An experiment with arithmetic precision in linear algebra computations

I.S. Duff, Jacques Laminie, A. Lichnewsky, F. Thomasset

IRIA
CENTRE DE ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tel. (1) 39 63 55 11

Rapports de Recherche

N° 643

AN EXPERIMENT WITH ARITHMETIC PRECISION IN LINEAR ALGEBRA COMPUTATIONS

Iain S. DUFF
Jacques LAMINIE
Alain LICHNEWSKY
François THOMASSET

Février 1987

QUELQUES EXPERIENCES SUR LA PRECISION ARTIHMETIQUE DANS LES METHODES NUMERIQUES EN ALGEBRE LINEAIRE

AN EXPERIMENT WITH ARITHMETIC PRECISION IN LINEAR ALGEBRA COMPUTATIONS*)

Iain S. Duff †
Harwell Laboratory, Oxfordshire, GB

Jacques Laminie
Université de Paris-Sud, 91405 Orsay

Alain Lichnewsky
Université de Paris-Sud, 91405 Orsay
et INRIA, 78153 Le Chesnay

François Thomasset
INRIA, BP. 153, Rocquencourt
78153 Le Chesnay

November 21, 1986

Abstract

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), several variants of preconditioned conjugate gradients. The results may be of interest as method selection criteria and thus complement *MFLOP* performance data available from several sources.

Résumé

Nous examinons plusieurs techniques pour déterminer la précision des calculs en virgule flottante dans les algorithmes numériques, en particulier en algèbre linéaire. Ces techniques, suffisamment simples pour être directement applicables à des codes industriels, exigent un volume limité de logiciel pour leur mise en œuvre sur la plupart des ordinateurs, tout en étant capables de fournir des informations intéressantes sur la précision des calculs.
A titre d'illustrations nous avons choisi : une résolution de système linéaire par méthode directe (plus précisément la routine *MA32* de la librairie Harwell); plusieurs variantes de gradiants conjugués préconditionnés. Les résultats peuvent être utilisés comme critère de sélection de méthode, et complémenter les évaluations de la vitesse en *MFLOPS* obtenues par d'autres moyens.

---

*) Computations performed on the CCVR equipment

†) I. Duff was at INRIA on leave from Harwell

Présenté à

. *Sixième Colloque International* sur la "Simulation d'écoulements par éléments finis"

. *Sixth International Symposium* on "Finite Element Methods in Flow Problems"

Antibes, Juin 1986.

1 Introduction

The evolution of computers large in both speed and memory size now makes practical 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 the 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 [JW86]. Our present view is that the systematic use of some easy to use precision evaluation tools can help both at algorithm design and coding stage 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 analyze 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 [KM][KKLW]. 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 analyze 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:[1]

$$\forall u \in \mathcal{FL} \quad ; \quad \forall v \in \mathcal{FL}$$

[1]Some machines may exhibit deviations from this model. e.g. Cray X-MP

$$u \boxed{\text{op}} v = (u op v) \times (1+\delta) \quad (1)$$
$$|\delta| \leq \mathbf{U_{err}}$$

Where $\mathbf{U_{err}}$ is a machine dependent constant, the maximum relative error, whose relation with the number representation is discussed in detail in [Br], [Kn] and [Hw].
The main problem in analyzing 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 $(\delta_1, ..., \delta_{nops})$ [2]:

$$\boxed{\text{R}} = \boxed{\text{F}}(\text{D}) \quad (2)$$
$$= \mathcal{F}_\delta\,(val_{\mathcal{R}}(\text{D}) \quad , \quad (\delta_1, ..., \delta_{nops}))$$

under the constraint:

$$|\delta_i| \leq \mathbf{U_{err}}$$

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

On the other hand, the exact result is expressed:

$$\mathcal{R} = \mathcal{F}(D) \quad (3)$$
$$= \mathcal{F}_\delta\,(val_{\mathcal{R}}(\text{D}) \quad , \quad (0, ..., 0))$$

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 in two very broad categories: *program flow* and *algebraic* related transformations.

