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS, VOL. 7, 1077–1092 (1987)

AN EXPERIMENT WITH ARITHMETIC PRECISION IN LINEAR ALGEBRA COMPUTATIONS*

I. DUFF[†]

Harwell Laboratory, Oxfordshire, U.K.

J. LAMINIE

Université de Paris-Sud, 91405 Orsay, France

A. LICHNEWSKY

Université de Paris-Sud, 91405 Orsay, France and I.N.R.I.A., 78153 Le Chesnay, France

AND

F. THOMASSET

INRIA, BP153 Rocquencourt, 78153 Le Chesnay, France

SUMMARY

Several techniques for experimental determination of floating point precision in practical computations are examined, and applied to linear algebra algorithms. These techniques are simple enough to be directly applicable to existing production codes, requiring a very limited amount of software on many machines, and yet they yield interesting information on the numerical precision of a computation.

Our choice of linear algebra algorithms includes a direct solver (namely the MA32 program from the Harwell Library) and several variants of preconditioned conjugate gradients (the methods DIAG, INV, MINV and POL of Reference 1). The results may be of interest as method selection criteria, and thus complement Mflop performance data available from several sources.

KEY WORDS Floating Point Arithmetic Rounding Errors Accuracy Linear Algebra

1. INTRODUCTION

The evolution of computers large in both speed and memory size now makes it practical to perform numerical computations that exceed 10^{12} floating point operations. In standard practice, programs are being 'stretched' to cover finer and finer discretized models, with little consideration given to their limitations in terms of arithmetic precision.

However, the basic floating point precision of the machines has not varied much in recent years, since it is generally felt that the speed issue is still the most important. Most of the effort has been towards a standardization of floating point formats and the construction of high precision arithmetic, which is not generally available and incurs high performance penalties.[2] Our present view is that the systematic use of some easy-to-use precision evaluation tools can help at both

Based on an invited lecture.

* Computations performed on the CCVR equipment

[†] I. Duff was at INRIA on leave from Harwell.

algorithm design and coding stage 5, and when selecting an algorithm for a particular application. This implies being able to apply the method on production codes with little, if any, programming cost and to analyse the results.

The introduction of vector computers with large memories, especially those with 32-bit floating point capabilities, brings us closer to the limit where precision consciousness may become a major issue. Moreover, this class of machine naturally comes with powerful restructuring and optimizing compilers which perform some transformations that can potentially affect the floating point precision of a result.[3,4] Here too we feel that a properly instrumented approach can help to cope with the issue.

2. A MODEL OF FLOATING POINT ARITHMETIC

2.1. Basic notions

The most difficult aspect is to have a model that can be precise enough to analyse the phenomena, yet abstract enough to permit meaningful study of whole algorithms. This question is not totally identical to the one of making floating point representation characteristics available to high-level language codes. In the latter case, one would be seeking to adapt at run time the algorithms to exploit the full precision of the computer. In the former case we want to assess how successful the algorithm is, and how hard the problem is.

The aim of normalized floating point representation is to bound the admissible relative error on arithmetic computation, under the constraint that the operands and results are neither in the overflow range, nor in the underflow range. If this is true, one has*

$$\begin{gathered}\forall u \in \mathcal{FL}, \quad \forall v \in \mathcal{FL}, \\ u \ \boxed{\text{op}} \ v = (u \text{ op } v) \times (1+\delta), \\ |\delta| \leqslant U_{\text{err}},\end{gathered} \tag{1}$$

where U_{err} is a machine-dependent constant, the *maximum relative error*, whose relation with the number representation is discussed in detail in References 5–7.

The main problem in analysing an algorithm's behaviour is that this relation introduces an auxiliary variable δ_{opnum} per operation in the computation. Setting *nops* to the number of actual floating point operations, the actual result is thus a mathematical function of the data D and the set of individual relative errors $(\delta_1, \dots, \delta_{nops})$:†

$$\begin{aligned}\boxed{\text{R}} &= \boxed{\text{F}}\,(D) \\ &= \mathcal{F}_\delta(\text{val}_{\mathcal{R}}(D),\ (\delta_1, \dots, \delta_{nops})),\end{aligned} \tag{2}$$

under the constraint

$$|\delta_i| \leqslant U_{\text{err}}.$$

(In the above formulae we have made use of the notation: program's floating point data, D; data value in the field $\mathcal{R}$, $\text{val}_{\mathcal{R}}(D)$; floating point result expressed in $\mathcal{R}$, $\mathcal{F}_\delta$.)

* Some machines may exhibit deviations from this model, e.g. the Cray X-MP

† Since we have not excluded here tests depending on floating point quantities, this amounts to saying that the program's output is a function of its inputs D, which may be highly non-differentiable.

On the other hand, the exact result is expressed as follows:

$$\begin{aligned}\mathcal{R} &= \mathcal{F}(D) \\ &= \mathcal{F}_\delta(\mathrm{val}_{\mathcal{R}}(D),\ (0,\ldots,0)).\end{aligned} \tag{3}$$

2.2. *On the influence of vectorization techniques*

The vectorization techniques are aimed at exploiting the existing potential parallelism in programs so as to obtain significant speed-ups on pipelined machines and more specifically vector computers. They can be classified into two very broad categories: *program flow* and *algebraic* related transformations.

Program flow. These techniques are aimed at restructuring the program, but do not require any knowledge of the properties of the basic operations, besides the fact that each operator $\boxed{\text{op}}$ defines a pure function from its operands to its results: res ← $\boxed{\text{op}}$ (o_1, o_2). This implies that the exact initial sequence of 'atomic' operations and results are produced on the binary representation of each partial result, thereby ensuring identical numerical properties.

These transformations are applied either source to source, in an appropriate high level language, or in the process of translating from such a source language to machine code. In both cases, we emphasize that the above statement is exact if the final implementation of floating point operation is identical when the machine code is generated. In particular, it must be true that scalar and vector floating point operations are exactly identical, that no *improvements*, such as keeping extra mantissa bits for register quantities, floating point operator strength reduction etc., are used.

Among these transformations, the most notable are loop splitting, blocking, reordering, alignment and distribution; replacement of IF by masks, or *scatter/gather* based constructs. A detailed description of these transformations can be found in References 4, 8 and 9.