2.2.1 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: $res \leftarrow \boxed{\text{op}}(o_1, o_2)$. This implies that the exact initial sequence of "atomic" operations

[2] 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.

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* like keeping extra mantissa bits for register quantities, floating point operator strength reduction,..., are used.

Among those 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 Kuck [KKLW], Kennedy [AK] and [LT] .

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,[3] unless they really abort the computation. In such a case, we would be much better off if the hardware returned NaN [4], as defined in the IEEE standard [Ste], and is capable of handling operations between NaN. If restructuring transformations are well designed, the final results would only involve well formed legal floating point numbers, ie. non NaN.

2.2.2 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 the floating point arithmetic, leading to modifications of the arithmetic behaviour of programs. Among these operations, we find:

- tree-height reduction of general expressions,
- reduction parallelizing by tree-height reduction for $\sum_{i=1}^{n} x_i$ and $\prod_{i=1}^{n} x_i$,
- recurrence solving, for instance by odd-even reduction / elimination.
- floating point operator strength reduction

[3] The adequate hardware and software features should really permit to inhibit these interrupts

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

Detailed references can be found in Brent & al. [BKM], Chen & Sameh [CS], Heller [Hel], and B.Philippe & M. Raphalen [PR].

3 Determining the Precision of a Computation

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

- worst case error, which corresponds to:

$$\max_{|\delta_i| \leq \mathbf{U_{err}}} \left| \mathcal{F}_\delta \left(val_{\mathcal{R}}(\mathtt{D}) \quad , \quad (\delta_1, ..., \delta_{nops}) \right) - \mathcal{F}_\delta \left(val_{\mathcal{R}}(\mathtt{D}) \quad , \quad (0, \ldots, 0) \right) \right| \tag{4}$$

- statistical error estimate.
- 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.[Wil]. However, they tend to overestimate errors, and their application requires a thorough mathematical analysis, especially if realistic estimates are sought.(Cf. §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 upon errors in the problem data:

$$\max_{|\rho_i| \leq \mathbf{U_{err}}} \left| \mathcal{F}_\delta \left(val_{\mathcal{R}} \left(\ldots, (\mathtt{D}_k + \rho_k) \right) \quad , \quad (0, \ldots, 0) \right) - \mathcal{F}_\delta \left(val_{\mathcal{R}}(\mathtt{D}) \quad , \quad (0, \ldots, 0) \right) \right| \tag{5}$$

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 relevant 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 by M.Laporte & J.Vignes [LV].

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:

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

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

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

$$= \xi \mathbf{U_{err}} \tag{9}$$

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

$$E[R(\Omega)] = \int \mathcal{F}_\delta\,(val_R(\mathtt{D}) \quad , \quad (\delta_1(\omega), ..., \delta_{nops}(\omega)))\, d\Omega \tag{10}$$

$$var(R(\Omega)) = \int (\mathcal{F}_\delta\,(val_R(\mathtt{D}) \quad , \quad (\delta_1(\omega), ..., \delta_{nops}(\omega))) - E[R(\Omega)])^2\, d\Omega \tag{11}$$

$$dev(R(\Omega)) = \sqrt{\int (\mathcal{F}_\delta\,(val_R(\mathtt{D}), (\delta_1(\omega), ..., \delta_{nops}(\omega))) - E[R(\Omega)])^2\, d\Omega}. \tag{12}$$

and the error is estimated by $\hat{\xi}^{-1} dev(R(\Omega))$. Here $\hat{\xi}$ is determined from the result distribution.[5] There remains 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 Cody & Kuki [CK], 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.

[5] 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 $\hat{\xi} = 1$.

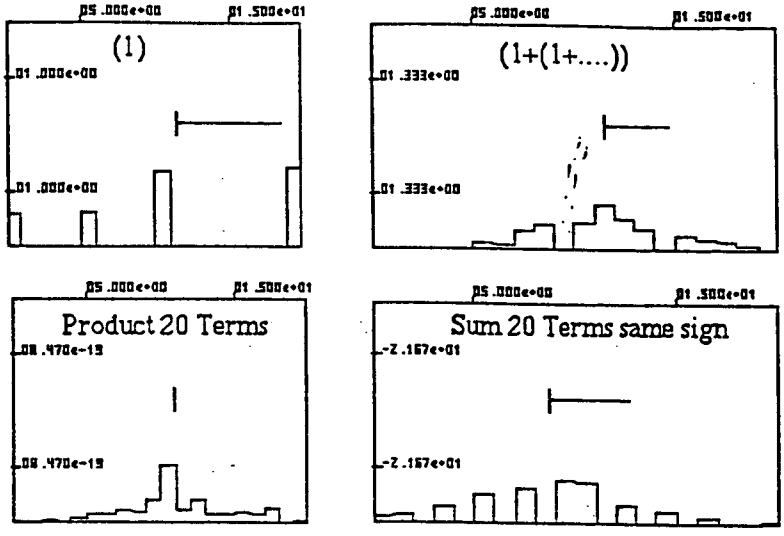

Figure 1: Error distribution for selected evaluations

4.1 Sensitivity Analysis Technique

To describe very briefly this method, introduced by W.Miller, we start by formally linearizing the expression:

$$\mathcal{E}(\mathrm{D},(\delta_1,...,\delta_{nops})) = \left| \mathcal{F}_\delta\,(val_{\mathcal{R}}(\mathrm{D}) \quad , \quad (\delta_1,...,\delta_{nops})) - \mathcal{F}_\delta\,(val_{\mathcal{R}}(\mathrm{D}) \quad , \quad (0,\ldots,0)) \right| \tag{13}$$

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

and then approximating the worst case error

$$\max_{|\delta_i| \leq \mathbf{U_{err}}} (\mathcal{E}_{\mathcal{L}}(\mathrm{D},0).(\ldots,\delta_i,))$$

by:

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

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:

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

The linearized error of an expression is then computed from its operands introducing a new [6] error variable δ_ν,

$$\begin{array}{lcl} A = B + C & \Rightarrow & \Phi(A) = (B+C)\delta_\nu + \Phi(B) + \Phi(C) \\ A = B - C & \Rightarrow & \Phi(A) = (B-C)\delta_\nu + \Phi(B) - \Phi(C) \\ A = B * C & \Rightarrow & \Phi(A) = (B*C)\delta_\nu + C*\Phi(B) + B*\Phi(C) \\ A = B \div C & \Rightarrow & \Phi(A) = (B \div C)\delta_\nu + \frac{C*\Phi(B) - B*\Phi(C)}{C^2} \end{array}$$

In the case of tests depending on floating point data, the function $\mathcal{E}$ is in general not differentiable and the whole approach fails. [7]

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 [BLTZ]; the source codes are available from the authors.[8]

4.2.1 FLOP2

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 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:

- rounding and truncation of the floating point results at any mantissa length shorter than 48 bits
- random perturbation of the floating point results at any mantissa length shorter than 48 bits

Both these operations are performed on the full precision floating point result, which is obtained by standard Cray Floating Point operations (cf. [CrH]). 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.

[6] i.e. unique for each operation in the program

[7] 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.

[8] By electronic mail `lich@inria.ARPA, ...!mcvax!inria!lich`

4.2.2 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 are 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. (Cf. Stroustrup [Str]).

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

- random perturbation after 56 mantissa bits.
- simplified sensitivity analysis.