When dealing with conditionals, some of these techniques can nevertheless produce additional intermediate floating point results—possibly invalid—which will not participate in the final results, but which must not produce interrupts when computed. Since the final result will not be affected, we do not have to deal with them here, other than to mention that the underflow and overflow diagnostics that may ensue should be ignored,* unless they really abort the computation. In such a case, we would be much better off if the hardware returned NaNs, as defined in the IEEE standard[10] and is capable of handling operations between NaNs. If restructuring transformations are well designed, the final results would only involve well formed legal floating point numbers, i.e. non-NaN.

Algebraic. The techniques involved here make use of the algebraic properties of the arithmetic operators. The simplest use only the commutativity and associativity of addition and multiplication in the reals. The more sophisticated make use of the *field* properties of the reals. Of course, such properties do not hold in floating point arithmetic, leading to modifications of the arithmetic behaviour of programs. Among these operations, we find

(a) tree-height reduction of general expressions
(b) reduction parallelizing by tree-height reduction for $\sum_{i=1}^{n} x_i$ and $\prod_{i=1}^{n} x_i$

* The adequate hardware and software features should really permit inhibition of these interrupts.

† NaNs are symbolic indicators encoded in the floating point format, meaning that the floating point item represents an invalid result or unavailable data.

(c) recurrence solving, for instance by odd-even reduction/elimination
(d) floating point operator strength reduction.

Details can be found in References 11–14.

3. DETERMINING THE PRECISION OF A COMPUTATION

Starting from formula (1), there are several measures of error we are interested in:

1. worst case error, which corresponds to:

$$\max_{|\delta_i| \leqslant U_{\text{err}}} \left| \mathscr{F}_\delta(\text{val}_{\mathscr{R}}(D),\ \ (\delta_1,\ldots,\delta_{nops})) = \mathscr{F}_\delta(\text{val}_{\mathscr{R}}(D),\ \ (0,\ldots,0)) \right| \tag{4}$$

2. statistical error estimate
3. sensitivities of errors on individual operations.

Worst case estimates have been widely used in connection with linear algebraic algorithms, and a systematic analytic study of numerical precision by Wilkinson and others has been most beneficial.[15] However, they tend to overestimate errors, and their application requires a thorough mathematical analysis, especially if realistic estimates are sought (cf. section 5.2).

As a reference in making these estimates, we can use the condition of the problem, which is generally taken to be the sensitivity of the result to errors in the problem data:

$$\max_{|\rho_i| \leqslant U_{\text{err}}} \left| \mathscr{F}_\delta(\text{val}_{\mathscr{R}}(\ldots,(D_k+\rho_k)),\ \ (0,\ldots,0)) - \mathscr{F}_\delta(\text{val}_{\mathscr{R}}(D),\ \ (0,\ldots,0)) \right|. \tag{5}$$

A sound requirement for a stable algorithm would be that the numerical error be no greater than the problem's condition.

3.1. *Perturbation techniques*

In order both to avoid the analytic difficulties, and to seek a statistically relevent error estimate, these methods consider that the δ_i are independent random variables, distributed according to a known distribution with zero mean, and *postulate* that the results will also share a known distribution. From the result distribution's variance the error estimate is then derived. These methods have been advocated and experimented or by Laporte and Vignes.[16]

To be more specific, the probability measure Ω is constructed by assigning equal probabilities to all computations in a sequence. The $\delta_i(\omega)$ are supposed to satisfy

$$\mu(\delta_i) = \int \delta_i(\omega)\,\mathrm{d}\Omega = 0, \tag{6}$$

$$\text{var}(\delta_i) = \int [\delta_i(\omega)]^2\,\mathrm{d}\Omega, \tag{7}$$

$$\text{dev}(\delta_i) = \sqrt{\int [\delta_i(\omega)]^2\,\mathrm{d}\Omega} \tag{8}$$

$$= \xi U_{\text{err}}. \tag{9}$$

Here ξ is a proportionality constant determined from the operation error distribution. This model leads to the following formulae:

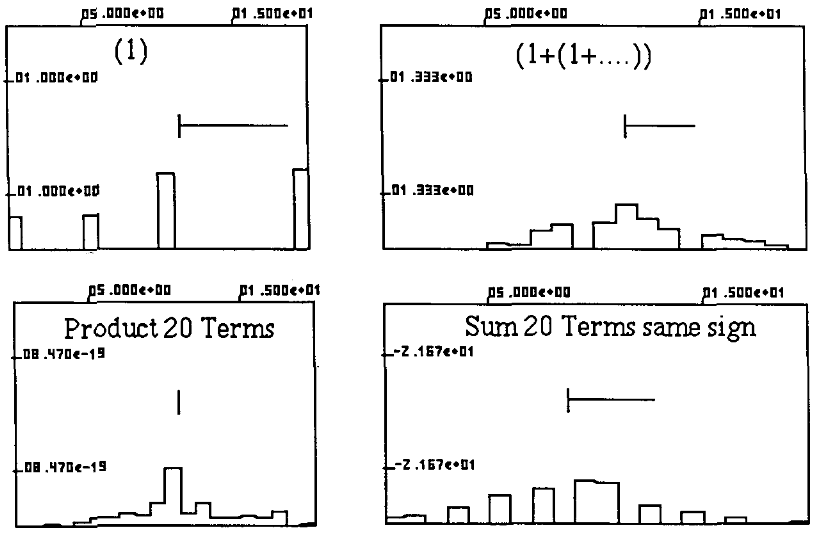

Figure 1. Error distribution for selected evaluations

$$E[R(\Omega)] = \int \mathscr{F}_\delta(\mathrm{val}_{\mathscr{R}}(D), \quad (\delta_1(\omega), \ldots, \delta_{nops}(\omega)))\, \mathrm{d}\Omega, \tag{10}$$

$$\mathrm{var}\,[R(\Omega)] = \int (\mathscr{F}_\delta(\mathrm{val}_{\mathscr{R}}(D), \quad (\delta_1(\omega), \ldots, \delta_{nops}(\omega))) - E[R(\Omega)])^2\, \mathrm{d}\Omega, \tag{11}$$

$$\mathrm{dev}\,[R(\Omega)] = \sqrt{\int (\mathscr{F}_\delta(\mathrm{val}_{\mathscr{R}}(D), \quad (\delta_1(\omega), \ldots, \delta_{nops}(\omega))) - E[R(\Omega)])^2\, \mathrm{d}\Omega}. \tag{12}$$

and the error is estimated by $\hat{\xi}^{-1}$ dev $[R(\Omega)]$. Here $\hat{\xi}$ is determined from the result distribution.* There remain two difficulties with this approach, the first one being that in any real computation, all the δ_i would be determined deterministically, the second being that the hypotheses made on the error distribution are far from satisfactory. On this last issue one may find a precise discussion in Reference 17, as well as some hints on the effect of some arithmetic sequences with uniform error distribution, obtained by random perturbations in Figure 1.

4. APPLICATION TO ITERATIVE COMPUTATIONAL ALGORITHMS

To cope with the first issue, one simply performs a series of computations, explicitly perturbing the arithmetic so as to ensure the hypothesis on the individual errors, but it must be noted that the error distribution on a typical result is very far from known, as can be seen in Figure 1.

4.1. *Sensitivity analysis technique*

To describe very briefly this method, introduced by Miller,[18,19] we start by formally linearizing the expression

*For the VAX experiments shown below, we have imposed $\xi \approx 1000$ and taken $\hat{\xi} = 1$. For the Cray experiments, $\xi \approx 2$ and $\hat{\xi} = 1$.

$$\mathscr{E}(D),(\delta_1,\ldots,\delta_{nops}) = |\mathscr{F}_\delta(\mathrm{val}_\mathscr{R}(D),\ (\delta_1,\ldots,\delta_{nops})) - \mathscr{F}_\delta(\mathrm{val}_\mathscr{R}(D),\ (0,\ldots,0))|, \tag{13}$$

$$\mathscr{E}_\mathscr{S}(D,0) = \frac{\mathrm{D}\mathscr{E}}{\mathrm{D}\delta}(D,(0,\ldots,0)), \tag{14}$$

and then approximating the worst case error,

$$\max_{|\delta_i| \leqslant U_{err}} (\mathscr{E}_\mathscr{S}(D,0)\cdot(\ldots,\delta_i)),$$

by

$$\sigma(D) = \sum_{j=1}^{nops} \left| \frac{\partial \mathscr{E}}{\partial \delta_j}(D,(0,\ldots,0)) \right| U_{\mathrm{err}}.$$

If we suppose that we are handling a fragment of straight-line code, or more generally a program in which tests do not depend in any way on floating point data, we can use the following set of relations to compute such derivatives effectively. To describe these relations, it is convenient to introduce the following notation for the linearized error of an expression A involving the error variables $(\delta_{\alpha(1)},\ldots,\delta_{\alpha(k)})$, making the expression A and the set of error variables apparent:

$$\Phi(A)(\delta_{\alpha(1)},\ldots,\delta_{\alpha(k)}) = \left(\sum_{j=1}^{k} \frac{\partial \mathscr{E}}{\partial \delta_{\alpha(j)}}(D,(0,\ldots,0))\delta_{\alpha(j)} \right) U_{\mathrm{err}}.$$

The linearized error of an expression is then computed from its operands, introducing a new* error variable δ_v,

$$A = B + C \Rightarrow \Phi(A) = (B + C)\delta_v + \Phi(B) + \Phi(C),$$

$$A = B - C \Rightarrow \Phi(A) = (B - C)\delta_v + \Phi(B) - \Phi(C),$$

$$A = B*C \Rightarrow \Phi(A) = (B*C)\delta_v + C*\Phi(B) + B*\Phi(C),$$

$$A = B \div C \Rightarrow \Phi(A) = (B \div C)\delta_v + \frac{C*\Phi(B) - B*\Phi(C)}{C^2}.$$

In the case of tests depending on floating point data, the function $\mathscr{E}$ is in general not differentiable and the whole approach fails.†

4.2. *Tools used in this study*

Two tools are used in this study: the first, FLOP2, runs on a Cray under CFT and is designed to handle whole production grade programs; the other, FLOPV, runs under UNIX and is designed to enable the user to redefine fully the floating point semantics. A more detailed description will be made available in Reference 20; the source codes are available from the authors.‡

Flop 2. This tool intercepts all floating point computations in selected program units by making use of a compiler option of the Cray CFT FORTRAN compiler. This is completely independent of

* i.e. unique for each operation in the program

† It is still possible to use the method, ignoring the tests, to obtain an estimate of rounding errors for a *frozen* sequence of branches. Indeed we shall just do so for pivot selection strategies in the sequel.

‡ By electronic mail: lich @ inria. ARPA,...! mcvax! inria! lich

vectorization options, which means that vectorized and possibly restructured code is intercepted 'as is'. It is of course possible to inhibit the effect of vectorization so as to compare with a less restructured version.

The precision assessment capabilities are as follows:

(a) rounding and truncation of the floating point results at any mantissa length shorter than 48 bits
(b) random perturbation of the floating point results at any mantissa length shorter than 48 bits.

Both of these operations are performed on the full precision floating point result, which is obtained by standard Cray floating point operations.[21] The mantissa length and processing option can be varied at run time. To make full use of the random perturbation technique, the user has to make a series of runs and compute the standard deviations.

FLOPV. In this case we have provided a Pascal environment in which the user can fully redefine the floating point semantics. This is done by the translation of the original Pascal program, so as to use a user-defined type **newreal**, which redefines the standard **real** type. The set of Pascal standard arithmetic functions is supported, as well as the full Pascal syntax. The floating point semantics are then simply defined by a user-written package, containing the realizations of our functions. It must be noted that the implementation of such packages in object-based languages permiting the overloading of operators makes the implementation of such a package most convenient.[22]

In these tests we have made use of two semantics for floating point operations:

(i) random perturbation after 56 mantissa bits
(ii) simplified sensitivity analysis.