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

$$\Phi(A)(\delta_{\alpha(1)},\ldots,\delta_{\alpha(k)}) \leq \left(\sum_{j=1}^{k}\left|\frac{\partial\mathcal{E}}{\partial\delta_{\alpha(j)}}(\mathrm{D},(0,\ldots,0)).\delta_{\alpha(j)}\right|\right).\mathbf{U_{err}} \quad (15)$$

$$\leq \Psi(A) \quad (16)$$

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

$$\begin{array}{lllll} A = B+C & \Rightarrow & \Psi(A) & = & |B+C|\,\mathbf{U_{err}} + \Psi(B) + \Psi(C) \\ A = B-C & \Rightarrow & \Psi(A) & = & |B-C|\,\mathbf{U_{err}} + \Psi(B) + \Psi(C) \\ A = B*C & \Rightarrow & \Psi(A) & = & |B*C|\,\mathbf{U_{err}} + C*\Psi(B) + B*\Psi(C) \\ A = B\div C & \Rightarrow & \Psi(A) & = & |B\div C|\,\mathbf{U_{err}} + \frac{C*\Psi(B)+B*\Psi(C)}{C^2} \end{array}$$

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 $\mathbf{U_{err}}$, and the method (Round / Trunc / RandP / 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.xxxxe±xx (nn) which means that the estimated error is 10^{nn} times greater that the result. For the RandP method we show the standard deviation as a measure of the estimated error. Our set of test problems contains:

- Polynomial interpolation

Uniformly distributed interpolation points

Model	RandP	(std.dev.)	Interp.Error
0.622194551463E1	6.221945514637 84e+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

Chebychev interpolation points

Model	RandP	(std.dev.)	Interp.Error
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

Figure 2: Results of Newton form interpolation

- Gaussian elimination
- Conjugate Gradient

5.1 Polynomial interpolation

Our test interpolates the function $(1+25x^2)^{-1}$ on the segment $[-1,+1]$ using Chebychev or uniformly spaced interpolation points. The Newton form is used in the computation. (Cf. DeBoor [DBo]). The results are shown in Figure 2. Both methods indicate that the floating point error vary 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:

- no pivoting.
- column pivoting using the maximum element in absolute value.
- 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

$$|a(i,i)| \geq \alpha * \max_{j \geq i} |a(j,i)|.$$

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

Our test problems are as follows:

- Hilbert Matrix of order n
- Van der Monde Matrix of order n with $x_i = \frac{n+i+1}{n+1}$
- Matrix M-A (Cf. Wilkinson [Wil]):

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

- Matrix M-B (Cf. [CR]):

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

- Matrix M-C (Cf. [FMM]):

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

- Matrix M-D

$$\begin{aligned} MD_{i,i} &= n \\ MD_{i,j} &= \frac{i-j}{i+j+1} \quad \text{for } i \neq j \end{aligned}$$

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

$$\|E\|_\infty \quad \leq \quad 8 \quad n^3 \quad \rho \quad \|A\|_\infty \quad \mathbf{U_{err}}$$

with:

$$\rho = \frac{\max_{i,j,k} \left| \hat{a}_{ij}^{(k)} \right|}{\|A\|_\infty}$$

	RandP	no interchange for pivoting (std.dev.)	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

Figure 3: Results of Gaussian elimination for Matrix M-A, n=20

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

$$r = \|E\|_\infty \left\|A^{-1}\right\|_\infty$$

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

$$\frac{\|x - \hat{x}\|_\infty}{\|x\|_\infty} \leq \frac{r}{1-r}$$

We give below the computed values of r, taking for $\mathbf{U_{err}}$ the value corresponding to full precision on DEC/VAX namely $2^{-57} = 6.7\ 10^{-18}$. We also use the classical notation for the condition number of A: $\kappa_\infty(A) = \|A\|_\infty \left\|A^{-1}\right\|_\infty$.

5.2.1 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 result's precision, as is shown [9] on Figure 3, whereas the exact solution is $x_1 = x_{10} = 1.0$; $x_{20} = 2.0$. The *a priori* estimate is $r = 8.9\ \ 10^{-12}$. In this case, both methods are coherent, the FULL Precision result shows that they are both exaggerating the error.