To obtain this simplified model, we simply overestimate $\Phi(A)$:

$$\Phi(A)(\delta_{\alpha(1)},\ldots,\delta_{\alpha(k)}) \leqslant \left(\sum_{j=1}^{k} \left| \frac{\partial \mathscr{E}}{\partial \delta_{\alpha(j)}}(D,(0,\ldots,0))\delta_{\alpha(j)} \right| \right) U_{\text{err}}, \tag{15}$$

$$\leqslant \Psi(A), \tag{16}$$

and use the following set of rules to compute the $\Psi(\cdot)$:

$$A = B + C \Rightarrow \Omega(A) = |B + C|\,U_{\text{err}} + \Psi(B) + \Psi(C),$$

$$A = B - C \Rightarrow \Psi(A) = |B - C|\,U_{\text{err}} + \Psi(B) + \Psi(C),$$

$$A = B*C \Rightarrow \Psi(A) = |B*C|\,U_{\text{err}} + C*\Psi(B) + B*\Psi(C),$$

$$A = B \div C \Rightarrow \Psi(A) = |B \div C|\,U_{\text{err}} + \frac{C*\Psi(B) + B*\Psi(C)}{C^2}.$$

5. METHOD VALIDATION ON TEST PROBLEMS

The following set of tests is intended to validate our approach and show its relevance on test problems. Most of the information is obtained by comparing the outcome of the *random perturbation* and the *simplified sensitivity* (model) methods. For each test result given below, we indicate the test software used (FLOP2/FLOPV), the value of U_{err}, and the method (Round/Trunc/R and P/Model). For the model method the results come out in the form of a floating point number whose mantissa shows only valid digits. When none is estimated valid, we show the result in the form $x{\cdot}xxxxe \pm xx(nn)$ which means that the estimated error is 10^{nn} times greater than

Table I. Results of Newton form interpolation

Model	Rand P	(Standard deviation)	Interpolation error
Uniformly distributed interpolation points			
0·622194551463E1	6·22194551463784e + 00	9·9e − 12	− 0·62E1
2·10011074E − 1	2·10011073912721e − 01	2·9e − 12	2·8E − 3
1·873294E − 1	1·87329439769213e − 01	4·3e − 10	− 1·2E − 1
Chebyshev interpolation points			
0·40969E − 1	4·09686104143964e − 02	6·3e − 10	1·6E − 3
2·13181015458E − 1	2·13181015458284e − 01	9·9e − 15	− 3·3E − 4
0·687007101873E − 1	6·87007101872744e − 02	3·4e − 15	− 1·2E − 3

the result. For the Rand P method we show the standard deviation as a measure of the estimated error. Our set of test problems contains:

(i) polynomial interpolation
(ii) Gaussian elimination
(iii) conjugate gradient.

5.1. Polynomial interpolation

Our test interpolates the function $(1 + 25x^2)^{-1}$ on the segment $[-1, +1]$ using Chebyshev or uniformly spaced interpolation points. The Newton form is used in the computation.[23] The results are shown in Table I. Both methods indicate that the floating point error varies both with the choice of interpolation point spacing and with the point where the result is evaluated. However they are coherent as they give the same relative information. For the points where the interpolation (method) error is small, computation error does become an interesting issue.

5.2. Gaussian elimination

We have tested the Gaussian elimination method with the following pivoting strategies:

(a) no pivoting
(b) column pivoting using the maximum element in absolute value
(c) column threshold pivoting, using the same heuristic as in the MA32 frontal code from the Harwell library. Namely, the next row i is selected if

$$|a(i, i) \geqslant \alpha \max_{j \geqslant i} |a(j, i)|;$$

the parameter $\alpha (0 < \alpha \leqslant 1)$ is used to set the threshold
(d) full pivoting using the maximum element in absolute value.

Our test problems are as follows:

1. Hilbert matrix of order n.
2. Van der Monde matrix of order n with $\alpha_i = (n + i + 1/n + 1)$.
3. Matrix M-A:[15]

$$\begin{pmatrix} 1 & 0 & \dots & 0 & 0 \\ -1 & 1 & \dots & 0 & 0 \\ -1 & -1 & 1\dots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ -1 & -1 & \dots & 1 & 0 \\ -1 & -1 & \dots & -1 & 1 \end{pmatrix}.$$

4. Matrix M-B:[24]

$$\begin{pmatrix} 1 & -1 & -2k & 0 \\ 0 & 1 & k & -k \\ 0 & 1 & k+1 & -(k+1) \\ 0 & 0 & 0 & k \end{pmatrix}.$$

5. Matrix M-C:[25]

$$\begin{pmatrix} 10 & -7 & 0 \\ -3 & 2 & 6 \\ 5 & -1 & 5 \end{pmatrix}.$$

6. Matrix M-D:

$$\mathrm{MD}_{i,i} = n,$$

$$\mathrm{MD}_{i,j} = \frac{i-j}{i+j+1}, \quad \text{for } i \neq j.$$

The results of our models can be compared with the estimates from Wilkinson's backward analysis technique.[15,26,27] Solving the system $\mathbf{Ax} = \mathbf{b}$ by the Gauss method, the computed solution $\hat{\mathbf{x}}$ satisfies exactly $(\mathbf{A} + \mathbf{E})\hat{\mathbf{x}} = \mathbf{b}$, where the perturbation matrix $\mathbf{E}$ is bounded by

$$\|\mathbf{E}\|_\infty \leqslant 8n^3\rho \|\mathbf{A}\|_\infty U_{\mathrm{err}},$$

with

$$\rho = \frac{\max\limits_{i,j,k} |\hat{a}_{ij}^{(k)}|}{\|\mathbf{A}\|_\infty},$$

where $\hat{a}_{ij}^{(k)}$ is the element in row i, column j at step k of elimination. If we set

$$r = \|\mathbf{E}\|_\infty \|\mathbf{A}^{-1}\|_\infty,$$

then, provided $r < 1$ and a total pivoting strategy is used,

$$\frac{\|\mathbf{x} - \hat{\mathbf{x}}\|_\infty}{\|\mathbf{x}\|_\infty} \leqslant \frac{r}{1-r}.$$

We give below the computed values of r, taking for U_{err} the value corresponding to full precision on DEC/VAX namely $2^{-57} = 6{\cdot}7 \times 10^{-18}$. We also use the classical notation for the condition number of $\mathbf{A}$: $\kappa_\infty(\mathbf{A}) = \|\mathbf{A}\|_\infty \|\mathbf{A}^{-1}\|_\infty$.

Test with matrix M-A. This matrix of dimension n has l_1 and l_∞ condition number $n2^{n-1}$. For $n = 20$, both model and random perturbation techniques indicate a degradation in the

Table II. Results of Gaussian elimination for matrix M-A, $n = 20$

	Rand P	No interchange for pivoting (Standard deviation)	Model	Full precision
$x[1]$	1·00000000000004	(1·99e − 14)	1·000000000000000	1·000000000000000
$x[10]$	1·00000000000209	(5·87e − 12)	1·000000000000	1·000000000000000
$x[20]$	2·00000000213507	(5·96e − 09)	2·000000000	2·000000000000000

Table III. Results of Gaussian elimination for matrix M-B

Rand P	No interchange for pivoting (Standard deviation)	Model	Full precision
5·760003025177282e − 01	(2·14)	1·0e + 00(0)	0·997656464576721
1·42400000379421e + 00	(2·14)	1·0e + 00(0)	1·00234353542328
9·99999957600011e − 01	(2·14e − 07)	1·0000000E0	0·999999999765646
2·00000020000001e + 00	(3·87e − 14)	2·000000200000000E0	2·00000020000000

Table VI. Results of Gaussian elimination with matrix M-C

Model	Full precision	Rand P	(Standard deviation)
No interchange for pivoting			
− 3·9e − 16(2)	− 3·885780586188805e − 16	− 1·00073282993663e − 13	(4·61e − 13)
− 1·000000000000	− 1·00000000000000	− 1·00000000000014	(9·65e − 15)
1·00000000000000	1·00000000000000	9·99999999999996e − 01	(9·65e − 15)
Partial pivoting			
0	0·00000000000000	1·00286445814391e − 14	(1·59e − 14)
− 1·00000000000000	− 1·00000000000000	− 9·99999999999996e − 01	(3·41e − 14)
1·000000000000000	1·00000000000000	9·99999999999999e − 01	(1·02e − 14)
Total pivoting			
− 4·4e − 17(1)	− 4·44089209850063e − 17	− 1·72251102270591e − 14	(2·79e − 14)
− 1·00000000000000	− 1·00000000000000	− 1·00000000000003	(3·98e − 14)
1·000000000000000	1·00000000000000	1·00000000000000	(1·20e − 14)

result's precision, as is shown* in Table II, whereas the exact solution is $x_1 = x_{10} = 1{\cdot}0$; $x_{20} = 2{\cdot}0$. The *a priori* estimate is $r = 8{\cdot}9 \times 10^{-12}$. In this case, both methods are coherent; the full precision result shows that they are both exaggerating the error.

Test with matrix M-B. This set of matrices is designed to have a large l_1 condition number, $8k^2 + 6k + 1$, that goes undetected by the Linpack estimator embodied by the SGECO routine. Our test shows that the modelling method shows the lack of estimated precision before the

* We will show here only a few of the result's component together with the precision estimates, more extensive results will appear in Reference 20.

Table V. Results of Gaussian elimination with matrix M-D, $n = 20$

Model		No interchange for pivoting Full precision	Rand P	(Standard deviation)
$x[1]$	1·00000000000000	1·00000000000000	1·00000000000002	(1·49e − 14)
$x[10]$	1·00000000000000	1·00000000000000	9·99999999999996e − 01	(2·77e − 14)
$x[20]$	2·00000000000000	2·00000000000000	2·00000000000002	(3·86e − 14)

Table VI. Results of Gaussian elimination with the Hilbert matrix, $n = 8$

Model		Full precision	Rand P	(Standard deviation)
No interchange for pivoting				
$x[1]$	1·0e + 00(5)	1·00000000005318	9·99999978821015e − 01	(6·16e − 08)
$x[4]$	1·0e + 00(4)	9·99999929467760e − 01	1·00002682549518	(8·30e − 05)
$x[5]$	2·0e + 00(3)	2·00000016657510	1·99993715219381	(1·96e − 04)
$x[8]$	2·0e + 00(1)	1·99999996258034	2·00001390746351	(4·42e − 05)
Column pivoting				
$x[1]$	1·0e + 00(5)	9·99999999803623e − 01	1·00000001023833	(1·4e − 07)
$x[4]$	1·0e + 00(5)	1·00000026078078	9·99989057077121e − 01	(2·02e − 04)
$x[5]$	2·0e + 00(4)	1·99999938555962	2·00002453764322	(4·79e − 04)
$x[8]$	2·0e + 00(2)	2·00000013700937	1·99999510679122	(1·08e − 04)
Total pivoting				
$x[1]$	1·0e + 00(4)	9·99999999895900e − 01	9·99999989005349e − 01	(6·79e − 08)
$x[4]$	1·0e + 00(4)	1·00000014331820	1·00001339724146	(8·73e − 05)
$x[5]$	2·0e + 00(3)	1·99999966032516	1·99996906549938	(2·04e − 04)
$x[8]$	2·0e + 00(3)	2·00000007659342	2·00000658881595	(4·51e − 05)

failure which occurs for $k = 1{\cdot}0E8$. For $k = 1{\cdot}0E7$ we obtain the results of Table III which is to be compared with the exact solution (1, 1, 1, 2·0000002). Note that here $r = 0{\cdot}14$, and $r/1 - r, = 0{\cdot}16$.

Test with matrix M-C. This matrix has been constructed to demonstrate that pivoting can be necessary to ensure precision. Our test with the model method confirms this fact (Table IV), at least for the partial column pivoting (exact solution $(0, -1, 1)$, $r = 4 \times 10^{-12}$). In this case both the modelling approach and the full precision show that total pivoting is certainly not justified, as it appears less precise.

Test with matrix M-D. The tests with this diagonally dominant well conditioned matrix show that the modelling method gives very good worst case estimates that are not systematically exaggerated. In comparison with the other tests, they are thus interesting to appreciate the diagnostic capabilities of such a method (see Table V). The random perturbation technique performs satisfactorily too. Note that pivoting is not relevant in this case and that $r = 2{\cdot}8 \times 10^{-13}$, $\kappa_\infty(A) = 2{\cdot}6$.