5.2.2 Test with Matrix M-B

This set of matrices are 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 failure which occurs for $k = 1.0E8$. For $k = 1.0E7$ we obtain the results of Figure 4, which is to be compared with the exact solution $(1, 1, 1, 2.0000002)$. Note that here $r = 0.14$, and $\frac{r}{1-r} = 0.16$.

[9] We will show here only a few of the result's component together with the precision estimates, more extensive results will appear in [BLT2]

```
                    no  interchange for pivoting
        RandP       (std.dev.)   Model                FULL Precision

5.76000302517782e-01 (2.14)      1.0e+00( 0)          .997656464576721
1.42400000379421e+00 (2.14)      1.0e+00( 0)          1.00234353542328
9.99999576000l1e-01 (2.14e-07) 1.0000000E0           .999999999765646
2.00000020000001e+00 (3.87e-14) 2.000000200000000E0 2.00000020000000
```

Figure 4: Results of Gaussian elimination for Matrix M-B

5.2.3 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 (Figure 5), at least for the partial column pivoting (exact solution $(0, -1, 1)$, $r = 4.\ 10^{-12}$). In this case both the Modelling approach and the full precision show that total pivoting is certainly not justified, as it appears less precise.

5.2.4 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 Figure 6). The random perturbation technique performs satisfactorily too. Note that pivoting is not relevant in this case and that $r = 2.8\ 10^{-13}$, $\kappa_\infty(A) = 2.6$.

5.2.5 Test with 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 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 Figure 7, 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 ($r = 0.023$, $\kappa_\infty(A) = 1.17\ 10^{11}$, exact solution: $x_1, x_4 = 1; x_5, x_8 = 2$).

5.2.6 Test with 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

no interchange for pivoting

Model	FULL Precision	RandP	(std.dev.)
-3.9e-16(2)	-3.885780586188805e-16	-1.00073282993663e-13	(4.61e-13)
-1.000000000000	-1.00000000000000	-1.00000000000014	(6.65e-13)
1.00000000000000	1.00000000000000	9.99999999999996e-01	(9.65e-15)

partial pivoting

Model	FULL Precision	RandP	(std.dev.)
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

Model	FULL Precision	RandP	(std.dev.)
-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)

Figure 5: Results of Gaussian Elimination with Matrix M-C

no interchange for pivoting

Model	FULL Precision	RandP	(std.dev.)
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)

Figure 6: Results of Gaussian Elimination with Matrix M-D, n=20

no interchange for pivoting

Model	FULL Precision	RandP	(std.dev.)
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

Model	FULL Precision	RandP	(std.dev.)
x[1] 1.0e+00(5)	9.99999999803623e-01	1.00000001023833	(1.48e-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

Model	FULL Precision	RandP	(std.dev.)
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)

Figure 7: Results of Gaussian Elimination with Hilbert Matrix, n=8

		no interchange for pivoting		
	Model	FULL Precision	RandP	(std.dev.)
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)	2.00000024863102	1.99999187598520	(1.01e-04)
		column pivoting		
	Model	FULL Precision	RandP	(std.dev.)
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		
	Model	FULL Precision	RandP	(std.dev.)
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)

Figure 8: Results of Gaussian Elimination with Van der Monde matrix, n=10.

$n = 10$ appear in Figure 8, and show that Wilkinson's bound can be over pessimistic indeed. (Here $r = 15.15$, $\kappa_\infty(A) = 2.16\ 10^{12}$, 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 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: (Cf. e.g. [GIV])

- iteratively update the residual $r(k)$ at iteration k with the formula: $r(k) := r(k) - alpha * A.p(k-1)$ where $p(k-1)$ is the search direction at previous step;

	Model	FULL Precision	RandP	(std.dev.)
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)

Figure 9: CONJUGATE GRADIENT - N=20 - updating strategy

	Model	FULL Precision	RandP	(std.dev.)
x[1]	2.0e+00(21)	2.00000000000000	1.99999999999994	(4.70e-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)

Figure 10: CONJUGATE GRADIENT - N=20 - recompute strategy

- fully recompute $r(k)$ for each iteration: $r(k) := b(k) - A.x(k)$.