Test with the Hilbert matrix. The Hilbert matrix is well known for its bad condition number. This fact is confirmed by the model method, as well as the quickly growing lack of precision on the

Table VII. Results of Gaussian elimination with the Van der Monde matrix, $n = 10$

Model		Full precision	Rand P	(Standard deviation)
No interchange for pivoting				
$x[1]$	1·0e + 00(11)	9·99999900346606e − 01	1·00000349807613	(4·01e − 05)
$x[4]$	1·0e + 00(9)	1·00000272587411	9·99906683532958e − 01	(1·11e − 03)
$x[5]$	2·0e + 00(8)	1·99999721092296	2·00009461643463	(1·14e − 03)
$x[6]$	1·9e − 06(13)	1·89448236242643e − 06	− 6·36642631962269e − 05	(7·73e − 04)
$x[8]$	2·0e + 00(5)	2·00000024663102	1·99999187598520	(1·01e − 04)
Column pivoting				
$x[1]$	1·0e + 00(8)	1·00000001357745	1·00000860118802	(2·0 − 04)
$x[4]$	1·0e + 00(7)	9·99999740385768e − 01	9·99751452865155e − 01	(5·47e − 03)
$x[5]$	2·0e + 00(5)	2·00000022695029	2·00025904077647	(5·61e − 03)
$x[6]$	− 1·3e − 07(12)	− 1·27844755655186e − 07	− 1·79219475005296e − 04	(3·81e − 03)
$x[8]$	2·0e + 00(2)	1·99999999017490	1·99997580451875	(4·97e − 04)
Total pivoting				
$x[1]$	1·0e + 00(8)	9·99999884076119e − 01	9·99963509745179e − 01	(6·82e − 05)
$x[4]$	1·0e + 00(7)	1·00000317866839	1·00101702733704	(1·89e − 03)
$x[5]$	2·0e + 00(5)	1·99999674480238	1·99895280591326	(1·94 − 03)
$x[6]$	− 1·3e − 07(12)	2·21309017549998e − 06	7·15804532045789e − 04	(1·32e − 03)
$x[8]$	2·0e + 00(2)	2·00000028866135	2·00009435845204	(1·74e − 04)

successive reduced matrices obtained during the forward elimination. However, this estimate shows that the total pivoting variant should behave much better than the variant without row or column interchange. This is not confirmed by the experiment, as evidenced in Table VI, and furthermore some worst case error estimates seem grossly over-estimated by this method. The random perturbation technique gives very precise information in this case and is thus preferable. We give the results for $n = 8(r = 0{\cdot}023, \kappa_\infty(A) = 1{\cdot}17 \times 10^{11}$, exact solution: $x_1, x_4 = 1; x_5, x_8 = 2)$.

Test with the Van der Monde matrix. The modelling method appears very pessimistic for small n but gives a clear warning before a sudden loss of precision which occurs for $n = 12$. The random perturbation method gives strikingly precise results here too. The results for $n = 10$ appear in Table VII and show that Wilkinson's bound can be over-pessimistic indeed (here $r = 15{\cdot}15$, $\kappa_\infty(A) = 2{\cdot}16 \times 10^{12}$, exact solution: $x_1, x_4 = 1$; $x_5 = 2$; $x_6 = 0$; $x_8 = 2$).

5.3. *Conjugate gradient*

In this case our test matrix of order n is

$$\begin{pmatrix} 2 & -1 & 0 & \dots & 0 \\ -1 & 2 & -1 & \dots & 0 \\ & \ddots & \ddots & \ddots & \\ & & & -1 & 2 \end{pmatrix}.$$

Our series of test using the modelling method shows that this method tends to overestimate arithmetic errors too much to be of diagnostic value. The apparent reason seems to be related to the extensive use of scalar products of nearly orthogonal quantities. The random permutation result is

Table VIII. Conjugate gradient: $N = 20$—updating strategy

Model		Full precision	Rand P	(Standard deviation)
x[1]	2·0e + 00(14)	2·00000000000000	2·00000000000030	(2·70e − 13)
x[2]	− 3·5e − 16(30)	− 3·51932025188795e − 16	− 3·90450952819646e − 13	(5·90e − 13)
x[3]	2·0e + 00(14)	2·00000000000000	2·00000000000093	(8·73e − 13)
x[4]	− 6·4e − 16(29)	− 6·44883452194378e − 16	− 7·62398965573755e − 13	(1·13e − 12)
x[5]	2·0e + 00(14)	2·00000000000000	2·00000000000135	(1·34e − 12)
x[6]	− 8·7e − 16(29)	− 8·72023807330091e − 16	− 1·03915940522642e − 12	(1·58e − 12)
x[7]	2·0e + 00(14)	2·00000000000000	2·00000000000172	(1·73e − 12)

Table IX. Conjugate gradient: $N = 20$—recompute strategy

Model		Full precision	Rand P	(Standard deviation)
x[1]	2·0e + 00(21)	2·00000000000000	1·99999999999994	(4·70 − 13)
x[2]	1·1e − 15(37)	1·14491749414469e − 15	3·07244160668848e − 13	(1·38e − 12)
x[3]	2·0e + 00(22)	2·00000000000000	2·00000000000005	(2·49e − 12)
x[4]	1·4e − 15(37)	1·37541887601511e − 15	2·43766150927889e − 13	(2·30e − 12)
x[5]	2·0e + 00(22)	2·00000000000000	2·00000000000050	(2·22e − 12)
x[6]	1·7e − 15(37)	1·67932074496280e − 15	8·14745406560368e − 16	(2·25e − 12)
x[7]	2·0e + 00(22)	2·00000000000000	2·00000000000067	(2·79e − 12)

applicable in this context and gives precise results on our test problems. We give the results for $n = 20$ with two strategies for computing the residual:[26]

1. Iteratively update the residual $\mathbf{r}(k)$ at iteration k with the formula: $\mathbf{r}(k) := \mathbf{r}(k) -$ alpha* $\mathbf{Ap}$ $(k-1)$ where $\mathbf{p}(k-1)$ is the search direction at the previous step.
2. Fully recompute $\mathbf{r}(k)$ for each iteration: $\mathbf{r}(k) := \mathbf{b}(k) - \mathbf{Ax}(k)$.

The results after 20 iterations (Tables VIII and IX) indicate a slight superiority for the first method, which is confirmed by the random permutation method. We plan to further investigate the reason for this behaviour checking in particular for orthogonality conditions with the help of this method.

6. APPLICATION TO DIRECT METHODS

We have applied the round and truncation methods on the multifrontal code MA32 from the Harwell library. The results shown in Figure 2 show the precision obtained in the result, in l_2 and l_∞ norms, as the arithmetic precision is varied. We have not been able to produce any significant anomalies using a set of test problems devised by Duff. We have also tested for the influence of the parameter α controlling the threshold for partial pivoting in this code. The benefit of varying α from 0·01 to 0·1 is clearly illustrated in Table X. Varying from 0·1 to 0·99 appears not to be interesting.

7. APPLICATION TO ITERATIVE METHODS

We have run a series of tests on the algorithms DIAG, INV, MINV and POLY of consus *et al.*[1] Both truncation and rounding methods show that the most precise results are obtained with the

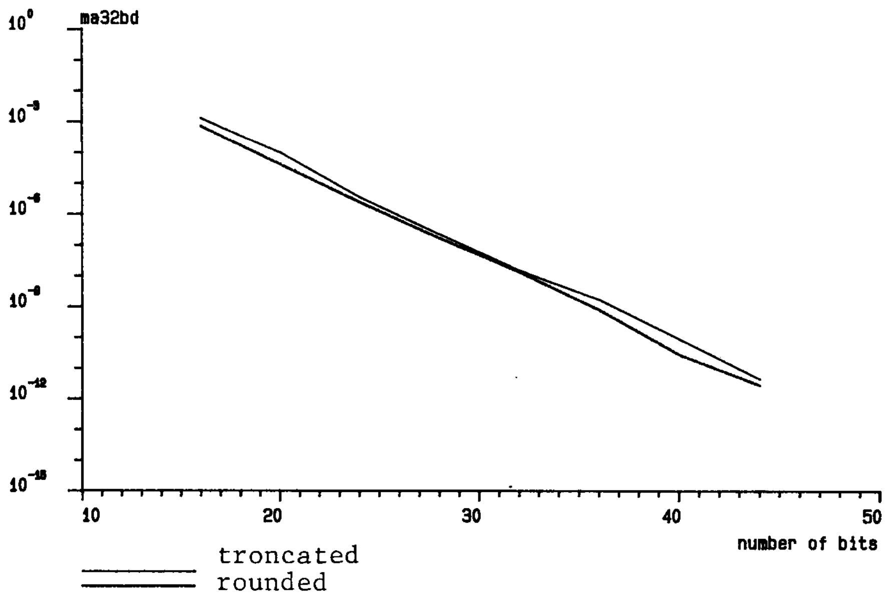

Figure 2. Effect of variable precision on MA32

Table X. Cost of pivoting strategies in MA32: solution of a system of linear equations of order 441, truncated arithmetic

α	CPU time (s)	No. of ops (internal loop)	Relative error (L_2 norm) for several mantissa lengths		
			36 bits	32 bits	28 bits
0·001	0·199	196,312	$1{\cdot}8 \times 10^{-9}$	$3{\cdot}1 \times 10^{-8}$	$5{\cdot}4 \times 10^{-7}$
0·01	0·199	196,312	$1{\cdot}8 \times 10^{-9}$	$3{\cdot}1 \times 10^{-8}$	$5{\cdot}4 \times 10^{-7}$
0·1	0·204	206,080	$4{\cdot}3 \times 10^{-10}$	$6{\cdot}4 \times 10^{-9}$	$9{\cdot}6 \times 10^{-8}$
0·99	0·704	921,599	$5{\cdot}8 \times 10^{-9}$	$1{\cdot}5 \times 10^{-7}$	$5{\cdot}6 \times 10^{-7}$

INV and MINV methods, and that the method POLY gives significantly less precise results. On the other hand, there is little variation in iteration counts as precision is varied, and almost none when the mantissa stays longer than 17 bits (see Figures 3 and 4).

8. CONCLUSION

We have used two methods to ascertain the precision of numerical software in practical situations. The worst case sensitivity technique appears overly pessimistic in many situations but still yield results comparable or better than 'by hand' mathematical analysis. It is also theoretically well founded. The random permutation technique of J. Vignes gives very accurate information in most cases, even when the other methods appear inadequate. Globally we find these methods useful in obtaining correct information on numerical precision, which we have often found very different from our intuition.