The results after 20 iterations (Figures 9 and 10) 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 method on the multifrontal code MA32 from the Harwell library. The results shown on Figure 11 show the precision obtained on the result, in ℓ_2 and ℓ_∞ norm, 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 Figure 12. Varying from 0.1 to 0.99 appears not to be interesting.

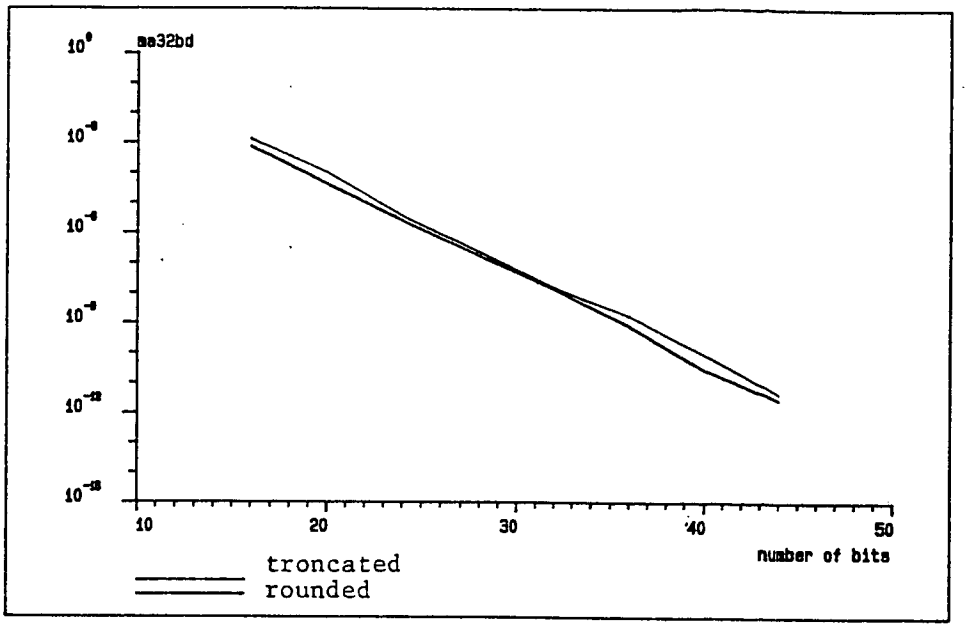

Figure 11: Effect of variable precision on MA32

α	cpu time(s)	# ops (internal loop)	relative error (L2 norm) for several mantissa lengths		
			36 bits	32 bits	28bits
0.001	0.199	196312	1.8×10^{-9}	3.1×10^{-8}	5.4×10^{-7}
0.01	0.199	196312	1.8×10^{-9}	3.1×10^{-8}	5.4×10^{-7}
0.1	0.204	206080	4.3×10^{-10}	6.4×10^{-9}	9.6×10^{-8}
0.99	0.704	921599	5.8×10^{-9}	1.5×10^{-7}	5.6×10^{-7}

Figure 12 : Cost of pivoting strategies in MA32 :
solution of system of linear equations of order 441, truncated arithmetic