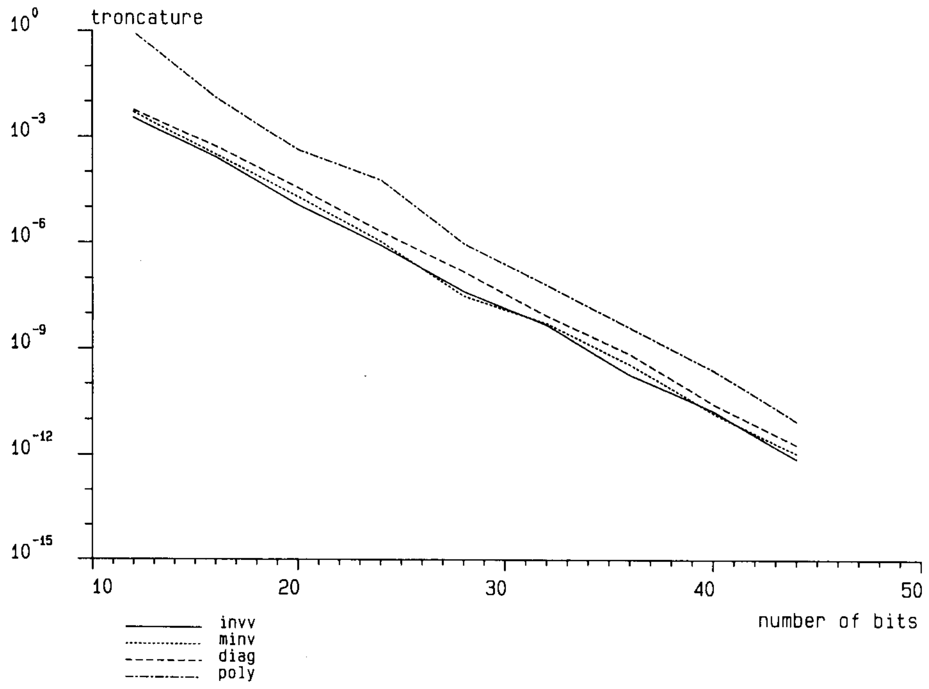

Figure 3. Effect of variable precision on conjugate gradient codes

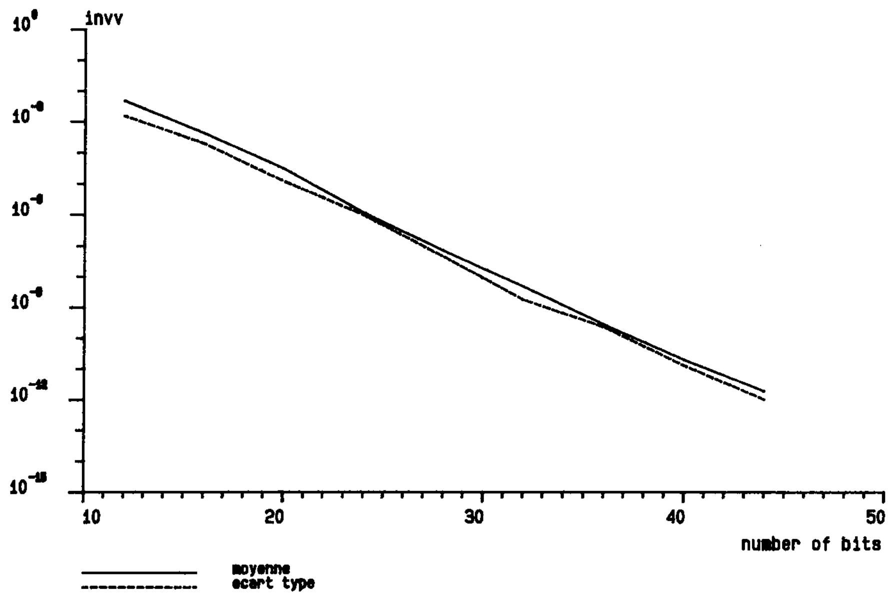

Figure 4. Random perturbation error estimate for conjugate codes

REFERENCES

1. P. Concus, G. H. Golub and G. Meurant, 'Block preconditioning for the conjugate gradient method', *Rep. Lawrence Livermore Berkeley Lab.*, LBL-14856, July 1986.
2. P. Jansen and P. Weidner, 'High-accuracy arithmetic software—some tests of the ACRITH problem-solving routines', *ACM Trans. on Math. Software*, **12** (1), 62–70 (1986).
3. D. J. Kuck and Y. Muraoka, 'Bounds on the parallel evaluation of arithmetic expressions using associativity and commutativity', *Acta Informatica*, **3**, 203–216 (1974).
4. D. J. Kuck, R. H. Kuhn, B. Leasure and M. Wolfe, 'The structure of an advanced vectorizer for pipelined processors', *Fourth Int. Computer Software and Applications Conference*, October 1980.
5. W. S. Brown, 'A simple but realistic model of floating point computation', *ACM Trans. Math. Software*, **7** (4), 445–480 (1981).
6. D. E. Kruth, *The Art of Computer Programming, Vol. 2, Semi-Numerical Algorithms*. Addison Wesley, 1979.
7. K. Hwang, *Computer Arithmetic: Principles, Architecture and Design*, Wiley, 1979.
8. J. R. Allen and K. Kennedy, 'PFC: a program to convert Fortran to parallel form', in K. Hwang (ed.), *Supercomputers: Design and Applications*, IEEE Press, 1985.
9. A. Lichnewsky and F. Thomasset, 'Techniques de base pour l' exploitation du Parallélisme dans les programmes', *Rapp. Rech. INRIA No 460*, 1985.
10. D. Stevenson, 'A proposed standard for binary floating point arithmetic', *IEEE Computer*, (March) 51–87 (1981).
11. R. Brent, D. J. Kuck and K. Maruyama, 'The parallel evaluation of arithmetic expressions without division', *IEEE Trans. Computers*, **C-22** (5), 532–534 (1973).
12. S. Chen and A. Sameh, 'On parallel triangular systems solvers', in T. Freg (ed.), *Proc. Sagamore Domput. Conference*.
13. D. Heller, 'A survey of parallel algorithms in numerical linear algebra', *SIAM Review*, **20**(4), 740–777 (1978).
14. B. Philippe and M. Raphalen, 'Précision numérique dans le cumul d'un grand nombre de termes', *Rapp. Rech. INRIA*, 1985.
15. J. H. Wilkinson, *Rounding Errors in Algebraic Processes*, Prentice Hall, Englewood Cliffs, N.J., 1963.
16. M. Laporte and J. Vignes, 'Etude statistique des erreurs dans l' arithmétique des ordinateurs. Application au contrôle des résultats d' algorithmes numériques', *Numer. Math.*, **23**, 63–72 (1971).
17. W. J. Cody and H. Kuki, 'A statistical study of the accuracy of floating point number systems', *Comm. ACM*, **16** (4), 223–230 (1973).
18. W. Miller, 'Software for roundoff analysis', *ACM Trans. on Math. Software*, **1** (2), 108–128 (1975).
19. E. Ukkonen, 'On the calculation of the effects of roundoff errors', *ACM Trans. Math. Software*, **7** (3), 259–271 (1981).
20. F. Bourdoncle, A. Lichnewsky, F. Thomasset and P. Zimmermann, 'Un logiciel pous l'estimation de la précision numérique', *Rapport Technique INRIA*, to appear.
21. Cray Research Inc., *Cray IS Series Hardware Reference Manual*, HR-808, June 1980.
22. B. Stroustrup, *The C++ Programming Language*, Addison Wesley, 1986.
23. C. De Boor, *A practical Guide to Splines*, Springer-Verlag, 1978.
24. A. K. Cline and R. K. Rew, 'A set of counter-examples to three condition number estimators', *SIAM J. Sci. Stat. Comput.*, **4** (4), 602–611 (1983).
25. George E. Forsythe, Michaël A. Malcoln and Clieve E. Moles, *Computer Methods for Mathematical Computations*, Prentice-Hall, 1977.
26. G. H. Golub and Ch. F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, 1983.
27. J. H. Wilkinson, 'Error analysis of direct methods of matrix inversion', *JACM*, (8), 281–330 (1961).