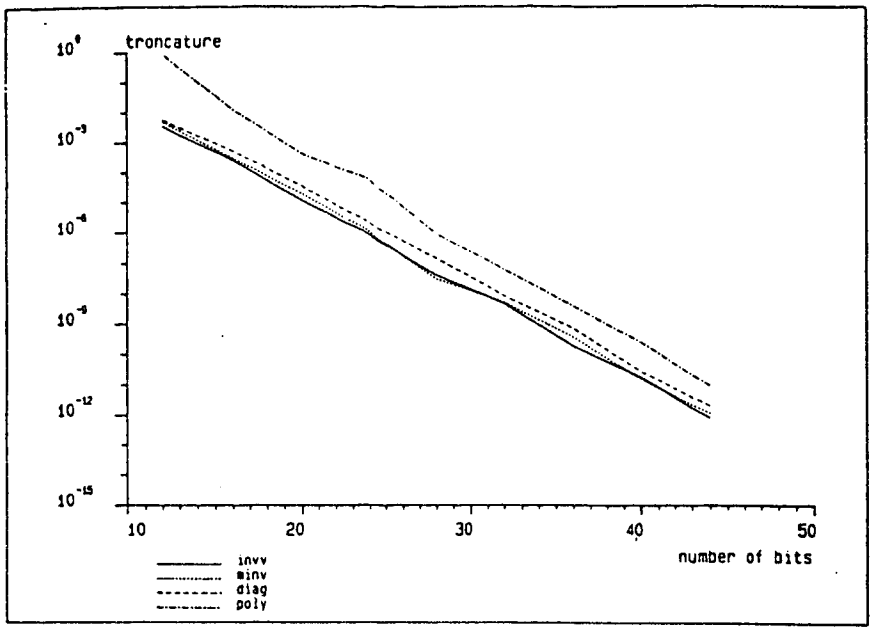

Figure 13: Effect of variable precision on Conjugate Gradient codes

7 Application to Iterative Methods

We have run a series of test on the algorithms DIAG, INV, MINV and POLY of Meurant & al. [CGM]. Both Truncation and Rounding methods show that the most precise results are obtained with the 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.

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 the use of these methods usefull in getting correct information on numerical precision, which we have found often very different from our intuition.

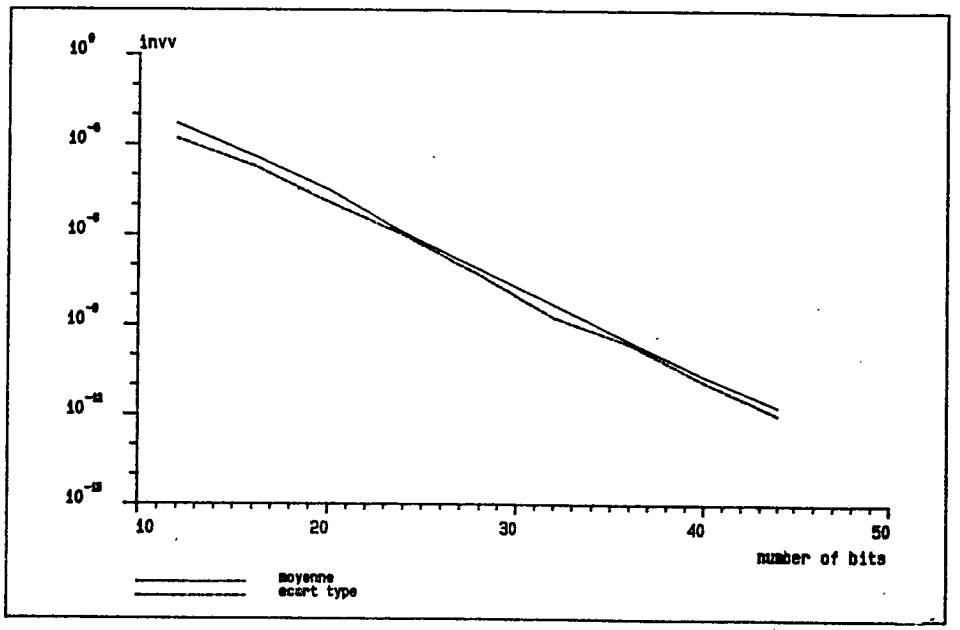

Figure 14: Random Perturbation error estimate for Conjugate Gradient codes

9 References

[AK] J.R.Allen & K.Kennedy, " PFC: a program to convert Fortran to parallel form ", *Supercomputers: Design and Applications, K. Hwang, editor, IEEE Press, 1985*

[BLTZ] F.Bourdoncle, A.Lichnewsky, F.Thomasset & P.Zimmermann, " ", *Rapport Technique INRIA, to appear.*

[BKM] R. Brent, D.J. Kuck & K. Maruyama, " The Parallel Evaluation of Arithmetic Expressions without Division ", *IEEE Trans. on Computers, Vol C-22, No 5, May 1973.*

[Br] W.S. Brown, " A Simple but Realistic Model of Floating Point computation ", *ACM Trans. Math. Software, Vol 7, No 4, December 1981*

[CrH] Cray Research Inc., " Cray 1S Series Hardware Reference Manual ", *HR-808, June 1980*

[CS] S. Chen & A. Sameh, " On Parallel Triangular Systems Solvers ", *Proc. Sagamore Comput. Conference, T. Feng ed.*

[CR] A.K.Cline & R.K. Rew, " A Set of Counter-Examples to three Condition Number Estimators ", *SIAM J. Sci. Stat. Comput., Vol 4, No 4, December 1983*

[CGM] P.Concus, G.H.Golub & G.Meurant, " Block Preconditioning for the Conjugate Gradient Method ", *Rep. Lawrence Livermore*

Berkeley Lab., LBL-14856, July 1986

[CK] W.J. Cody & H. Kuki, " A Statistical Study of the Accuracy of Floating Point Number Systems ", *Comm. ACM 1973*

[DBo] C. De Boor, " A practical guide to Splines ", *Springer-Verlag,1978*

[GvL] G.H. Golub & Ch.F. Van Loan, " Matrix Computations ", *The John Hopkins University Press,1983*

[Hel] D.Heller, " A Survey of Parallel Algorithms in Numerical Linear Algebra ", *SIAM Review, Vol 20, No 4, Oct 1978, pp. 740 - 777*

[JW86] P.Jansen & P.Weidner, " High-Accuracy Arithmetic Software - Some tests of the ACRITH Problem-Solving Routines ", *ACM Trans. on Math. Software, Vol12, No1, March 1986*

[KM] D.J.Kuck & Muraoka, " Bounds on the parallel evaluation of arithmetic expressions using associativity and commutativity ", *Acta Informatica 3 (1974),pp.203-216*

[KKLW] D.J.Kuck, R.H.Kuhn, B.Leasure & M.Wolfe, " The Structure of an Advanced Vectorizer for Pipelined Processors ", *Fourth Int. Computer Software and Applications Conference, Oct 1980*

[LT] A.Lichnewsky & F.Thomasset, " Techniques de Base pour l' Exploitation du Parallélisme dans les programmes ", *Rapp. Rech. INRIA No 460, 1985*

[LV] M. Laporte & J. Vignes, " Etude statistique des erreurs dans l' arithmétique des ordinateurs. Application au contrôle des résultats d' algorithmes numériques ", *Numer. Math. 23 (1974), pp 63-72*

[Mil] W. Miller, " Software for Roundoff Analysis ", *ACM Trans. on Math. Software*

[PR] B. Philippe & M. Raphalen , " Précision numérique dans le cumul d'un grand nombre de termes ", *Rapp. Rech. INRIA, 1985*

[Str] B. Stroustrup, " The C++ programming language ", *Addison Wesley, 1986*

[Ukk] E. Ukkonen, " On the Calculation of the Effects of Roundoff Errors ", *ACM Trans. Math. Software, Vol 7, No 3, September 1981*

[Wil] J.H. Wilkinson, " Rounding Errors in Algebraic Processes ", *Prentice Hall, Englewood Cliffs, N.J. 1963*

[Will] J.H. Wilkinson, " Error Analysis of Direct Methods of Matrix Inversion ", *JACM, No 8, 1961*

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique