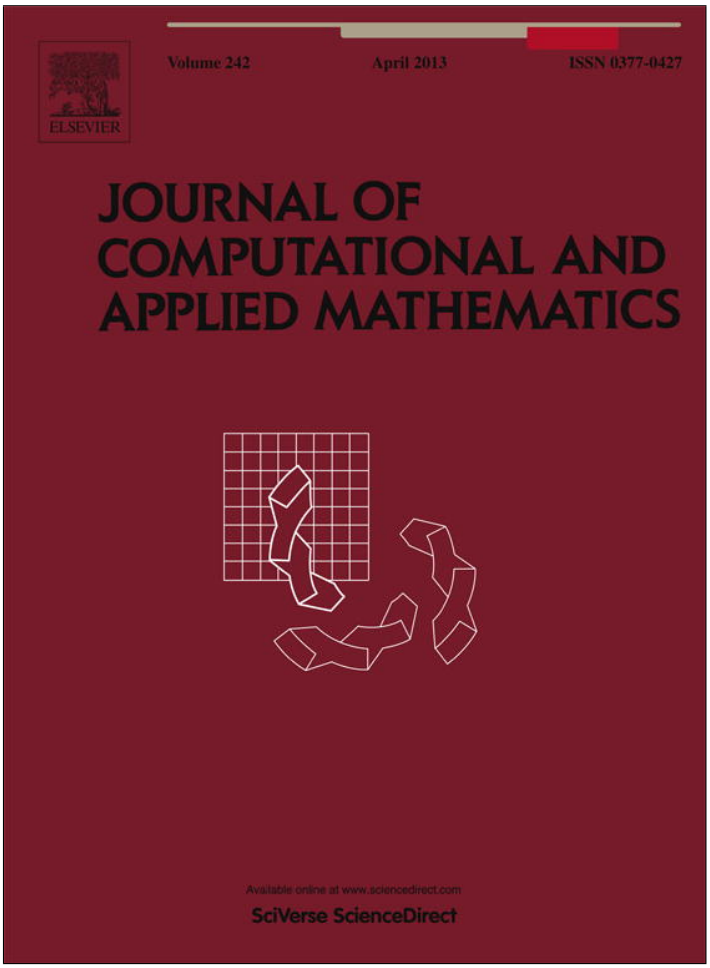

Journal of Computational and Applied Mathematics 242 (2013) 157–184

Contents lists available at SciVerse ScienceDirect

Journal of Computational and Applied Mathematics

journal homepage: www.elsevier.com/locate/cam

Accurate solution of dense linear systems, part I: Algorithms in rounding to nearest

Siegfried M. Rump *

Institute for Reliable Computing, Hamburg University of Technology, Schwarzenbergstraße 95, Hamburg 21071, Germany
Waseda University, Faculty of Science and Engineering, 3–4–1 Okubo, Shinjuku-ku, Tokyo 169–8555, Japan

ARTICLE INFO

Article history:
Received 12 August 2011
Received in revised form 22 August 2012

Keywords:
Linear system
Rounding to nearest
(extremely) ill-conditioned
Error-free transformation
Error analysis
Rigorous error bounds

ABSTRACT

We investigate how extra-precise accumulation of dot products can be used to solve ill-conditioned linear systems accurately. For a given p-bit working precision, extra-precise evaluation of a dot product means that the products and summation are executed in $2p$-bit precision, and that the final result is rounded into the p-bit working precision. Denote by $\mathbf{u} = 2^{-p}$ the relative rounding error unit in a given working precision. We treat two types of matrices: first up to condition number $\mathbf{u}^{-1}$, and second up to condition number $\mathbf{u}^{-2}$. For both types of matrices we present two types of methods: first for calculating an approximate solution, and second for calculating rigorous error bounds for the solution together with the proof of non-singularity of the matrix of the linear system. In the first part of this paper we present algorithms using only rounding to nearest, in Part II we use directed rounding to obtain better results. All algorithms are given in executable Matlab code and are available from my homepage.

1. Introduction and notation

The solution of a linear system $Ax = b$ is a ubiquitous task in numerical computations. In Part I and II of this paper we present different methods to compute guaranteed error bounds for the solution of a linear system, i.e. with a *certified* accuracy. The methods in this Part I are based on norm estimates, in particular verifying convergence of some residual matrix by approximating its Perron vector, whereas the methods in Part II are based on the verification of the H-property of some matrix. Moreover, in the present Part I of the paper we (1) present a method to compute an approximation for *extremely* ill-conditioned linear systems which is *likely* to be accurate.

In the present Part I all algorithms use only the four basic floating-point operations in rounding to nearest, in Part II directed rounding is used as well. The challenge for the first part is to use only standard Matlab code in rounding to nearest without additional mex-files and to derive simple and fast algorithms. All algorithms in both parts are presented in executable Matlab-code.

Let a floating-point format with relative rounding error unit $\mathbf{u}$ be given. The forward error of an approximation $\widetilde{x}$ computed by a standard algorithm like Gaussian elimination is of the order $\mathbf{u} \cdot \mathrm{cond}(A)$ [1]. This naturally bounds the applicability to matrices A with $\mathrm{cond}(A) \lesssim \mathbf{u}^{-1}$, which means $\mathrm{cond}(A) \lesssim 10^{16}$ in IEEE 754 double precision (binary64). For larger condition numbers, $\widetilde{x}$ is expected to have no correct digit.

Skeel [2] showed that one step of the classical residual iteration with the residual computed in *working precision* produces a backward stable result. It is also known that, for condition numbers up to $\mathrm{cond}(A) \lesssim \mathbf{u}^{-1}$, an almost maximally accurate

* Correspondence to: Institute for Reliable Computing, Hamburg University of Technology, Schwarzenbergstraße 95, Hamburg 21071, Germany.
E-mail address: rump@tu-harburg.de.

0377-0427/$ – see front matter
doi:10.1016/j.cam.2012.10.010

Table 1.1
Constants for single (binary32) and double (binary64) precision in the IEEE 754 floating-point standard.

	p	$e_{\min}$	$e_{\max}$
Single precision	24	−126	127
Double precision	53	−1022	1023

result (of order $\mathbf{u}$) is achieved if the dot products in the residual iteration are accumulated in twice the working precision (see also [3]). But for condition numbers of the order $\mathbf{u}^{-1}$ and larger, again no correct digit can be expected. So basically we face the dichotomy of either high accuracy or no accuracy at all.

We will show that the accumulation of dot products in twice the working precision (with result rounded into working precision) suffices to compute an accurate approximation of the solution of a linear system for condition numbers up to $\mathrm{cond}(A) \lesssim \mathbf{u}^{-2}$. Moreover, we show how rigorous error bounds can be computed including the proof on non-singularity of the input matrix A. Practical experience suggests that this approach works successfully up to condition numbers $c \cdot \mathrm{cond}(A) \lesssim \mathbf{u}^{-2}$ with $c \approx n^2$ in IEEE 754 binary64 (double precision). This factor will be improved to about $c \approx n$ in Part II of this paper.

We want to stress that our bounds are mathematically completely rigorous including all possible sources of errors (provided the compiler and operating system work to their specification). Although it is in principle known how to compute rigorous error bounds in rounding to nearest, the corresponding algorithms are involved, and taking care of underflow they become unwieldy. One reason to divide the paper in two parts is to clearly distinguish between algorithms using solely rounding to nearest (Part I), and those using directed rounding (Part II).

There are other, very good but not completely rigorous approaches. For example, an upper bound of the condition number $\mathrm{cond}(A)$ implies an error bound of an approximate solution of $Ax = b$. There are many $\mathcal{O}(n^2)$ condition number estimators (cf. [4,1]), usually providing good approximations. By the principle of the methods these are *lower bounds* for the condition number, and for every estimator counterexamples are known where the condition number is grossly underestimated.

In another approach [5,6] the authors use a statistic way for estimating rounding errors. Using a so-called stochastic arithmetic they propose a method to determine the number of significant digits of a computed result. Also those results are correct with a high degree of certainty, but not with complete rigor.

Yet another approach [3] uses the vast experience in solving linear systems very thoughtfully to produce approximations with "likely correct error terms" [3]. It seems that no counterexample is known where the claimed accuracy is not valid, but it is not *proved* to be correct.

To repeat it, beyond accurate approximations for very ill-conditioned linear systems, we are also interested in mathematically rigorous error bounds. Such rigorous bounds are, for example, mandatory in so-called "computer-assisted proofs" [7], which recently gain interest. For example, Tucker [8] received the 2004 EMS prize awarded by the European Mathematical Society for "giving a rigorous proof that the Lorenz attractor exists for the parameter values provided by Lorenz. This was a long standing challenge to the dynamical system community, and was included by Smale in his list of problems for the new millennium. The proof uses computer estimates with rigorous bounds based on higher dimensional interval arithmetics".

Concerning notation denote by $\mathbb{F}$ a set of p-bit binary floating-point numbers including ∞ and NaN, i.e. [9]

$$\mathbb{F} = \{M \cdot 2^{e-p+1} \mid M, e \in \mathbb{Z},\ |M| \le 2^p - 1,\ e_{\min} \le e \le e_{\max}\} \cup \{-\infty, +\infty, \mathrm{NaN}\}. \tag{1.1}$$

For single (binary32) and double (binary64) precision in the IEEE 754 floating-point standard [10,11] the constants are as in Table 1.1. We assume floating-point operations in rounding to nearest, tie to even, as in the IEEE 754 standard. That means there is a mapping $\mathrm{fl} : \mathbb{R} \to \mathbb{F}$ such that $|\mathrm{fl}(x) - x| = \min_{f \in \mathbb{F}} |f - x|$ for all $x \in \mathbb{R}$, and for $a, b \in \mathbb{F}$ floating-point operations $\circ_{\mathrm{fl}} : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ with $\circ \in \{+, -, \cdot, /\}$ are defined by

$$a \circ_{\mathrm{fl}} b := \mathrm{fl}(a \circ b). \tag{1.2}$$

Therefore the result $a \circ_{\mathrm{fl}} b \in \mathbb{F}$ is a best approximation of $a \circ b \in \mathbb{R}$. The relative rounding error unit is defined by $\mathbf{u} = 2^{-p}$. A floating-point number $M \cdot 2^{e-p+1}$ is normalized if $|M| \ge 2^{p-1}$, the smallest normalized positive floating-point number is $\texttt{realmin} = 2^{e_{\min}}$, and the smallest unnormalized positive floating-point number is $\texttt{eta} = 2^{e_{\min}-p+1}$. All algorithms are given in executable Matlab code using IEEE 754 double precision, but the results are valid in any floating-point arithmetic complying with the IEEE 754 standard.

Comparison between vectors and matrices is always to be understood entrywise, for example $x \le y$ for $x, y \in \mathbb{R}^n$ means $x_i \le y_i$ for $1 \le i \le n$. Executable Matlab-code is written using the "verbatim"-font. For instance, `C=A*B` means that `C` is the result of the floating-point multiplication `A*B`, where `A` and `B` are compatible quantities (scalar, vector, matrix). For analyzing the error we use ordinary mathematical notation, for example in $P = \mathtt{A} \cdot \mathtt{B}$ the verbatim-font is used for floating-point quantities so that P is the exact (real) product of `A` and `B`. For $\mathtt{A}, \mathtt{B} \in \mathbb{F}^{n \times n}$ this implies $|P - \mathtt{C}| \sim \mathbf{u}|\mathtt{A}| \cdot |\mathtt{B}|$.

Table 2.1
Results for `n=10; A=hilb(10)` and `x=A \ b` and `y=inv(A)*b`.

	$\|A\tilde{x}-b\|_2$	$\|A\tilde{y}-b\|_2$	$\|A^{-1}b-\tilde{x}\|_2/\|A^{-1}b\|_2$	$\|A^{-1}b-\tilde{y}\|_2/\|A^{-1}b\|_2$
`b = randn(n,1);`	$2.1\cdot 10^{-4}$	$1.7\cdot 10^{-3}$	$1.2\cdot 10^{-4}$	$1.5\cdot 10^{-5}$
`b = A*randn(n,1);`	$1.2\cdot 10^{-16}$	$3.9\cdot 10^{-5}$	$5.3\cdot 10^{-5}$	$1.4\cdot 10^{-4}$

In principle, it is not difficult to transform an algorithm using directed rounding into an algorithm using only rounding to nearest [12,13]. However, often the code becomes unwieldy. In this paper we use new estimates on floating-point summation and dot products [14] to derive particularly simple algorithms to compute verified error bounds using only rounding to nearest.

The paper is organized as follows. In Section 2 we describe an algorithm for calculating an approximation of the solution of extremely ill-conditioned linear systems, i.e. up to condition number $\mathbf{u}^{-2} \approx 10^{32}$ in IEEE 754 double precision. An algorithm for the necessary extra-precise evaluation of dot products is presented in Section 3. It uses only basic floating-point operations in rounding to nearest. The following Section 4 splits in five subsections including the computation of rigorous error bounds in rounding to nearest for ill-conditioned linear systems, for extremely ill-conditioned linear systems as well as for extra-precise dot product accumulation. This suggests a hybrid algorithm addressing both ill-conditioned and extremely ill-conditioned systems which will be discussed in Part II of this paper. Computational results and a conclusion finish the paper.

2. Accurate approximations for very ill-conditioned linear systems

The aim of this section is an algorithm to compute an accurate approximation of a linear system $Ax = b$ with $\mathbf{u}^{-1} \leq \mathrm{cond}(A) \lesssim \mathbf{u}^{-2}$. Besides the basic floating-point operations $\{+, -, \cdot, /\}$ the algorithm requires only the accumulation of dot products in twice the working precision. For $x, y \in \mathbb{F}^n$ this is denoted by

$$\texttt{res} = \texttt{Dot2Near(x', y)}; \quad \text{\% accumulation of dot product } x^T y \text{ in twice the working precision.} \tag{2.1}$$

For given p-precision $x, y \in \mathbb{F}^n$ this means that $\texttt{res} \in \mathbb{F}$ is the result obtained when calculating the products $x_i y_i$ in 2p-precision, accumulating the sum $\sum x_i y_i$ in 2p-precision and rounding the result of the sum into working precision (p-precision). Therefore the error of `res` is bounded by $\mathbf{u}|\texttt{res}| + c\mathbf{u}^2\big(|x|^T|y|\big)$ for a small constant c, the second term addressing the accumulation error and the first term the rounding into working precision. In [3] this is called "extra-precise" accumulation of dot products. We use similarly

$$\begin{aligned} &\texttt{res} = \texttt{Dot2Near(R, b)}; \quad \text{\% accumulation of dot products in } R\cdot b \text{ in twice the working precision} \\ &\texttt{res} = \texttt{Dot2Near(R, A)}; \quad \text{\% accumulation of dot products in } R\cdot A \text{ in twice the working precision.} \end{aligned} \tag{2.2}$$

There is a vast amount of literature devoted to the accurate computation of sums and dot products, among them [15–26]. One way to implement `Dot2Near` using only basic floating-point operations in working precision is XBLAS [27,28]. Here two IEEE 754 double precision (binary64) floating-point numbers are used to represent a quadruple precision number, and accurate arithmetical operations on pairs are defined. This does more than necessary for our purposes because the result is a pair, but for `Dot2Near` we need only the first part. Another way is to use some multiple-precision package like [29,30].

Since the accurate computation of residuals, possibly with rigorous error bounds, is of central importance for this paper, we discuss several methods and give executable code in Section 3. For the moment assume an algorithm `Dot2Near` as described to be given.

Let $A \in \mathbb{F}^{n\times n}$ with $\mathrm{cond}(A) \geq \mathbf{u}^{-1}$ be given. Then an approximation $\tilde{x}$ of the solution of a linear system with matrix A computed by Gaussian elimination in double precision is expected to be heavily corrupted by rounding errors, and no digit of $\tilde{x}$ can be expected to be correct.

It is well-known [31,32] that "in the vast majority of practical computational problems, it is unnecessary and inadvisable to actually compute A^{-1}". In particular an approximation $\tilde{y} := Rb$ using some approximate inverse R of A requires not only three times as much operations than Gaussian elimination, it is also, in general, less accurate and less stable [32]. Nevertheless an approximate inverse is our key to solve such extremely ill-conditioned linear systems.

More precisely, numerical evidence suggests that on the one hand, depending on the right hand side b, the residual $\|A\tilde{x} - b\|_2$ for $\tilde{x} = A \setminus b$ is sometimes much smaller than $\|A\tilde{y} - b\|_2$ with $\tilde{y} = Rb$. However, there seems to be on the other hand, in general, not much difference between $\|A^{-1}b - \tilde{x}\|_2/\|A^{-1}b\|_2$ and $\|A^{-1}b - \tilde{y}\|_2/\|A^{-1}b\|_2$. Results for a not untypical example are given in Table 2.1.

Let R be an approximate inverse computed in working precision (e.g. by the Matlab command `R=inv(A)`), and assume $\mathrm{cond}(A) \geq \mathbf{u}^{-1}$. Then R is also expected to be entirely corrupted by rounding errors with no correct digit. Nevertheless the approximate inverse R contains useful information. This corresponds to the fact that the rounding errors in Gaussian elimination are by no means random, see [33,34].

In about 1984 I derived an algorithm squeezing out this information. Because of lack of analysis, I did not publish it. In [35] Oishi et al. analyzed a modification of this algorithm, and in [36] I analyzed the original algorithm. It requires the

accumulation of dot products in some K-fold precision and storing the result in an unevaluated sum of L floating-point numbers; otherwise floating-point operations in working precision are used.

For this algorithm it is important to store an approximate inverse in an unevaluated sum of matrices $R_1 + \cdots + R_k$, where the summands are computed recursively starting with the first approximate inverse R. Under reasonable assumptions it is shown in [36] that with k summands matrices A up to condition number $\mathbf{u}^{-k}$ can be treated, i.e. the spectral radius of $I - \left(\sum_{\nu=1}^{k} R_\nu\right) A$ becomes less than 1.

The first step of this algorithm is given by the following executable Matlab code. We abbreviate "accumulation of dot products in twice the working precision and rounding into working precision" by "extra-precise accumulation" in the comments.

```
R = inv(A);                                  % approximate inverse
while any(isinf(R(:))) || any(isnan(R(:)))
  R = inv(A.*(1+randn(n)*eps));              % inversion of perturbed matrix
end
C = Dot2Near(R,A);                           % extra-precise accumulation
Cinv = inv(C);                               % multiplicative correction for R
```

For $\mathtt{A} \in \mathbb{F}^{n\times n}$ with $\mathrm{cond}(\mathtt{A}) \geq \mathbf{u}^{-1}$ it may happen that the "approximate inverse" R computed in the first line contains infinity- and NaN-components. In that case the matrix A is slightly perturbed and inverted again. Note that mathematically, due to the large condition number of A, this may change the entries of R in the first digit. In any case, R is completely corrupted by rounding errors.

Nevertheless it can be observed that

$$\mathrm{cond}(\mathtt{C}) \sim \mathbf{u} \cdot \mathrm{cond}(\mathtt{A}), \quad \text{even for } \mathrm{cond}(A) \gg \mathbf{u}^{-2}. \tag{2.3}$$

We cannot expect a mathematically rigorous analysis, but in [36] arguments are given for that. Examples with condition number up to 10^{300}, just before overflow, confirm this observation.

For $\mathrm{cond}(\mathtt{A}) \sim \beta\mathbf{u}^{-1}$ this means $\mathrm{cond}(\mathtt{C}) \sim \beta$, so that for $\beta \lesssim \mathbf{u}^{-1}$ we can expect some accuracy in Cinv. The next step in [36] is to compute Cinv*R in twice the working precision but to store the result in an unevaluated sum $\mathtt{R}_1 + \mathtt{R}_2$. Then it is shown that $I - (\mathtt{R}_1 + \mathtt{R}_2)\mathtt{A}$ is convergent for $\mathrm{cond}(\mathtt{A}) \lesssim \mathbf{u}^{-2}$.

One might use the single matrix $\mathtt{Dot2Near}(\mathtt{Cinv}, \mathtt{R}) \in \mathbb{F}^{n\times n}$, certainly a good if not best approximation to Cinv*R, directly as a preconditioner for A. However, for $\mathrm{cond}(\mathtt{A}) > \mathbf{u}^{-1}$ a single preconditioning matrix $\mathtt{B} \in \mathbb{F}^{n\times n}$ cannot, in general, force $I - BA$ to be convergent. Even for B being the nearest floating-point matrix to $\mathtt{A}^{-1}$, that is $\mathtt{B} = \mathtt{A}^{-1} + \Delta$ with $\|\Delta\| \sim \mathbf{u}\|\mathtt{A}^{-1}\|$ for some norm, we have $\|I - \mathtt{B}\cdot\mathtt{A}\| = \|\Delta \cdot \mathtt{A}\| \sim \mathbf{u} \cdot \mathrm{cond}(\mathtt{A}) > 1$. Thus the unevaluated sum $\mathtt{R}_1 + \mathtt{R}_2$ rather than Cinv*R is necessary in [36] to serve as a preconditioning matrix.

For computing an accurate approximation of the solution of a linear system, even for $\mathrm{cond}(\mathtt{A}) > \mathbf{u}^{-1}$, no unevaluated sum as a preconditioning matrix is necessary. In the following algorithm we use only Dot2Near as specified in (2.2), i.e. accumulation of dot products in twice the working precision with the result stored in working precision, and only floating-point operations in rounding to nearest.

Algorithm 2.1. Accurate approximate solution xs of $Ax = b$ for extremely ill-conditioned A.

```
function xs = LssIllcoApprox(A,b)
  n = size(A,1);                             % dimension of linear system
  R = inv(A);                                % approximate inverse
  while any(isinf(R(:))) || any(isnan(R(:)))
    R = inv(A.*(1+randn(n)*eps));            % inversion of perturbed matrix
  end
  C = Dot2Near(R,A);                         % extra-precise accumulation
  Cinv = inv(C);                             % multiplicative correction for R
  xs = Cinv*Dot2Near(R,b);                   % first approximate solution
  N = inf; iter = 0;                         % initialization of constants
  while iter<5                               % at most 5 residual iterations
    iter = iter+1; Nold = N;                 % update constants
    res = Dot2Near([A b],[xs;-1]);           % residual A*xs-b (extra-precise acc.)
    d = Cinv*Dot2Near(R,res);                % correction for xs
    N = norm(d,1);                           % norm of correction
    if N<Nold, xs = xs-d; end                % correction acceptable
    if N>=0.1*Nold, break, end               % stop iteration if no improvement
  end
```

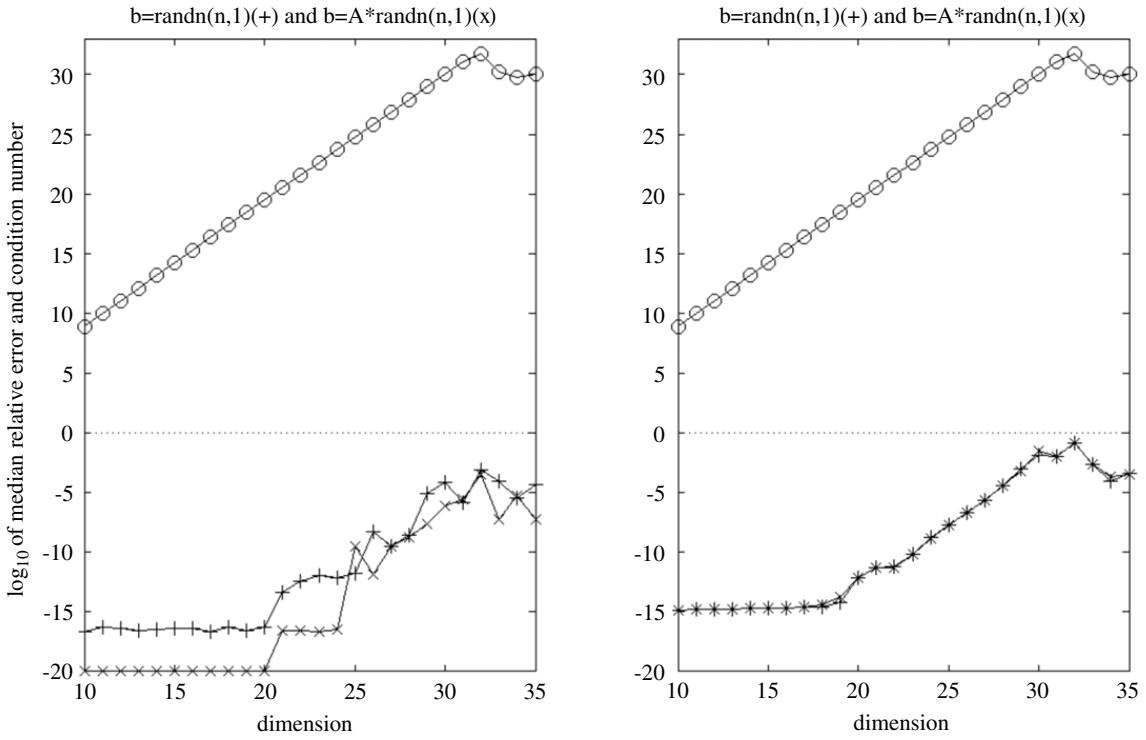

Fig. 2.1. Results of Algorithm 2.1 (`LssIllcoApprox`) (left) and of Algorithm 4.20 (`LssIllcoErrBndNear`) (right) for Pascal matrices with right hand sides `b=randn(n,1)` and `b=A*randn(n,1)`. The upper parts show the condition number, the lower parts the relative error of the results.

Observation 2.2. *Let* $\mathbb{F}$ *be a set of floating-point numbers with relative rounding error unit* $\mathbf{u}$ *together with floating-point operations complying with the IEEE* 754 *arithmetic standard* [10,11]. *Let* `xs` *be the result of Algorithm* 2.1 (`LssIllcoApprox`) *applied to a matrix* $\mathtt{A} \in \mathbb{F}^{n\times n}$ *with* $\mathrm{cond}(\mathtt{A}) \lesssim \mathbf{u}^{-2}$ *and to a right hand side* $\mathtt{b} \in \mathbb{F}^n$. *Then numerical evidence suggests that the relative error of* `xs` *to the exact solution* $\mathtt{A}^{-1}\mathtt{b}$ *is of the order* $\mathbf{u} + \mathbf{u}^2\mathrm{cond}(\mathtt{A})$.

To start with, we do not fully understand why Algorithm 2.1 works that good for extremely ill-conditioned matrices. Obviously $(RA)^{-1}R = A^{-1}$, so that without the presence of rounding errors $\mathtt{Cinv}\cdot\mathtt{R}\cdot\mathtt{b} = \mathtt{A}^{-1}\mathtt{b}$. This is the main idea explored in [36]. Over there, we identified $\mathtt{Cinv}\cdot\mathtt{R}$ as a suitable preconditioning matrix for A; here we are interested in solving the linear system and compute $\mathtt{Cinv}\cdot(\mathtt{R}\cdot\mathtt{b})$.

An approximate solution `xs` is computed in line 9, and in lines 11–18 some residual iteration is applied to it. An approximate solution `d` of the residual system is computed in the same way. Since `d` is the correction to `xs`, the pair (`xs`,`d`) might be regarded as an unevaluated sum. However, the pair is not proceeded but added (in working precision) into the new `xs` in line 16.

The extra-precise accumulation of dot products is used for the preconditioning $\mathtt{R}\cdot\mathtt{A}$ in line 7, for the residual $\mathtt{A}\cdot\mathtt{xs} - \mathtt{b}$ in line 13, and for the multiplication of a right hand side by `R` in lines 9 and 14. If one of these dot product operations is replaced by the usual computation in working precision, the results deteriorate significantly or the algorithm fails completely. One may be inclined to use `Dot2Near` for the multiplication by `Cinv` in lines 9 and 14 as well; however, numerical evidence suggests that usually this does not improve the results (sometimes to the contrary): only in some extreme situations it is advantageous.

To this end we give one typical example for the performance of Algorithm 2.1 (`LssIllcoApprox`). It is not obvious how to construct an extremely ill-conditioned matrix with floating-point entries.[1] Ways to construct ill-conditioned floating-point matrices are introduced in [37–39]. Here we use Pascal matrices defined by $A_{ij} := \binom{i+j-2}{j-1}$. Up to dimension $n = 31$ the entries are double precision floating-point numbers; for $n > 31$ the entries are rounded to the nearest floating-point number. To increase numerical stability we use always equilibrated matrices as in [3], see Section 5.

As can be seen in Fig. 2.1, the condition number (upper line, computed by the symbolic toolbox in Matlab) increases monotonically up to dimension $n = 32$, then the matrix entries become corrupted by the conversion into floating-point. Although the Pascal matrix is not exactly representable for $n = 32$, the condition number is accidentally larger than for $n = 31$.

[1] For a discussion of "well-known suspects" of ill-conditioned matrices A and alternative ways to solve $Ax = b$ see the beginning of Section 5.

The lower lines in the left graph show the median relative error of the approximation computed by Algorithm 2.1 against the exact solution computed by the Matlab symbolic toolbox: for given $\mathtt{A} \in \mathbb{F}^{n\times n}$ and $\mathtt{b} \in \mathbb{F}^n$, the solution $x \in \mathbb{Q}^n$ of the linear system as a vector of rational numbers is computed and compared to the computed approximation $\mathtt{xs} \in \mathbb{F}^n$. For two different right hand sides `b = randn(n,1)` (+) and `b = A*randn(n,1)` (x) the median relative error of `xs` to x is displayed. The right graph shows the quality of rigorous error bounds computed by Algorithm 4.20 (`LssIllcoErrBndNear`) and will be discussed later.

It can be seen that up to condition number 10^{16} the approximate solution is of full accuracy, whereas for condition number 10^k with $k > 16$ about $32 - k$ digits are correct. This acknowledges Observation 2.2. Extensive numerical tests with various types of matrices and right hand sides confirm that up to condition numbers of about 10^{30} the computed approximations can be expected to have some accuracy, see Section 5.

To this end, we can deduce the accuracy of the computed approximation by *knowing* the exact solution by some oracle or by computing it in rational arithmetic. In the next sections we show how mathematically rigorous error bounds can be *computed* in floating-point rounding to nearest.

3. Accurate dot products in rounding to nearest

Following we describe an algorithm realizing `Dot2Near` as used in Section 2. Although Algorithm 2.1 (`LssIllcoApprox`) needs only an accurate *approximation*, the following algorithm also computes rigorous error bounds for a dot product. Therefore it is also suitable for our algorithms computing rigorous error bounds to be discussed in the next section.

The algorithm to be presented requires only the basic floating-point operations $+, -, \cdot, /$ in working precision and no additional features such as access to mantissa and/or exponent, assembly language routines or alike, and the algorithm is also free of branches. This improves, as analyzed by Langlois [40], the computational performance on today's architectures significantly.

In [18] Neumaier, as a young student, developed a fast algorithm with provably improved accuracy. It computes an approximation of a dot product with a quality "as if" computed in twice the working precision. His paper is written in German and did not receive wide attention. A modern formulation of this algorithm was presented in [41] and is based on error-free transformations.

We first need an error-free transformation to split a floating-point number into a high and low order part. This is done by Dekker's method as in Algorithm 3.1.

Algorithm 3.1. Error-free splitting of a floating-point number `a` into two parts `x`, `y` such that $\mathtt{a} = \mathtt{x} + \mathtt{y}$.

```
function [x,y] = Split(a,s)
  c = (2^s+1)*a;                % splitting in (53-s)-bit and (s-1)-bit part
  x = c - (c-a);                % high order part
  y = a - x;                    % low order part
```

As a result, $\mathtt{a} = \mathtt{x} + \mathtt{y}$ for all $\mathtt{a} \in \mathbb{F}$, also in the presence of underflow. Moreover, for 53-bit precision input `a`, the summands `x` and `y` have at most $53 - s$ and $s - 1$ significant bits, respectively. So for $s = 27$, both summands have at most 26 significant bits adding to a 53-bit number, an apparent contradiction. This is possible because Dekker's ingenious and fast algorithm uses the sign bit as an extra bit of information. Note that Algorithm 3.1 works correctly for vector and matrix input as well, and possible overflow may be avoided by some scaling.

For completeness we repeat algorithms `TwoSum` and `TwoProduct`, error-free transformations of the sum and product of two floating-point numbers into the nearest floating-point approximation and the true error, respectively.

Algorithm 3.2. Error-free transformation of $\mathtt{a} + \mathtt{b}$ into $\mathtt{x} + \mathtt{y}$.

```
function [x,y] = TwoSum(a,b)
  x = a + b;                        % floating-point approximation of a+b
  z = x - a;
  y = ( a - (x-z)) + (b-z);         % exact error of x
```

Algorithm 3.3. Error-free transformation of $\mathtt{a} \cdot \mathtt{b}$ into $\mathtt{x} + \mathtt{y}$.

```
function [x,y] = TwoProduct(a,b)
  x = a * b;                        % floating-point approximation of a*b
  [a1,a2] = Split(a,27);            % error-free splitting a = a1+a2
  [b1,b2] = Split(b,27);            % error-free splitting b = b1+b2
  y = a2*b2 - (((x-a1*b1)-a2*b1)-a1*b2);   % exact error of x (if no underflow)
```

Algorithm 3.2 (TwoSum) is due to Knuth [42], and Algorithm 3.3 (TwoProduct) is due to G.W. Veltkamp (see [43]). The two algorithms satisfy for all $\mathtt{a}, \mathtt{b} \in \mathbb{F}$ the rigorous error estimates [9]

$$\begin{aligned} &[x, y] = \mathtt{TwoSum}(a, b) \Rightarrow a + b = x + y \quad \text{and} \\ &[x, y] = \mathtt{TwoProduct}(a, b) \Rightarrow a \cdot b = x + y + \eta \quad \text{with } |\eta| \leq 3\mathtt{eta}. \end{aligned} \tag{3.1}$$

Recall that eta denotes the smallest positive (unnormalized) floating-point number. In IEEE 754 double precision $\mathtt{eta} = 2^{-1074}$, so that the smallest positive normalized floating-point number is $\mathtt{realmin} = \frac{1}{2}\mathbf{u}^{-1}\mathtt{eta}$.

Note that the results of TwoSum always satisfy $\mathtt{x} + \mathtt{y} = \mathtt{a} + \mathtt{b}$, also in the presence of underflow, whereas the results of TwoProduct satisfy $\mathtt{x} + \mathtt{y} = \mathtt{a} \cdot \mathtt{b}$ if no underflow occurs.

The routine TwoSum requires 17 floating-point operations. The new IEEE 754 floating-point standard [11] requires an FMA (Fused Multiply and Add) operation, which is already available on some processors. It computes $\mathtt{a} \cdot \mathtt{b} + \mathtt{c}$ with one final rounding to nearest. With FMA, TwoProduct can be replaced by

```
x = a*b;
y = FMA(a,b,-x);
```

thus requiring only two floating-point operations instead of 17. Based on those routines a summation algorithm was presented in [41], which is almost identical with Neumaier's [18]. We formulate it directly for matrix multiplication using rank-1 updates.

Algorithm 3.4. Approximation of the matrix product $\mathtt{A} \cdot \mathtt{B}$ "as if" accumulated in twice the working precision and rounded into working precision with rigorous error term.

```
    function [res,err] = Dot2Near(A,B)
      [p,e] = TwoProduct(A(:,1),B(1,:));    % error-free transformation of first product
*     E = abs(e);                           % for error term
      k = size(A,2);                        % inner dimension
      for i=2:k                             % matrix product by rank-1 updates
        [h,r] = TwoProduct(A(:,i),B(i,:));  % error-free transformation of i-th product
        [p,q] = TwoSum(p,h);                % error-free transformation of accumulated sum
        t = q + r;                          % sum of errors
        e = e + t;                          % accumulation of errors
*       E = E + abs(t);                     % accumulation for error term
      end
      res = p + e;                          % extra-precise approximation
*     epss = 0.5*eps;                       % relative rounding error unit 2^(-53)
*     err0 = max(6*k*epss,1)*realmin + (k+2)*epss*ufp(E) + epss*ufp(res);
*     err = err0 + 3*epss*ufp(err0);        % rigorous error bound for res
```

Apparently Neumaier did not know about the error-free transformations TwoSum and TwoProduct. He developed his algorithm as a sequence of floating-point operations with some reminiscence to the Kahan–Babuška algorithm [44].

Here we added a simplified computation of a rigorous error term which we need and explain later. It is based on the new analysis of floating-point summation in [14]. The error term is computed in the lines marked with an asterisk. If no error term is needed, all those lines can be omitted.

Theorem 3.5. *Let $A \in \mathbb{F}^{m\times k}$ and $B \in \mathbb{F}^{k\times n}$ with $(k+2)\mathbf{u} \leq 1$ be given, and let* res *and* err *be the quantities computed by Algorithm 3.4 (*Dot2Near*). Then, also in the presence of underflow,*

$$|\mathtt{A} \cdot \mathtt{B} - \mathtt{res}| \leq \mathbf{u}|\mathtt{A} \cdot \mathtt{B}| + \gamma_k^2|\mathtt{A}||\mathtt{B}| + 5k\mathtt{eta}, \tag{3.2}$$

where eta *denotes the smallest positive (unnormalized) floating-point number. Moreover,*

$$|\mathtt{A} \cdot \mathtt{B} - \mathtt{res}| \leq \mathtt{err}. \tag{3.3}$$

The factor $k + 2$ in the second last line of Algorithm 3.4 cannot be replaced by $k + 1$.

The first estimate (3.2) follows by the corresponding estimate in [41] for dot products, the correctness of the error bound (3.3) will be shown in Section 4.2. The first term in the right hand side of (3.2) reflects the unavoidable error of rounding the final sum res=p+e into working precision, the second term reflects the accuracy of the unevaluated sum $\mathtt{p} + \mathtt{e}$ before this addition, and the third term covers possible underflow. This means that the quality is "as if" computed in twice the working precision and rounded into working precision as in (2.2).

The number of floating-point operations up to $\mathcal{O}(1)$ for Dot2Near is as in Table 3.1. For comparison, the data for XBLAS [27] are displayed as well.

Table 3.1
Number of floating-point operations of `Dot2Near` without and with error bound, and without and with FMA.

	Without FMA	With FMA
`Dot2Near` without error bound	$25n$	$10n$
`Dot2Near` with error bound	$27n$	$12n$
`DotXBLAS` without error bound	$37n$	$22n$

4. Rigorous error bounds for linear systems including extremely ill-conditioned matrices with $\mathrm{cond}(A) \lesssim \mathbf{u}^{-2}$

For a matrix $E \in \mathbb{R}^{n\times n}$ with $\|E\| < 1$ in some matrix norm it is well-known [4,45] that $I \pm E$ is non-singular and $\|(I \pm E)^{-1}\| \le (1 - \|E\|)^{-1}$. For any given R, $A \in \mathbb{R}^{n\times n}$ and $\tilde{x}$, $b \in \mathbb{R}^n$, $\|I - RA\| < 1$ implies A (and R) to be non-singular and

$$\|\widetilde{x} - A^{-1}b\|_\infty = \|\big(I - (I - RA)\big)^{-1}R(A\tilde{x} - b)\|_\infty \le \frac{\|R(A\tilde{x} - b)\|_\infty}{1 - \|I - RA\|_\infty}. \tag{4.1}$$

Moreover, we may use $(I - E)^{-1} = (I - E^2)^{-1}(I + E)$ for $E := I - RA$ to deduce

$$\|\widetilde{x} - A^{-1}b\|_\infty \le \frac{\|(I + E)R(A\tilde{x} - b)\|_\infty}{1 - \|I - RA\|_\infty^2}. \tag{4.2}$$

If $\|I - RA\|_\infty$ is not close to one, then the accuracy of the bounds is determined by the size of the residual $\|A\tilde{x} - b\|_\infty$. The computation of the residual is, of course, subject to heavy cancellation. However, when computed with Algorithm 3.4 (`Dot2Near`) presented in the previous section we may expect accurate error bounds for the maximum error of $\tilde{x}$ to the exact solution $A^{-1}b$.

Both (4.1) and (4.2) are uniform, normwise error bounds for all entries of the approximation $\widetilde{x}$. If the entries of $\widetilde{x}$ differ largely in magnitude, it is superior to use the following entrywise error estimate by Yamamoto [46].

Theorem 4.1. *Let A, $R \in \mathbb{R}^{n\times n}$ and b, $\widetilde{x} \in \mathbb{R}^n$ be given. Define $E := I - RA$ and $\delta := R(A\widetilde{x} - b)$, and assume $\|E\|_\infty < 1$. Then A is non-singular and*

$$|\widetilde{x} - A^{-1}b| \le |\delta| + \frac{\|\delta\|_\infty}{1 - \|E\|_\infty} \cdot |E|e, \tag{4.3}$$

where $e := (1, \ldots, 1)^T \in \mathbb{R}^n$.

Proof. For u, v, $x \in \mathbb{R}^n$ it holds $|u^T v| \le \|u\|_1\|v\|_\infty$, and therefore $|Ex| \le \|x\|_\infty \cdot |E|e \in \mathbb{R}^n$. Hence $(I - E)^{-1} = I + E(I - E)^{-1}$ yields

$$\begin{aligned}|\widetilde{x} - A^{-1}b| &= |(I - E)^{-1}R(A\widetilde{x} - b)| = |(I + E(I - E)^{-1})\delta| \le |\delta| + \|(I - E)^{-1}\delta\|_\infty \cdot |E|e \\ &\le |\delta| + \frac{\|\delta\|_\infty}{1 - \|E\|_\infty} \cdot |E|e. \quad \square\end{aligned}$$

Next we describe how to compute a rigorous upper bound of the right hand side in (4.3) in rounding to nearest. It is applicable up to condition numbers of about $\mathbf{u}^{-1}$, and it is valid under all circumstances, also in the presence of underflow.

A result in overflow is rounded in IEEE 754 to $\pm\infty$, which means for all of the following algorithms that the final result contains either ∞ or NaN. Therefore we may safely assume that no intermediate overflow occurs because this is monitored in the final result.

4.1. Bounds for summation and dot product in rounding to nearest (accumulation in working precision)

Algorithm 3.4 (`Dot2Near`) estimates the error of a dot product in rounding to nearest. However, often a less accurate dot product, only accumulated in working precision, is sufficient. Let x, $y \in \mathbb{F}^n$ be given, and denote by $\widetilde{s}$ the result of the floating-point approximation of $x^T y$ computed by a standard for-loop. Then, provided no underflow occurs, the classical Wilkinson estimate [1] is

$$|\widetilde{s} - x^T y| \le \gamma_n |x^T||y| \quad \text{for } n\mathbf{u} < 1, \tag{4.4}$$

where $\gamma_n := n\mathbf{u}/(1 - n\mathbf{u})$. Since the computation of γ_n in floating-point causes again rounding errors to be controlled, the code for rigorous estimates is unwieldy, in particular if underflow is allowed. Moreover, the right hand side is not known.

Fortunately there is a simple way to avoid the nasty γ_n terms but to obtain nevertheless rigorous error estimates *including possible underflow*. For given $\mathtt{A} \in \mathbb{F}^{m\times k}$ and $\mathtt{B} \in \mathbb{F}^{k\times n}$ consider the following Algorithm 4.2, where `realmin` is the Matlab constant denoting the smallest positive normalized floating-point number.[2]

[2] The rounding error unit `eps` in Matlab is $2\mathbf{u}$.

Algorithm 4.2. Rigorous error bound $|\mathtt{A}\cdot\mathtt{B}-\mathtt{res}|\le\mathtt{err}$, also in the presence of underflow.

```
function [res,err] = DotErr(A,B)
  res = A*B;                                % approximation of product
  D = abs(A)*abs(B);                        % product of absolute values
  U = ufp(D);                               % unit in the first place of D
  k = size(A,2);                            % inner dimension
  err = (k+2)*(eps/2*U) + 1.5*realmin;      % error bound for res
```

We assume that the products `A*B` and `abs(A)*abs(B)` is executed in the same order of evaluation. The algorithm uses the "unit in the first place" $\mathrm{ufp}(r)$ of $r\in\mathbb{R}$ defined by

$$\mathrm{ufp}(r) := 2^{\lfloor\log_2|r|\rfloor}, \tag{4.5}$$

with $\mathrm{ufp}(0) := 0$, which may be smaller than $|r|$ by up to a factor of 2. Algorithm 3.5 in [22], which we repeat for convenience, computes the unit in the first place in floating-point rounding to nearest without branch.

Algorithm 4.3. Unit in the first place of a floating-point number, vector or matrix.

```
function res = ufp(x)
  q = (1/eps+1)*x;                          % eps = 2^(-52) in double precision
  res = abs(q-(1-eps/2)*q);                 % unit in the first place of x
```

A possible overflow in the computation of q may be avoided by some scaling. The algorithm works for vectors and matrices as well. Concerning Algorithm 4.2, we proved in [14] the following.

Theorem 4.4. *Let* $\mathtt{A}\in\mathbb{F}^{m\times k}$ *and* $\mathtt{B}\in\mathbb{F}^{k\times n}$ *with* $(k+2)\mathbf{u}\le 1$ *be given, and let* `res` *and* `err` *be the quantities computed by Algorithm 4.2 (*`DotErr`*). Then, also in the presence of underflow,*

$$|\mathtt{A}\cdot\mathtt{B}-\mathtt{res}|\le\mathtt{err}. \tag{4.6}$$

The factor $k+2$ *in Algorithm 4.2 cannot be replaced by* $k+1$.

The ufp-concept proved to be very useful in verifying the validity of floating-point estimates, and to obtain sharp estimates. I introduced this concept in [21] to prove the delicate estimations in there. Among the many properties of the unit in the first place (cf. [21]) we only need the following. For $a, b\in\mathbb{F}$ and $\circ\in\{+,-,\cdot,/\}$, as in (1.2), $\mathrm{fl}(a\circ b)$ is the result of the floating-point approximation of $a\circ b$. Then the standard estimate of the error of $\mathrm{fl}(a\circ b)$ is improved into

$$f=\mathrm{fl}(a\circ b)\Rightarrow f=a\circ b+\delta\quad\text{with }|\delta|\le\mathbf{u}\cdot\mathrm{ufp}(a\circ b)\le\mathbf{u}\cdot\mathrm{ufp}(f)\le\mathbf{u}|f| \tag{4.7}$$

for $\circ\in\{+,-\}$. If $\circ\in\{\cdot,/\}$ and $a\circ b$ is not in the underflow range, then (4.7) is true as well. If $\circ\in\{\cdot,/\}$ and $a\circ b$ is in the underflow range, then $|\delta|\le\frac{1}{2}\mathtt{eta}$, where `eta` denotes the smallest positive unnormalized floating-point number. Using $\mathrm{ufp}(x)\le|x|$ for $x\in\mathbb{R}$ this implies, for example,

$$a\circ b=(1+\varepsilon_1)\cdot\mathrm{fl}(a\circ b)=\mathrm{fl}(a\circ b)/(1+\varepsilon_2)\quad\text{with }|\varepsilon_\nu|\le\mathbf{u},\ 1\le\nu\le 2 \tag{4.8}$$

if $\circ\in\{+,-\}$, or if $\circ\in\{\cdot,/\}$ and $a\circ b$ is not in the underflow range.

This means that the error bound (4.6) computed by Algorithm 4.2 (`DotErr`) is not only rigorous and simple and valid if underflow occurs, but using the unit in the first place (ufp) it may also be sharper than the classical Wilkinson-estimate (4.4) by up to a factor 2. For numerical evidence we compared in [14] the error estimate by `DotErr` and (4.4) for a matrix product `R*A`, where `A` is randomly generated with specified condition number and `R=inv(A)`. For dimensions from 10 to 1000 and condition numbers from 1 to $\mathbf{u}^{-1}$, the value of the bound by `DotErr` is uniformly about 0.7 times the classical Wilkinson-estimate (4.4).

For the sum of floating-point numbers rigorous bounds are computed by the following Algorithm 4.5.

Algorithm 4.5. Rigorous error bound $\left|\sum_{i=1}^{n}\mathtt{p(i)}-\mathtt{res}\right|\le\mathtt{err}$, also in the presence of underflow.

```
function [res,err] = SumErr(p)
  n = length(p);                            % number of summands
  res = sum(p);                             % approximation of sum
  D = sum(abs(p));                          % sum of absolute values
  U = ufp(D);                               % unit in the first place of D
  err = (n-1)*(eps/2*U);                    % error bound for d
```

Theorem 4.6 ([14]). *Let $p \in \mathbb{F}^n$ with $n\mathbf{u} \le 1$ be given, and let* res *and* err *be the quantities computed by Algorithm 4.5 (*SumErr*). Then, also in the presence of underflow,*

$$\left| \sum_{i=1}^{n} \mathtt{p(i)} - \mathtt{res} \right| \le \mathtt{err}. \tag{4.9}$$

The estimation is sharp for all $n \le \mathbf{u}^{-1}$.

Underflow is no problem for summation since a floating-point sum with a result in the underflow range is exact. For later usage we note that for U as computed in SumErr

$$\left| \sum_{i=1}^{n} \mathtt{p(i)} - \mathtt{res} \right| \le (n-1) \cdot \mathbf{u} \cdot \mathtt{U} \tag{4.10}$$

is always true, also for $n > \mathbf{u}^{-1}$. If U is in the underflow range, the right hand side can be replaced by zero. Besides, we need an upper bound for the sum of nonnegative floating-point numbers as computed by the following algorithm.

Algorithm 4.7. Rigorous error bound $\sum_{i=1}^{n} \mathtt{p(i)} \le \mathtt{ubnd}$, also in the presence of underflow, for nonnegative $\mathtt{p} \in \mathbb{F}^n$.

```
function ubnd = SumPosBnd(p)
  n = length(p);                          % number of summands
  S = sum(p);                             % approximation of sum
  ubnd = S + (n+1)*(0.5*eps*ufp(S));      % upper bound for sum
```

Theorem 4.8. *Let nonnegative $\mathtt{p} \in \mathbb{F}^n$ with $(n+1)\mathbf{u} \le 1$ be given, and let* ubnd *be the quantity computed by Algorithm 4.7 (*SumPosBnd*). Then, also in the presence of underflow,*

$$\sum_{i=1}^{n} \mathtt{p(i)} \le \mathtt{ubnd}. \tag{4.11}$$

Proof. If $\mathtt{U} := \mathrm{ufp}(\mathtt{S})$ is in the underflow range, then, because the summands are nonnegative, all intermediate sums are in the underflow range and no error occurs at all. If U is not in the underflow range, then there is no error in the floating-point computation of $\mathtt{delta} := (\mathtt{n}+1) * (0.5 * \mathtt{eps} * \mathrm{ufp}(\mathtt{S}))$ because U is a power of 2, $\mathtt{eps}/2 = \mathbf{u}$ and $(n+1)\mathbf{u} \le 1$. Hence $\mathtt{delta} = (\mathtt{n}+1) \cdot \mathbf{u} \cdot \mathtt{U} \le \mathtt{U} \le \mathtt{S}$ and $\mathrm{ufp}(\mathtt{S} + \mathtt{delta}) \le \mathrm{ufp}(2\mathtt{S}) = 2\mathrm{ufp}(\mathtt{S}) = 2\mathtt{U}$, so that (4.7) implies

$$\mathtt{ubnd} \ge \mathtt{S} + \mathtt{delta} - \mathbf{u} \cdot \mathrm{ufp}(\mathtt{S} + \mathtt{delta}) \ge \mathtt{S} + (n-1) \cdot \mathbf{u} \cdot \mathtt{U} \ge \sum_{i=1}^{n} \mathtt{p(i)}$$

by (4.10) and the nonnegativity of the summands. □

Before we come to the computation of a rigorous upper bound of the right hand side in (4.3) in rounding to nearest, we have to prove the correctness of the error bound of Algorithm 3.4 (Dot2Near). We need the following auxiliary lemma.

Lemma 4.9. *Let $a, b \in \mathbb{F}$ be given, and let* s *and* sabs *be computed by the following Matlab commands:*

```
s = a+b;                          % floating-point approximation
sigma1 = 1+eps;                   % successor of 1
sabs = sigma1*abs(s);             % upper bound for |a+b|
```

Then, also in the presence of underflow,

$$|\mathtt{a} + \mathtt{b}| \le \mathtt{sabs}. \tag{4.12}$$

Proof. If s is in the underflow range, then $\mathtt{s} = \mathtt{a} + \mathtt{b}$ and the result follows. Otherwise $|\mathtt{a} + \mathtt{b}| \le |\mathtt{s}| + \mathbf{u} \cdot \mathrm{ufp}(\mathtt{s}) < |\mathtt{s}| + 2\mathbf{u} \cdot \mathrm{ufp}(\mathtt{s})$ by (4.7). But $|\mathtt{s}| + 2\mathbf{u} \cdot \mathrm{ufp}(\mathtt{s})$ is the successor of s and therefore a floating-point number, so that $|\mathtt{s}| + 2\mathbf{u} \cdot \mathrm{ufp}(\mathtt{s}) \le (1 + 2\mathbf{u})|\mathtt{s}| = \mathtt{sigma1} \cdot |\mathtt{s}|$ and the monotonicity of the rounding proves the result. □

4.2. Correctness of the error bound of Algorithm 3.4 (Dot2Near)

For the proof of correctness of the error term, we need the following error estimation of recursive floating-point summation. As in (1.2) denote by $\mathrm{fl} : \mathbb{R} \to \mathbb{F}$ the rounding such that $\mathrm{fl}(a \circ b) \in \mathbb{F}$ is nearest to $a \circ b \in \mathbb{R}$ for $a, b \in \mathbb{F}$ and $\circ \in \{+, -, \cdot, /\}$. This is the definition in the IEEE 754 floating-point standard. For a given vector $p \in \mathbb{F}^n$ of floating-point numbers, let $\widetilde{s}_n$ and $\widetilde{S}_n$ be computed by the following algorithm:

Algorithm 4.10. Recursive summation with error estimation.

$$\widetilde{s}_1 = p_1; \qquad \widetilde{S}_1 = |p_1|$$
$$\text{for } k = 2 : n$$
$$s_k = \widetilde{s}_{k-1} + p_k; \qquad \widetilde{s}_k = \mathrm{fl}(s_k)$$
$$S_k = \widetilde{S}_{k-1} + |p_k|; \qquad \widetilde{S}_k = \mathrm{fl}(S_k).$$

Then (4.7) implies the following rigorous error estimate, also in the presence of underflow:

$$\left|\widetilde{s}_n - \sum_{i=1}^{n} p_i\right| \le (n-1)\mathbf{u}\cdot \mathrm{ufp}(\widetilde{S}_n) \quad \left[\le (n-1)\mathbf{u}\widetilde{S}_n\right]. \tag{4.13}$$

To analyze Algorithm 3.4 (`Dot2Near`), we first rephrase it to distinguish intermediate results in the loop and to identify the true and the rounded result of the intermediate operations. It suffices to analyze a single dot product $x^T y$ for $x, y \in \mathbb{F}^n$ with $(n+2)\mathbf{u} \le 1$. Then, using (3.1), Algorithm 3.4 is equivalent to the following:

$$p_1 + \widetilde{e}_1 = x_1 \cdot y_1 + \eta_1; \qquad E_1 = |\widetilde{e}_1|$$
$$2 \le i \le n \begin{cases} h_i + r_i = x_i \cdot y_i + \eta_i \\ p_i + q_i = p_{i-1} + h_i \\ t_i = q_i + r_i; \qquad \widetilde{t}_i = \mathrm{fl}(t_i) \\ e_i = \widetilde{e}_{i-1} + \widetilde{t}_i; \qquad \widetilde{e}_i = \mathrm{fl}(e_i) \\ E_i = E_{i-1} + |\widetilde{t}_i|; \qquad \widetilde{E}_i = \mathrm{fl}(E_i) \end{cases}$$
$$\rho = p_n + \widetilde{e}_n; \qquad \mathtt{res} = \mathrm{fl}(\rho).$$

It follows

$$x^T y + \sum_{i=1}^{n} \eta_i = p_1 + \widetilde{e}_1 + \sum_{i=2}^{n}(h_i + r_i) = p_n + \widetilde{e}_1 + \sum_{i=2}^{n}(q_i + r_i)$$
$$= p_n + \widetilde{e}_1 + \sum_{i=2}^{n}\widetilde{t}_i + \sum_{i=2}^{n}(t_i - \widetilde{t}_i). \tag{4.14}$$

Now $\widetilde{e}_n$ is the floating-point sum of $\widetilde{e}_1 + \sum_{i=2}^{n}\widetilde{t}_i$ and $\widetilde{E}_n$ is the floating-point sum of the absolute values, so (4.10) gives

$$\left|\widetilde{e}_1 + \sum_{i=2}^{n}\widetilde{t}_i - \widetilde{e}_n\right| \le (n-1)\mathbf{u}\cdot \mathrm{ufp}(\widetilde{E}_n). \tag{4.15}$$

Furthermore, $|t_i - \widetilde{t}_i| \le \mathbf{u}|\widetilde{t}_i|$, so again (4.10) and $(n+2)\mathbf{u} \le 1$ imply

$$\sum_{i=2}^{n}|t_i - \widetilde{t}_i| \le \mathbf{u}\cdot\sum_{i=2}^{n}|\widetilde{t}_i| \le \mathbf{u}\cdot\left[\widetilde{E}_n + (n-2)\mathbf{u}\cdot\mathrm{ufp}(\widetilde{E}_n)\right] \le 3\mathbf{u}\cdot\mathrm{ufp}(\widetilde{E}_n). \tag{4.16}$$

Combining (4.14)–(4.16) with (3.1) yields

$$|x^T y - (p_n + \widetilde{e}_n)| \le 3n\mathtt{eta} + (n+2)\mathbf{u}\cdot\mathrm{ufp}(\widetilde{E}_n). \tag{4.17}$$

Now $|\mathtt{res} - (p_n + \widetilde{e}_n)| = |\mathrm{fl}(p_n + \widetilde{e}_n) - (p_n + \widetilde{e}_n)| \le \mathbf{u}\cdot\mathrm{ufp}(\mathtt{res})$ and $\mathtt{realmin} = \frac{1}{2}\mathbf{u}^{-1}\mathtt{eta}$ give

$$|x^T y - \mathtt{res}| \le \mathbf{u}\cdot\mathrm{ufp}(\mathtt{res}) + \max(6n\mathbf{u}, 1)\cdot\mathtt{realmin} + (n+2)\mathbf{u}\cdot\mathrm{ufp}(\widetilde{E}_n). \tag{4.18}$$

If `res` is in the underflow range, then there is no rounding error in the addition $p_n + \widetilde{e}_n$, so that $\mathtt{res} = p_n + \widetilde{e}_n$, and $\mathbf{u}\cdot\mathrm{ufp}(\mathtt{res})$ in (4.18) can be omitted. If $\widetilde{E}_n$ as a sum of nonnegative numbers is in the underflow range, then all $\widetilde{t}_i$ are in the underflow range, and there is no error in the sums $q_i + r_i$. Hence $t_i = \widetilde{t}_i$ for $2 \le i \le n$, the right hand side of (4.16) can be replaced by zero so that $(n+2)\mathbf{u}\cdot\mathrm{ufp}(\widetilde{E}_n)$ in (4.18) can be omitted.

If neither `res` nor $\widetilde{E}_n$ is in the underflow range, then the floating-point computation of the three summands in (4.18) does not cause a rounding error, so that in any case only the rounding errors in the two additions in the right hand side of (4.18) have to be taken care of. Again applying (4.10) to this floating-point sum of three non-negative summands proves[3]

$$|x^T y - \mathtt{res}| \le \mathtt{err0} + 2\mathbf{u}\cdot\mathrm{ufp}(\mathtt{err0}) \le \mathrm{fl}\big(\mathtt{err0} + 3\mathbf{u}\cdot\mathrm{ufp}(\mathtt{err0})\big) = \mathtt{err}. \tag{4.19}$$

This proves the error estimate (3.3) in Theorem 3.5.

The presented algorithms for computing error bounds are suitable for a compiled programming language such as C of Fortran or a Matlab mex-file; a pure Matlab implementation suffers significantly from interpretation overhead.

[3] The Matlab constant `eps` is $2\mathbf{u}$, so `epss` $= 0.5 *$ `eps` is used in Algorithm 3.4.

4.3. Rigorous error bounds for linear systems in rounding to nearest (up to $\mathrm{cond}(A) \lesssim \mathbf{u}^{-1}$*)*

In the remaining of this subsection we will prove that the quantity `err` computed by the following Algorithm 4.11 (`LssErrBndNear0`) is an upper bound of the right hand side in (4.3) in Theorem 4.1 and thus an error bound for the approximate solution `xs`. For didactical reasons we first state this preliminary version of the final Algorithm 4.16 (`LssErrBndNear`).

Algorithm 4.11. Rigorous error bound for the solution of a linear system $Ax = b$, preliminary version.

```
function [xs,err] = LssErrBndNear0(A,b)
  err = NaN(size(b));                          % initialize result
  R = inv(A);                                  % approximate inverse of A
  xs = ResidIter(A,b,R);                       % approximate solution
  n = size(A,2);                               % dimension of the linear system
  [D,eD] = Dot2Near([A b],[xs;-1]);            % error bound D+/-eD for residual
  [aRD,eRD] = DotErr(R,D);                     % error bound aRD+/-eRD of R*D
  [aReD,eReD] = DotErr(abs(R),eD);             % error bound aReD+/-eReD of |R|*eD
  dd = abs(aRD) + eRD + aReD + eReD;           % not yet upper bound of |R*(A*xs-b)|
  delta = dd + 2.5*(eps*ufp(dd));              % upper bound of |R*(A*xs-b)|
  [aRA,eRA] = DotErr(R,A);                     % bounds for R*A
  RA_I = (1+eps)*abs(aRA-eye(n));              % upper bound of |aRA-I|
  E = (1+eps)*(RA_I+eRA);                      % upper bound of |R*A-I|
  aE1 = sum(E,2);                              % approximation of |R*A-I|*ones(n,1)
  uE1 = aE1 + (n+1)*(0.5*eps*ufp(aE1));        % upper bound of |R*A-I|*ones(n,1)
  Den = (1-max(uE1)) - 1.5*eps;                % lower bound of 1-||E||_inf
  if Den>0                                     % algorithm successful
    err0 = (max(delta)/Den)*uE1 + realmin;     % almost final error bound
    err = (1+eps)*(delta+err0);                % final error bound
  end
```

The command `Dot2Near([A b],[xs;-1])` in line 5 can obviously be replaced by a specialized algorithm `Dot2Near` taking into account the special structure. For ease of exhibition we refrain from that in this article.

In all of the following analysis we use the "verbatim"-font for the computed quantities, and all operations in the following analysis are the *exact, real* operations. For example, $\mathtt{R}\cdot\left(\mathtt{A}\cdot\mathtt{xs}-\mathtt{b}\right) \in \mathbb{R}^n$ is the *true* correction of $\mathtt{xs} \in \mathbb{F}^n$. Note that taking the maximum and the absolute value of a vector does not cause any rounding error, so we may use $\max(\mathtt{x})$ or $\mathtt{max(x)}$ without causing any ambiguousness, and similarly for the absolute value.

For the moment assume that the command `xs = ResidIter(A,b,R)` in line 3 computes some vector $\mathtt{xs} \in \mathbb{F}^n$ of floating-point numbers. Of course, a good approximation to $A^{-1}b$ is preferable, but for the proof of correctness of the bound this is irrelevant. Also assume $(n+2)\mathbf{u} \le 1$. Then (3.3) in Theorem 3.5 implies

$$|\mathtt{D} - (\mathtt{A}\cdot\mathtt{xs} - \mathtt{b})| \le \mathtt{eD}, \tag{4.20}$$

and (4.6) in Theorem 4.4 gives

$$|\mathtt{aRD} - \mathtt{R}\cdot\mathtt{D}| \le \mathtt{eRD} \quad \text{and} \quad |\mathtt{aReD} - |\mathtt{R}|\cdot\mathtt{eD}| \le \mathtt{eReD}. \tag{4.21}$$

Define $\delta := \mathtt{R}\cdot\left(\mathtt{A}\cdot\mathtt{xs}-\mathtt{b}\right)$, observe that `dd` is the sum of four nonnegative summands and that the computation of `delta` implements Algorithm 4.7 (`SumPosBnd`) for those four summands. Hence Theorem 4.8 is applicable and implies

$$|\delta| = |\mathtt{R}\cdot\left(\mathtt{A}\cdot\mathtt{xs}-\mathtt{b}\right)| \le |\mathtt{R}\cdot\mathtt{D}| + |\mathtt{R}|\cdot\mathtt{eD} \le |\mathtt{aRD}| + \mathtt{eRD} + \mathtt{aReD} + \mathtt{eReD} \le \mathtt{delta}. \tag{4.22}$$

Next

$$|\mathtt{aRA} - \mathtt{R}\cdot\mathtt{A}| \le \mathtt{eRA}, \tag{4.23}$$

and Lemma 4.9 implies

$$|\mathtt{aRA} - I| \le \mathtt{RA\_I} \quad \text{and} \quad \mathtt{RA\_I} + \mathtt{eRA} \le \mathtt{E}.$$

Putting things together yields

$$|I - \mathtt{R}\cdot\mathtt{A}| \le |I - \mathtt{aRA}| + |\mathtt{aRA} - \mathtt{R}\cdot\mathtt{A}| \le \mathtt{E}. \tag{4.24}$$

Again using the code in Algorithm 4.7 (`SumPosBnd`) and Theorem 4.8 gives

$$|I - \mathtt{R}\cdot\mathtt{A}|\cdot e \le \mathtt{E}\cdot e \le \mathtt{uE1} \quad \text{and} \quad \|I - \mathtt{R}\cdot\mathtt{A}\|_\infty \le \mathtt{max(uE1)}. \tag{4.25}$$

To continue we first split the computation of Den into two parts by the Matlab statements

$$\texttt{Den0} = 1 - \max(\texttt{uE1}); \qquad \texttt{Den} = \texttt{Den0} - 1.5 * \texttt{eps};. \tag{4.26}$$

Note that $\texttt{eps} = 2\mathbf{u}$. If Algorithm 4.11 (LssErrBndNear0) finishes successfully, then $1 \geq \texttt{Den0} \geq \texttt{Den} > 0$, so that

$$\|I - \texttt{R} \cdot \texttt{A}\|_\infty \leq \max(\texttt{uE1}) < 1 \tag{4.27}$$

and Theorem 4.1 is applicable. Abbreviate $\texttt{NE} := \max(\texttt{uE1}) \geq \|I - \texttt{R} \cdot \texttt{A}\|_\infty$ and suppose for the moment $\texttt{NE} > 0$. Then $\operatorname{ufp}(1 - \texttt{NE}) \leq \frac{1}{2}$ and (4.7) give

$$\texttt{Den0} \leq 1 - \texttt{NE} + \mathbf{u} \cdot \operatorname{ufp}(1 - \texttt{NE}) \leq 1 - \texttt{NE} + \frac{1}{2}\mathbf{u}, \tag{4.28}$$

so that again by (4.7) and $\texttt{eps} = 2\mathbf{u}$

$$\texttt{Den} \leq \texttt{Den0} - 3\mathbf{u} + \mathbf{u} \cdot \operatorname{ufp}(\texttt{Den0} - 3\mathbf{u}) \leq 1 - \texttt{NE} + \frac{1}{2}\mathbf{u} - 3\mathbf{u} + \frac{1}{2}\mathbf{u} = 1 - \texttt{NE} - 2\mathbf{u}. \tag{4.29}$$

Consider the Matlab statements

$$\texttt{M} = \max(\texttt{delta}); \qquad \texttt{F1} = \texttt{M}/\texttt{Den}; \qquad \texttt{F2} = \texttt{F1} * \texttt{uE1}; \qquad \texttt{err0} = \texttt{F2} + \texttt{realmin};. \tag{4.30}$$

Note that (4.22) implies $\|\texttt{R} \cdot (\texttt{A} \cdot \texttt{xs} - \texttt{b})\|_\infty \leq \texttt{M}$ and that the computations of err0 in (4.30) and in Algorithm 4.11 (LssErrBndNear0) are identical. The computation of delta and Algorithm 4.2 (DotErr) imply $\texttt{delta} \geq \texttt{eRD} \geq \texttt{realmin}$, and therefore $\texttt{M}/\texttt{Den} \geq \texttt{realmin}$ because $\texttt{Den} < 1$. Hence the quotient F1 is not in the underflow range, and (4.8) implies

$$\texttt{F1} \geq (1 - \mathbf{u})\frac{\texttt{M}}{\texttt{Den}} \geq \frac{(1 - \mathbf{u})\texttt{M}}{1 - \texttt{NE} - 2\mathbf{u}} \geq \frac{(1 + \mathbf{u})\texttt{M}}{1 - \texttt{NE}} \tag{4.31}$$

with a little computation for the last inequality. The product F2 may be in the underflow range. If so, then

$$\texttt{err0} \geq \texttt{realmin} \geq \texttt{F1} \cdot \texttt{uE1},$$

and otherwise by (4.8),

$$\texttt{err0} \geq \texttt{F2} \geq \frac{\texttt{F1} \cdot \texttt{uE1}}{1 + \mathbf{u}}.$$

In any case (4.31) and (4.25) imply

$$\texttt{err0} \geq \frac{\texttt{F1} \cdot \texttt{uE1}}{1 + \mathbf{u}} \geq \frac{\texttt{M} \cdot \texttt{uE1}}{1 - \texttt{NE}} \geq \frac{\|\delta\|_\infty}{1 - \|I - \texttt{R} \cdot \texttt{A}\|_\infty} \cdot |I - \texttt{R} \cdot \texttt{A}| \cdot e. \tag{4.32}$$

Finally Lemma 4.9, (4.22) and (4.32) yield

$$\texttt{err} \geq \texttt{delta} + \texttt{err0} \geq |\delta| + \frac{\|\delta\|_\infty}{1 - \|I - \texttt{R} \cdot \texttt{A}\|_\infty} \cdot |I - \texttt{R} \cdot \texttt{A}| \cdot e. \tag{4.33}$$

This finishes the proof for the case $\texttt{NE} > 0$. If $\texttt{NE} = 0$, then $\texttt{NE} = \max(\texttt{uE1}) \geq \|I - \texttt{R} \cdot \texttt{A}\|_\infty$ implies that $\texttt{R} = \texttt{A}^{-1}$, so that the right hand side in (4.3) reduces to $|\delta|$. Observing $\texttt{err} \geq |\delta|$ proves the following theorem.

Theorem 4.12. *Let a set $\mathbb{F}$ of floating-point numbers with relative rounding error unit $\mathbf{u}$ together with floating-point operations complying with the IEEE 754 arithmetic standard [10,11] be given.*

Let res *and* err *be the results of Algorithm 4.11 (*LssErrBndNear0*) applied to a matrix $\texttt{A} \in \mathbb{F}^{n \times n}$ and a vector $\texttt{b} \in \mathbb{F}^n$ with $(n + 2)\mathbf{u} \leq 1$. If the algorithm ends successfully, then* A *is non-singular and*

$$|\texttt{xs} - A^{-1}b| \leq \texttt{err}. \tag{4.34}$$

It remains to explain xs=ResidIter(A,b,R) in line 3 of Algorithm 4.11 (LssErrBndNear0). The simplest is to perform just one residual iteration in working precision to ensure backward stability [2] of the computed xs by the Matlab statements

$$\texttt{xs} = \texttt{R} * \texttt{b}; \qquad \texttt{xs} = \texttt{xs} - \texttt{R} * (\texttt{A} * \texttt{xs} - \texttt{b}); \tag{4.35}$$

and also to use DotErr([A b],[xs;-1]) rather than Dot2Near([A b],[xs;-1]) two lines later. In that case no extra-precise dot products are used at all and the quality of the inclusion is of the order $\mathbf{u} \cdot \operatorname{cond}(\texttt{A})$. A much better result and often inclusions of almost maximum accuracy, also for ill-conditioned matrices, is obtained by a residual iteration using Dot2Near. The stopping criterion implements numerical experience heuristically.

Table 4.1
Ratio of measured computing times between Algorithm 4.11 (LssErrBndNear0) and the Matlab command A \ b. Rigorous inclusions by LssErrBndNear0 are computed using the Matlab implementation of Algorithm 3.4 (Dot2Near) and a C-implementation using mex-files.

	$n = 100$	$n = 200$	$n = 500$	$n = 1000$
Matlab Dot2Near	35.8	19.7	11.8	9.2
C-program Dot2Near	3.5	4.3	7.5	7.3

Algorithm 4.13. Improvement of xs by extra-precise residual iteration.

```
function xs = ResidIter(A,b,R)
  xs = R*b;                                  % first approximate solution
  normxs = norm(xs,inf); N = inf;            % initialization of constants
  for iter=1:10                              % at most 10 residual iterations
    Nold = N;                                % update constants
    d = R*Dot2([A b],[xs;-1]);               % correction for xs
    N = norm(d,inf);                         % norm of correction
    if N<Nold, xs = xs-d; end                % correction acceptable
    if ( ( iter==1 ) && ( N<1e-9*normxs ) ) || ( N<eps*normxs ) || ( N>=0.3*Nold )
      break                                  % stop iteration if well-conditioned
    end                                      %    or no improvement
  end
```

As has been mentioned, the implementation of Dot2Near in Matlab suffers severely from interpretation overhead. Nevertheless the main computational effort in Algorithm 4.11 (LssErrBndNear0) is the computation of the approximate inverse R = inv(A) and the bounds for the residual I − R ∗ A, the latter requiring two matrix multiplications R ∗ A and abs(R) ∗ abs(A) in Algorithm 4.2 (DotErr). Therefore the theoretical time ratio between LssErrBndNear0 and the Gaussian elimination by the Matlab command xs = A \ b is 9.

In practice the ratio is better because matrix multiplication performs usually better than Gaussian elimination. Table 4.1 shows the ratio between the computing times of Algorithm 4.11 (LssErrBndNear0) and the Matlab command xs = A \ b for different dimensions, the former first with the Matlab implementation of Algorithm 3.4 (Dot2Near), and second using a C-program and mex-file for Dot2Near.[4] The matrix and right hand side are chosen randomly. For ill-conditioned matrices, where more residual iterations are performed, the ratio increases because Matlab uses no residual iteration, apparently even not a single one in working precision.

As can be seen, using the Matlab implementation the ratio decreases because of the decreasing interpretation overhead; using the C-program there is an increase, possibly due to a (relatively) better performance of the Matlab command A \ b and because the C-program is written straightforwardly without blocking.

Needless to say that Table 4.1 compares apples with oranges because Algorithm 4.11 (LssErrBndNear0) computes rigorous error bounds which are almost accurate to working precision, whereas the Matlab command A \ b delivers only an approximation supposedly of quality $\mathbf{u} \cdot \mathrm{cond}(A)$.

Replacing the computation [aRA,eRA]=DotErr(R,A) of the bounds for R · A by [aRA,eRA]=Dot2Near(R,A) extends the range of applicability of Algorithm 4.11 (LssErrBndNear0), i.e. error bounds are computed for more ill-conditioned matrices. Moreover, the quality of the error bound improves for $\mathrm{cond}(A) \lesssim \mathbf{u}^{-1}$ at the price of increasing computing time. This will be shown in the next subsection.

4.4. Improved bounds for ill-conditioned linear systems

The successful computation of an error bound by Algorithm 4.11 (LssErrBndNear0) depends on $\|I - RA\|_\infty < 1$. For ill-conditioned A, i.e. $\mathrm{cond}(A) \approx \mathbf{u}^{-1}$, it is not uncommon that $\|I - RA\|_\infty \geq 1$ for a computed approximate inverse R, but some diagonal scaling rescues the situation by $\|D^{-1}(I - RA)D\|_\infty < 1$. Since $\|I - RA\|_\infty = \| \, |I - RA| \, \|_\infty$, the Perron vector of $|I - RA|$ is the best choice.[5]

In Fig. 4.1 the percentage of cases where $\|D^{-1}(I - RA)D\|_\infty < 1$ but $\|I - RA\|_\infty \geq 1$ using $D := diag(u)$ for u denoting the Perron vector of $|I - RA|$ for different dimensions and condition numbers is displayed. It shows that $\|I - RA\|_\infty \geq 1$ begins to happen for condition numbers roughly starting with $\mathbf{u}^{-1}/n$, thus matching theory, whereas $\|D^{-1}(I - RA)D\|_\infty < 1$ is still true for a little larger condition numbers.

[4] Many thanks to Prof. Ogita from Tokyo Woman's Christian University for providing the C- and mex-programs

[5] In the rare case that $|I - RA|$ is not irreducible, the Perron vector u of $|I - RA| + \epsilon$ for small $\epsilon > 0$ may be used to ensure $u > 0$, so that $D = diag(u)$ is non-singular.

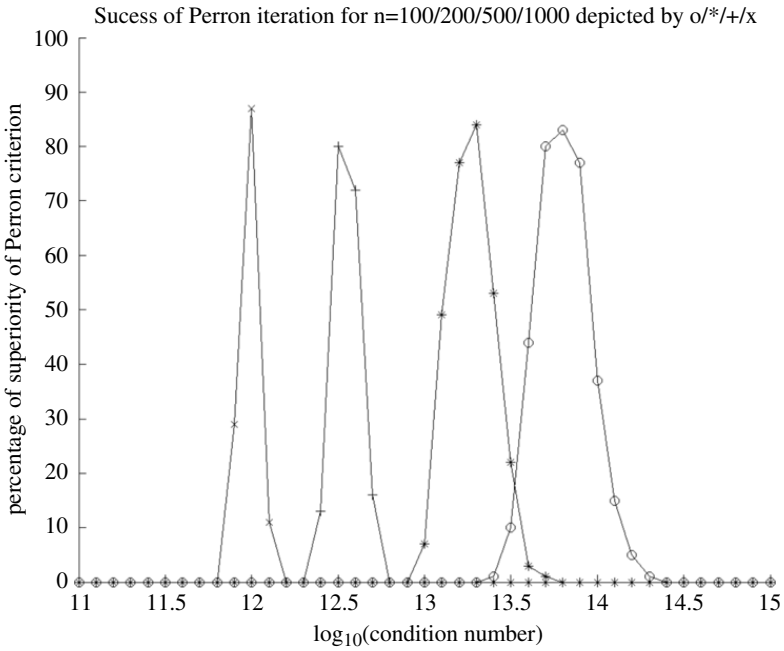

Fig. 4.1. Percentage that $\|D^{-1}(I - RA)D\|_\infty < 1$ but $\|I - RA\|_\infty \geq 1$ using $D := diag(u)$ for u denoting the Perron vector of $|I - RA|$ for dimensions $n = 100$ (o), $n = 200$ ($*$), $n = 500$ ($+$) and $n = 1000$ ($\times$) and condition numbers ranging from 10^{13} to 10^{15}.

For corresponding improved error bounds consider the following refinement of Theorem 4.1. It approaches the theoretically necessary condition that the spectral radius of $|I - RA|$ is strictly less than one and shows the range of applicability of our approach.

Theorem 4.14. *Let $A, R \in \mathbb{R}^{n\times n}$ and $b, \widetilde{x}, u \in \mathbb{R}^n$ be given. Assume $u > 0$, and let $D \in \mathbb{R}^{n\times n}$ be the diagonal matrix with u on the diagonal. Define $E := I - RA$ and $\delta := R(A\widetilde{x} - b)$, and assume $\|D^{-1}|E|u\|_\infty < 1$. Then A is non-singular and*

$$|\widetilde{x} - A^{-1}b| \leq |\delta| + \frac{\|D^{-1}\delta\|_\infty}{1 - \|D^{-1}|E|u\|_\infty} \cdot |E|u. \tag{4.36}$$

Proof. Define $\hat{A} := AD$, $\hat{R} := D^{-1}R$, and $\hat{x} := D^{-1}\widetilde{x}$. Then

$$\hat{\delta} := \hat{R}(\hat{A}\hat{x} - b) = D^{-1}\delta \quad \text{and} \quad \hat{E} := I - \hat{R}\hat{A} = D^{-1}ED.$$

Therefore (4.3) in Theorem 4.1 implies

$$|\widetilde{x} - A^{-1}b| = D|\hat{x} - \hat{A}^{-1}b| \leq D|\hat{\delta}| + \frac{\|\hat{\delta}\|_\infty}{1 - \|\hat{E}\|_\infty} \cdot D|\hat{E}|e,$$

so that $De = u$ and $\|\hat{E}\|_\infty = \||\hat{E}|e\|_\infty$ finishes the proof. □

Next we show how this is used to compute rigorous bounds for linear systems in rounding to nearest. Suppose that some positive vector `u` $\in \mathbb{F}^n$ is given and that the quantities `delta` and `E` in Algorithm 4.11 (`LssErrBndNear0`) have been computed. Then consider the following Matlab commands.

```
Dd = delta./u;                                  % approximation of D^-1*delta
uDd = max((1+eps)*Dd) + realmin;                % upper bound of ||D^-1*delta||_inf
aEu = E*u;                                      % approximation of |I-RA|u
e = (n+2)*(eps/2*ufp(aEu)) + 1.5*realmin;       % bound |I-RA|u <= aEu+e
uEu = (1+eps)*(aEu+e);                          % upper bound of |I-RA|u
N0 = max(uEu./u);                               % approximation of ||D^-1|I-RA|u||_inf
N1 = (1+eps)*N0;                                % not yet bound of ||D^-1|I-RA|u||_inf
N2 = 1 - N1;                                    % approximation of 1-||D^-1|I-RA|u||_inf
Den = N2 - 1.5*eps;                             % suitable denominator
if Den>0                                        % algorithm successful
  N3 = uDd./Den;
  err0 = N3*uEu + realmin;                      % upper bound of second term
  err = (1+eps)*(delta+err0);                   % final error term
end
```

To analyze the code we first need a version of Lemma 4.9 for multiplication and division.

Lemma 4.15. *Let $a, b \in \mathbb{F}$ be given, and let* `r` *and* `rabs` *be computed by the following Matlab commands:*

```
r = a*b;                              % floating-point approximation
rabs = (1+eps)*abs(r) + realmin;      % upper bound for |a*b|
```

Then, also in the presence of underflow,

$$|\mathtt{a} \cdot \mathtt{b}| \le \mathtt{rabs}. \tag{4.37}$$

The statement is also true when replacing multiplication by division in the first line of the Matlab code and in (4.37).

Proof. In case of underflow, $|\mathtt{a} \cdot \mathtt{b}|$ is bounded by $\frac{1}{2}\mathtt{eta} < \mathtt{realmin}$, and otherwise the result follows as in the proof of Lemma 4.9 because $\mathtt{eps} = 2\mathbf{u}$ implies that `1+eps` is the successor of 1. □

First note that (4.22) and Lemma 4.15 imply for positive `u`

$$\|D^{-1}\delta\|_\infty = \max_i \frac{\delta_i}{u_i} \le \max_i \frac{\mathtt{delta(i)}}{\mathtt{u(i)}} \le \mathtt{uDd}. \tag{4.38}$$

Furthermore, (4.24), Theorem 4.4 and Lemma 4.9 imply

$$|I - \mathtt{R} \cdot \mathtt{A}| \cdot \mathtt{u} \le \mathtt{E} \cdot \mathtt{u} \le \mathtt{aEu} + \mathtt{e} \le \mathtt{uEu}. \tag{4.39}$$

Assume for the moment that $\max(\mathtt{N0}) \ne 0$, and that in the Matlab statement `N0 = max(uEu./u)` no underflow occurs. Then also $\mathtt{N1} \ne 0$, and again using Lemma 4.15 yields

$$\mu := \|D^{-1}|I - \mathtt{R} \cdot \mathtt{A}| \cdot \mathtt{u}\|_\infty \le \mathtt{N1}. \tag{4.40}$$

Assume an error bound is computed, i.e. $\mathtt{Den} > 0$. Then, as in the proof of Theorem 4.12, $1 > \mathtt{N2} > \mathtt{Den} > 0$ so that $\max\{\mathrm{ufp}(\mathtt{Den}), \mathrm{ufp}(1 - \mathtt{N1})\} \le \frac{1}{2}$, $\mathtt{eps} = 2\mathbf{u}$, (4.7) and (4.40) show

$$\mathtt{Den} \le \mathtt{N2} - 3\mathbf{u} + \mathbf{u} \cdot \mathrm{ufp}(\mathtt{Den}) \le 1 - \mathtt{N1} + \mathbf{u} \cdot \mathrm{ufp}(1 - \mathtt{N1}) - \frac{5}{2}\mathbf{u} \le 1 - \mu - 2\mathbf{u}, \tag{4.41}$$

and $\mathtt{uDd} \ge \mathtt{realmin}$ and (4.7) yield

$$\mathtt{N3} \ge (1 - \mathbf{u})\frac{\mathtt{uDd}}{\mathtt{Den}} \ge (1 - \mathbf{u})\frac{\mathtt{uDd}}{1 - \mu - 2\mathbf{u}} \ge (1 + \mathbf{u}) \cdot \frac{\mathtt{uDd}}{1 - \mu}. \tag{4.42}$$

If the multiplication $\mathtt{N3} * \mathtt{uEu}$ causes underflow, then $\mathtt{err0} \ge \mathtt{realmin} \ge \mathtt{N3} \cdot \mathtt{uEu}$, and otherwise (4.38)–(4.40) give

$$\mathtt{err0} \ge \frac{\mathtt{N3} \cdot \mathtt{uEu}}{1 + \mathbf{u}} \ge \frac{\mathtt{uDd} \cdot \mathtt{uEu}}{1 - \mu} \ge \frac{\|D^{-1}\delta\|_\infty}{1 - \|D^{-1}|I - \mathtt{R} \cdot \mathtt{A}| \cdot \mathtt{u}\|_\infty} \cdot |(I - \mathtt{R} \cdot \mathtt{A})| \cdot \mathtt{u}. \tag{4.43}$$

Finally (4.22) and Lemma 4.9 show that `err` is an upper bound of the right hand side in (4.36). It remains the case $\max(\mathtt{N0}) = 0$ or that in the Matlab statement `N0 = max(uEu./u)` an underflow occurs. In that case (4.39) implies

$$\mu = \|D^{-1}|I - \mathtt{R} \cdot \mathtt{A}| \cdot \mathtt{u}\|_\infty = \max_i \frac{\left[\,|I - \mathtt{R} \cdot \mathtt{A}| \cdot \mathtt{u}\right]_i}{u_i} \le \max_i \frac{\mathtt{uEu(i)}}{\mathtt{u(i)}} < \mathtt{realmin}, \tag{4.44}$$

so that $\mathtt{N1} \le \mathtt{realmin}$, $\mathtt{N2} = 1$ and $\mathtt{Den} = 1 - 3\mathbf{u}$. As in (4.42) we conclude

$$\mathtt{N3} \ge (1 - \mathbf{u})\frac{\mathtt{uDd}}{\mathtt{Den}} = \frac{1 - \mathbf{u}}{1 - 3\mathbf{u}} \cdot \mathtt{uDd} \ge \frac{1 + \mathbf{u}}{1 - \mathtt{realmin}} \cdot \mathtt{uDd} > (1 + \mathbf{u})\frac{\mathtt{uDd}}{1 - \mu}, \tag{4.45}$$

so that the lower bound of `N3` as in (4.42) is satisfied and we can continue as before. This proves that the computed vector `err` is indeed an upper bound for the right hand side in (4.36).

Both bounds (4.3) in Theorem 4.1 and (4.36) in Theorem 4.14 estimate the componentwise error of $|\delta|$, so the componentwise minimum of the bounds does as well. The additional effort to compute the bound in (4.36) is few $\mathcal{O}(n^2)$ operations. Nevertheless it can be saved if the first bound in (4.3) is already accurate enough, i.e. if the relative error $\max_i \mathtt{err}_i/|\mathtt{xs}|_i$ is small enough.

Putting things together, the following Algorithm 4.16 (`LssErrBndNear`) computes an approximation of the solution of a linear system together with a rigorous error bound. Note that we avoid to calculate the second bound based on Theorem 4.14 if $\max_i \mathtt{err0}_i/|\mathtt{xs}|_i < 2\mathtt{eps}$ because only `err0` can be improved and the offset `delta` appears also in Theorem 4.1.

Algorithm 4.16. Approximation xs and rigorous error bound err for the solution of a linear system $Ax = b$.

```
function [xs,err] = LssErrBndNear(A,b)
  ... lines in LssErrBndNear0 before "if Den>0" ...
  if Den>0                                          % algorithm successful
    err0 = (max(delta)/Den)*uE1 + realmin;          % almost final error bound
    err = (1+eps)*(delta+err0);                     % final error bound
    if max(err0./abs(xs))<2*eps, return, end        % sufficiently accurate bound
  end
  for i=1:2
    if i==1, u = PerronIter(E); end                 % Perron vector for E
    if i==2, u = delta; end                         % residual correction
    if u>0                                          % vector u is suitable
      uDd = max((1+eps)*(delta./u)) + realmin;      % upper bound of ||D^-1*delta||_inf
      aEu = E*u;                                    % approximation of |I-RA|u
      e = (n+2)*(eps/2*ufp(aEu)) + 1.5*realmin;     % bound |I-RA|u <= aEu+e
      uEu = (1+eps)*(aEu+e);                        % upper bound of |I-RA|u
      Den = (1-(1+eps)*max(uEu./u))-1.5*eps;        % suitable denominator
      if Den>0                                      % choice of u successful
        err0 = (uDd./Den)*uEu + realmin;            % upper bound of second term
        err = min(err,(1+eps)*(delta+err0));        % improved final error bound
      end
    end
  end
```

The computed error bound is correct for any positive vector u. To achieve accurate error bounds an approximation of the Perron vector of E, the upper bound of $|I - R \cdot A|$, is preferable. To equilibrate the error terms, also u=delta proved sometimes to be a good choice. Since the additional effort is small, we use both. An approximation of the Perron vector is computed by the following algorithm.

Algorithm 4.17. Approximation of the Perron vector of the nonnegative matrix E.

```
function u = PerronIter(E)
  u = ones(size(E,2),1);                        % first guess (1,...,1)^T
  for iter=1:10                                 % at most 10 iterations
    v = E*u;                                    % Perron iteration
    rho = v./u;                                 % ratio of iterates
    if min(rho)>=1, u=-1; return, end           % matrix not convergent
    if max(rho)/min(rho)<1.05, break, end       % sufficiently accurate
    u = v/norm(v,inf) + eps;                    % update taking care of 0
  end
```

Note that for a nonnegative matrix $E \in \mathbb{R}^{n\times n}$ and positive vector $u \in \mathbb{R}^n$ Collatz [47] proved

$$\min_i \frac{(Eu)_i}{u_i} \le \varrho(E) \le \max_i \frac{(Eu)_i}{u_i} \tag{4.46}$$

for $\varrho(\cdot)$ denoting the spectral radius, so that for min(v./u)>=1 the input matrix is not convergent and the iteration is stopped. Otherwise, adding eps in the second last line ensures that the updated u is positive.

4.5. Rigorous error bounds for extremely ill-conditioned linear systems in rounding to nearest (up to $\mathrm{cond}(A) \lesssim \mathbf{u}^{-2}$*)*

The obvious approach to calculate rigorous error bounds for extremely ill-conditioned linear systems is to combine Algorithm 2.1 (LssIllcoApprox) with an error bound by Theorem 4.1 or 4.14. However, this is not working.

Let a linear system $Ax = b$ with $\mathrm{cond}(A) \gtrsim \mathbf{u}^{-1}$ be given. It seems reasonable to assume that the distance $d := A^{-1}b - \mathrm{xs}$ of the exact solution $A^{-1}b$ to its nearest floating-point vector xs is of the order[6] $\|d\| \sim \mathbf{u}\|A^{-1}b\|$ (indeed there are probabilistic arguments for that, see [36]). Moreover, for a matrix $M \in \mathbb{R}^{n\times n}$ and a vector $x \in \mathbb{R}^n$ which are not correlated we can expect $\|Mx\|$ to be of the order $n^{-1/2}\|M\|\|x\|$, so that

$$\|A \cdot \mathrm{xs} - b\| = \|A \cdot d\| \sim \varphi' \cdot \mathbf{u}\|A\| \cdot \|A^{-1}b\| \sim \varphi \cdot \mathbf{u} \cdot \mathrm{cond}(A) \cdot \|b\| \gtrsim \varphi \cdot \|b\| \tag{4.47}$$

[6] Since matrix norms are equivalent, we do not to specify the norm for the following heuristic arguments.

for φ', φ not too far from 1. Note this is true without the presence of rounding errors in the computation of the residual. Let R be an approximate inverse of A. Of course, mathematically $\mathrm{cond}(A) = \mathrm{cond}(A^{-1})$; however, R is a floating-point approximation, and rounding into floating-point has some smoothing effect [36], comparable to regularization, so that cond(R) can be expected to be not much larger than $\mathbf{u}^{-1}$. Hence

$$\texttt{delta} := \|\texttt{R} \cdot (A \cdot \texttt{xs} - b)\| \gtrsim \varphi \cdot \mathbf{u}^{-1}\|b\|. \tag{4.48}$$

Note that this is true in general, no matter how accurate R and xs are. The reason is that both R and xs have floating-point entries and are thus of limited precision. Following Algorithm 2.1 (LssIllcoApprox), delta is multiplied by Cinv, the approximate inverse of R · A. Under ideal circumstances, cond(Cinv) = 1, but the approaches in Theorems 4.1 or 4.14 are expected to deliver useless error bounds of the size $\mathbf{u}^{-1}\|b\|$.

To avoid this we construct an error bound depending directly on R · b, without using an approximate solution xs. Another way would be to store xs in two parts as in [3]. However, we wanted to avoid that and to use strictly only (2.1) and (2.2) beyond ordinary floating-point arithmetic. Consider the following theorem.

Theorem 4.18. *Let $A, S \in \mathbb{R}^{n\times n}$ and $b, u \in \mathbb{R}^n$ be given. Assume $u > 0$ and let $D \in \mathbb{R}^{n\times n}$ be the diagonal matrix with u on the diagonal. Define $E := I - SA$ and assume $\|D^{-1}|E|u\|_\infty < 1$. Then A is non-singular and*

$$|A^{-1}b - S \cdot b| \le \frac{\|D^{-1}S \cdot b\|_\infty}{1 - \|D^{-1}|E|u\|_\infty} \cdot |E|u. \tag{4.49}$$

For $e := (1, \ldots, 1)^T$ it follows in particular

$$|A^{-1}b - S \cdot b| \le \frac{\|S \cdot b\|_\infty}{1 - \|E\|_\infty} \cdot |E|e. \tag{4.50}$$

Proof. Using $(I - E)^{-1} = D(I - D^{-1}ED)^{-1}D^{-1}$ and $|Ex| \le \|x\|_\infty \cdot |E|e \in \mathbb{R}^n$ for $x \in \mathbb{R}^n$ as in the proof of Theorem 4.1, and $De = u > 0$ and $\|E\|_\infty = \||E|e\|_\infty$ it follows

$$\begin{aligned} |A^{-1}b - S \cdot b| &= |E(I - E)^{-1}Sb| = |ED \cdot (I - D^{-1}ED)^{-1}D^{-1}Sb| \\ &\le \|(I - D^{-1}ED)^{-1}D^{-1}Sb\|_\infty \cdot |ED|e \\ &\le \frac{\|D^{-1}S \cdot b\|_\infty}{1 - \|D^{-1}|E|u\|_\infty} \cdot |E|u. \end{aligned}$$

Setting $u := e$ finishes the proof. □

Note that there is no restriction on S. Now the trick is, as explained following Observation 2.2, to define $S :=$ Cinv · R, but rather than computing Sb to use Cinv · (R · b). This has the appealing advantage to be faster and more accurate than (Cinv · R) · b.

Corresponding error bounds in rounding to nearest based on Theorem 4.18 are not difficult to compute using the results of the previous subsections. First consider the following algorithm to compute error bounds for the product of two matrices where the second factor is afflicted with an error term.

Algorithm 4.19. Rigorous bounds $R \pm E$ of matrix products $Q * \tilde{P}$ for $P - eP \le \tilde{P} \le P + eP$.

```
function [R,E] = Prod2Bnd(Q,P,eP)
  [R,eR] = DotErr(Q,P);                  % error bound R+/-eR of Q*P
  [aD,eD] = DotErr(abs(Q),eP);           % error bound aD+/-eD of |Q|*eP
  aE = eR + aD + eD;                     % not yet upper bound of |R-Q*(P+/-eP)|
  E = aE + eps*ufp(aE);                  % upper bound of |R-Q*(P+/-eP)|
```

Let $\texttt{Q} \in \mathbb{F}^{m\times k}$ and P, eP $\in \mathbb{F}^{k\times n}$ with eP ≥ 0 be given, and assume $(n + 2)\mathbf{u} \le 1$. Let $\tilde{P} \in \mathbb{R}^{k\times n}$ with

$$\texttt{P} - \texttt{eP} \le \tilde{P} \le \texttt{P} + \texttt{eP}$$

be given. Then

$$|\texttt{Q} \cdot \tilde{P} - \texttt{Q} \cdot \texttt{P}| \le |\texttt{Q}| \cdot \texttt{eP}.$$

The analysis of DotErr in Theorem 4.4 yields

$$|\texttt{Q} \cdot \texttt{P} - \texttt{R}| \le \texttt{eR} \quad \text{and} \quad |\texttt{Q}| \cdot \texttt{eP} \le \texttt{aD} + \texttt{eD},$$

hence

$$|\texttt{Q} \cdot \tilde{P} - \texttt{R}| \le \texttt{eR} + \texttt{aD} + \texttt{eD}.$$

Therefore the analysis of SumPosBnd in Theorem 4.8 and eps $= 2\mathbf{u}$ prove

$$|\mathtt{Q}\cdot\tilde{P}-\mathtt{R}|\le \mathtt{E}. \tag{4.51}$$

With these preliminaries we can state the algorithm to compute error bounds for extremely ill-conditioned linear systems in rounding to nearest.

Algorithm 4.20. Approximation xs and rigorous error bound err for the solution of a linear system $Ax = b$ for extremely ill-conditioned matrix A.

```
function [xs,err] = LssIllcoErrBndNear(A,b)
  err = NaN(size(b));                              % initialize result
  n = size(A,2);                                   % dimension of the linear system
  R = inv(A);                                      % approximate inverse
  while any(isinf(R(:))) || any(isnan(R(:)))
    R = inv(A.*(1+randn(n)*eps));                  % inversion of perturbed matrix
  end
  [P,eP] = Dot2Near(R,A);                          % error bound P+/-eP for R*A
  Q = inv(P);                                      % approximate inverse of P
  [aSA,eSA] = Prod2Bnd(Q,P,eP);                    % error bound aSA+/-eSA for Q*(R*A)
  [y,ey] = Dot2Near(R,b);                          % error bound y+/-ey for R*b
  [xs,eSb] = Prod2Bnd(Q,y,ey);                     % error bound xs+/-eSb for Q*(R*b)
  delta = (1+eps)*(abs(xs)+eSb);                   % upper bound of |Q*(R*b)|
  SA_I = (1+eps)*abs(aSA-eye(n));                  % upper bound of |aSA-I|
  E = (1+eps)*(SA_I+eSA);                          % upper bound of |Q*(R*A)-I|
  aE1 = sum(E,2);                                 % approximation of |Q*(R*A)-I|*ones(n,1)
  uE1 = aE1 + (n+1)*(0.5*eps*ufp(aE1));            % upper bound of |Q*(R*A)-I|*ones(n,1)
  Den = (1-max(uE1)) - 1.5*eps;                    % lower bound of 1-||E||_inf
  if Den>0                                         % algorithm successful
    err = (max(delta)/Den)*uE1 + realmin;          % final error bound
    if max(err./abs(xs))<2*eps, return, end        % bound sufficiently accurate
  end
  for i=1:2
    if i==1, u = PerronIter(E); end                % Perron vector for E
    if i==2, u = delta; end                        % residual correction
    if u>0                                         % vector u is suitable
      uDd = max((1+eps)*(delta./u)) + realmin;     % upper bound of ||D^-1*delta||_inf
      aEu = E*u;                                   % approximation of |I-RA|u
      e = (n+2)*(eps/2*ufp(aEu)) + 1.5*realmin;    % bound |I-RA|u <= aEu+e
      uEu = (1+eps)*(aEu+e);                       % upper bound of |I-RA|u
      Den = (1-(1+eps)*max(uEu./u))-1.5*eps;       % suitable denominator
      if Den>0                                     % choice of u successful
        err = min(err,(uDd./Den)*uEu + realmin); % final error bound
      end
    end
  end
```

In the first lines the matrices R, P, eP, $\mathtt{Q}\in\mathbb{F}^{n\times n}$ are computed with

$$|\mathtt{P}-\mathtt{R}\cdot\mathtt{A}|\le \mathtt{eP}. \tag{4.52}$$

Otherwise there are no assumptions on A, R or Q. Abbreviate $\mathtt{S}:=\mathtt{Q}\cdot\mathtt{R}$, then (4.51) implies

$$|\mathtt{S}\cdot\mathtt{A}-\mathtt{aSA}|\le \mathtt{eSA}. \tag{4.53}$$

Similarly, $|\mathtt{y}-\mathtt{R}\cdot\mathtt{b}|\le \mathtt{ey}$ and (4.51) imply

$$|\mathtt{S}\cdot\mathtt{b}-\mathtt{xs}|\le \mathtt{eSb}, \tag{4.54}$$

and Lemma 4.9 yields

$$|\mathtt{S}\cdot\mathtt{b}|\le \mathtt{delta}\quad\text{and therefore}\quad \|\mathtt{S}\cdot\mathtt{b}\|_\infty\le \mathtt{max(delta)}. \tag{4.55}$$

As in (4.23)ff. in the analysis of Algorithm 4.11 (LssErrBndNear0) it follows

$$|I-\mathtt{S}\cdot\mathtt{A}|\le \mathtt{E},\qquad \|I-\mathtt{S}\cdot\mathtt{A}\|_\infty\le \mathtt{max(uE1)}\quad\text{and finally}\quad \frac{\|\mathtt{S}\cdot\mathtt{b}\|_\infty}{1-\|E\|_\infty}\cdot|I-\mathtt{S}\cdot\mathtt{A}|\cdot e\le \mathtt{err} \tag{4.56}$$

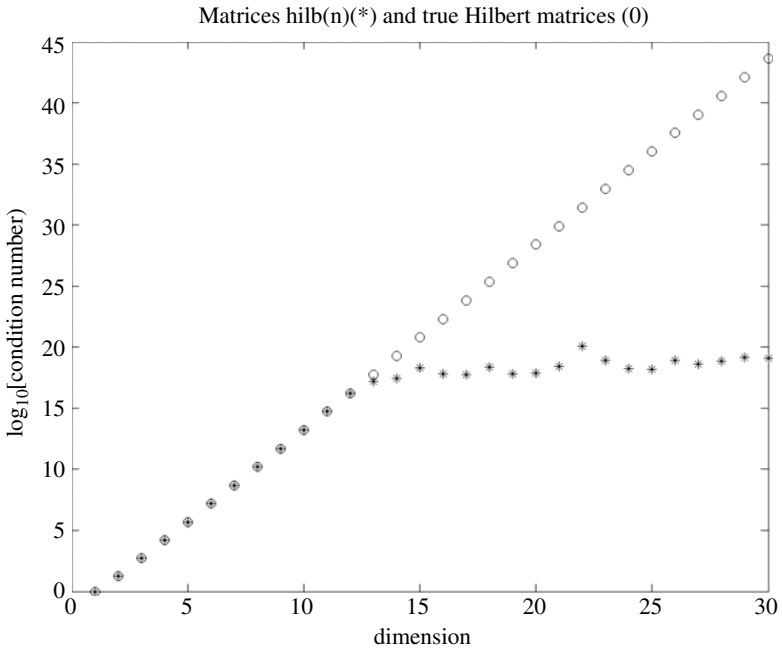

Fig. 5.1. Logarithm of the condition number of `hilb(n)` (*) and of the true Hilbert matrix $H_{ij} = 1/(i+j-1)$ (o).

provided `Den` > 0, so that in this case `err` is an upper bound of the right hand side in (4.50).[7] The remaining of the analysis is analogous to the proof of correctness of Algorithm 4.16 (`LssErrBndNear`) in (4.38)ff. with replacing `R` by `S`. Both bounds (4.49) and (4.50) are componentwise upper bounds for the error of $\mathtt{S}\cdot\mathtt{b}$, so the minimum of both is a valid bound as well. As before we use for `u` both an approximation of the Perron vector of `E` as well as `delta`. If the computation of the first bound was not successful it is set to `NaN`, and observing that Matlab ignores `NaN`'s when computing a minimum, we proved the following theorem.

Theorem 4.21. *Let a set $\mathbb{F}$ of floating-point numbers with relative rounding error unit* $\mathbf{u}$ *together with floating-point operations complying with the IEEE* 754 *arithmetic standard* [10,11] *be given.*

Let `xs` *and* `err` *be the results of Algorithm* 4.20 (`LssIllcoErrBndNear`) *applied to a matrix* $\mathtt{A} \in \mathbb{F}^{n\times n}$ *and a vector* $\mathtt{b} \in \mathbb{F}^n$ *with* $(n+2)\mathbf{u} \leq 1$. *If the algorithm ends successfully, then* `A` *is non-singular and*

$$|\mathtt{xs} - A^{-1}b| \leq \mathtt{err}. \tag{4.57}$$

Computational evidence suggests that the algorithm ends successfully for condition numbers up to about $\mathbf{u}^{-2}$.

As in Algorithm 2.1 (`LssIllcoApprox`) the precondition matrix `R` is replaced by $\mathtt{S} = \mathtt{Q}\cdot\mathtt{R}$, and as before it is important to compute $\mathtt{S}\cdot\mathtt{A}$ and $\mathtt{S}\cdot\mathtt{b}$ in the order $\mathtt{Q}\cdot(\mathtt{R}\cdot\mathtt{A})$ and $\mathtt{Q}\cdot(\mathtt{R}\cdot\mathtt{b})$, respectively. In the latter case this also reduces the computing time to $\mathcal{O}(n^2)$ operations.

5. Computational results

Following we report computational results. All algorithms are tested in Matlab version 7.11.0.584 (R2010b) on an Intel Core i7 CPU M640 with 2.8 GHz, INTLAB version 6 and Windows 7 operating system. INTLAB [48] is the Matlab toolbox for reliable computing written by the author of this paper. To our knowledge there is no other publicly available Matlab code to compare with for rigorously solving linear systems using only rounding to nearest. If not stated otherwise, we always use the 2-norm condition number $\|A^{-1}\|_2\|A\|_2$.

It is not obvious how to generate matrices with floating-point entries with condition number much larger than $\mathbf{u}^{-1}$. Of course, one may try the "usual suspects" like Hilbert matrices or the notoriously ill-conditioned Vandermonde matrices. However, as has been mentioned before, extremely ill-conditioned matrices hardly have a condition number much larger than $\mathbf{u}^{-1}$ when rounded into floating-point. The situation is shown in Fig. 5.1, where the condition number (computed by the symbolic toolbox in Matlab) of the (Matlab) floating-point approximation `hilb(n)` and the true Hilbert matrix is shown. This behavior is typical, with some exceptions mentioned below, that the condition number of non-singular floating-point matrices is roughly bounded by $\mathbf{u}^{-1}$.

[7] Note that unlike (4.3) there is no additive term in (4.50).

Table 5.1
Test matrices of small dimension (equilibrated).

Matrix	Matlab command	Definition	$n_{\max}$	$\mathrm{cond}_{1\le n\le 40}^{\max}$
Pascal	`pascal(n)`	$\binom{i+j-1}{j-1}$	31	$6.0\cdot 10^{31}$
Hilbert	`hilb(n)`	$1/(i+j-1)$	1	$6.3\cdot 10^{19}$
Scaled Hilbert		$\mathrm{lcm}(1,\ldots,2n-1)/(i+j-1)$	21	$1.1\cdot 10^{28}$
Inverse Hilbert	`invhilb(n)`	Defined by explicit formula	12	$4.8\cdot 10^{37}$
Boothroyd		$\frac{\binom{n+i-1}{i-1}\cdot n\cdot\binom{n-1}{n-j}}{i+j-1}$	20	$1.1\cdot 10^{30}$
Vandermonde	`vander(1:n)`	i^{n-j}	14	$4.7\cdot 10^{61}$

Larger condition numbers are possible for some of the usual test matrices as long as they are exactly representable in floating-point. Well-known examples are listed in Table 5.1. Some of them are directly available in Matlab. The matrix entries like i^{n-j} for the Vandermonde matrix are computed in floating-point and coincide with the mathematical definition until dimension $n \le n_{\max}$. For larger dimensions the entries are corrupted by rounding errors and the matrix does not coincide with the mathematical definition. Usually this has a smoothing effect so that the condition number does not increase any more with increasing dimension. The maximum condition number for dimensions $1 \le n \le 40$ is displayed in the last column of Table 5.1.

Usually it is preferable to equilibrate the input matrix to improve the condition number. This is also done in [3]. To avoid rounding errors we use the nearest power of 2 to equilibrate the input matrix. This works well except for Vandermonde matrices which show a very special and in some way strange behavior, see Section 5.4. Henceforth all matrices, also in Table 5.1, are equilibrated.

Our algorithms are designed to solve general linear systems, not taking advantage of any structure of the matrix. Some of the test matrices mentioned so far do have a special structure, for example being totally nonnegative. Taking into account this property, many numerical problems can be solved with high relative accuracy of the result [49–52]. A famous example is that the smallest singular value of the 100×100 Hilbert matrix, which is of size 10^{-151}, can be computed in IEEE 754 double precision to almost full accuracy. This was noted in [53], possibly the starting point for an extensive research on this topic.

A reason for this unexpected behavior, seemingly contradicting common perturbation analysis, is that it can be shown that some algorithms applied to such structured matrices perform only certain arithmetic operations, prohibiting in particular catastrophic cancellation [54].

Although this important research allows to solve a number of very ill-conditioned problems, certain structural properties of the matrix are mandatory. In particular a number of the "usual suspects" satisfy those properties. However, in contrast to our algorithms, those methods do not apply to general matrices.

5.1. Results for `LssIllcoApprox`

We start with some timing comparison. Note that all our algorithms are completely implemented in Matlab, and in particular the extra-precise accumulation of dot products suffers from interpretation overhead. Matlab offers in the symbolic toolbox some multi-precision arithmetic (vpa) for approximate calculations, and a rational arithmetic to compute the exact solution of linear systems. We compare the computing times for Algorithm 2.1 (`LssIllcoApprox`) (producing an approximation) with the variable precision arithmetic package (vpa), and Algorithm 4.20 (`LssIllcoErrBndNear`) (producing a rigorous error bound) with the precise result of the solution of the linear system $Ax = b$ computed by `sym(A,'f') \ sym(b,'f')`.[8]

As can be seen in Table 5.2 our algorithms are faster than the competitors from the symbolic toolbox. We note, however, that the comparison is not fair because `sym(A,'f') \ sym(b,'f')` computes the exact, rational solution whenever the input matrix in non-singular, whereas `LssIllcoErrBnd` computes only error bounds and may fail. Apparently using the symbolic toolbox seems the only possibility in Matlab to compute rigorous results for extremely ill-conditioned matrices. Note that `LssIllcoApprox` improves the initial approximation by an extra-precise residual iteration whereas `LssIllcoErrBndNear` does not. This explains why for smaller dimension the former algorithm is slower than the latter.

Concerning the accuracy of the results, the right graph in Fig. 2.1 shows already the performance of Algorithm 2.1 (`LssIllcoApprox`) for Pascal matrices for right hand sides `b=randn(n,1)` and `b=A*randn(n,1)`. Recall that the Matlab function `rand` generates pseudo-random values drawn from a uniform distribution on the unit interval, whereas `randn` produces pseudo-random values drawn from a normal distribution with mean zero and standard deviation one. As can be seen, the accuracy of the bounds decreases with increasing condition number. For all examples with maximum condition number up to $6.0\cdot 10^{31}$, on the median at least 3 to 4 digits of the solution are correct.

For this and all of the following examples for Algorithm `LssIllcoApprox` the number of residual iterations is mostly 1 or 2, in very few cases 3 iterations, but never more.

[8] The extra parameter `'f'` ensures that the input A and b is converted into long format without error, respectively.

Table 5.2
Absolute and relative computing time in seconds for Algorithm 2.1 (`LssIllcoApprox`), the variable precision arithmetic package (vpa), Algorithm 4.20 (`LssIllcoErrBndNear`) and `sym(A,'f') \ sym(b,'f')`. Based on an algorithm given in [37], matrices of condition number 10^{25} are generated in INTLAB [48] by `randmat(n,1e25)` with random right hand side.

Dimension	Absolute computing time [s]				Relative computing time	
	Algorithm 2.1	Vpa	Algorithm 4.20	Sym	Vpa/Approx	Sym/ErrBnd
10	0.010	0.031	0.0040	0.039	3.2	9.7
20	0.019	0.089	0.0073	0.12	4.6	16.6
50	0.051	0.38	0.019	0.71	7.6	36.4
100	0.13	1.73	0.063	7.49	13.2	117.7
200	0.39	10.3	0.28	105.8	26.6	375.2
500	8.8	149.1	8.6	4140	16.9	479.1

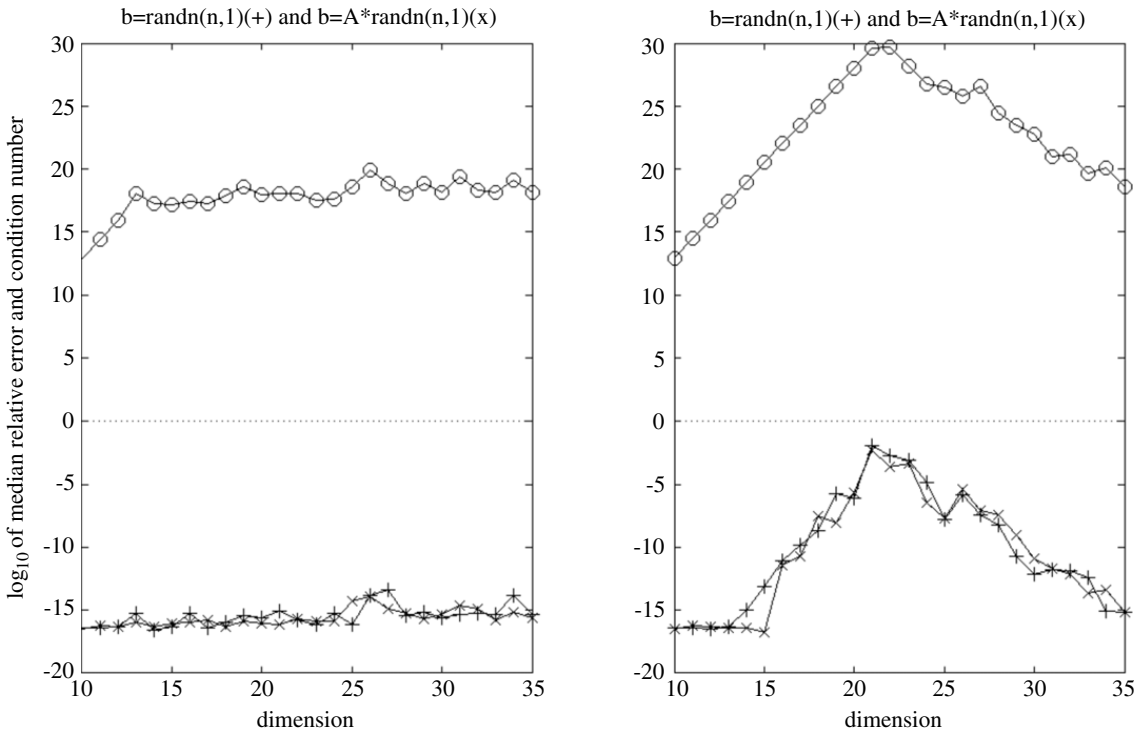

Fig. 5.2. Accuracy of Algorithm 2.1 (`LssIllcoApprox`) for Hilbert and scaled Hilbert matrices as defined in Table 5.1. The upper parts show the condition number, the lower parts the relative error of the results.

Next we use `A = hilb(n)` as defined in Matlab with entries approximating $1/(i+j-1)$, and second the scaled Hilbert matrix with entries $lcm(1,\ldots,2n-1)/(i+j-1)$. Up to dimension $n = 21$ the entries are an integer multiple of the original Hilbert matrix, for $n > 21$ the entries are corrupted by rounding errors. Therefore, the scaled Hilbert matrices achieve much larger condition numbers, as the original Hilbert matrix, than `hilb(n)`, see Fig. 5.1. In Fig. 5.2 the median relative error of Algorithm 2.1 for the right hand sides `b=randn(n,1)` and `b=A*randn(n,1)` are printed in one graph each. Again the accuracy of the result is inverse proportional to the condition number.

In the left of Fig. 5.3 we show the same results of Algorithm 2.1 (`LssIllcoApprox`) for `A = invhilb(n)`, the approximation of the inverse Hilbert matrix for right hand sides `b=randn(n,1)` and `b=A*randn(n,1)`. Now the relative error is constantly less than 10^{-14} although the condition number rises to almost 10^{40}. A similar behavior is observed for Vandermonde matrices and will be discussed in Section 5.4.

As has been mentioned, in lines 9 and 14 in Algorithm 2.1 one might use extra-precise dot products by `Dot2Near` in the multiplication of the residual by `Cinv`. The results for matrices proposed by Boothroyd [55] with integer entries, where a checkerboard sign distribution produces its inverse, are displayed in the right of Fig. 5.3. The results for right hand side `b=randn(n,1)` are displayed; the results for `b=A*randn(n,1)` are completely similar.

As can be seen there is not too much difference whether `Dot2Near` is used in lines 9 and 14 or not, and for other matrices and other right hand sides a similar behavior is observed. Therefore, this extra computing time in Algorithm 2.1 (`LssIllcoApprox`) is saved. Note again how the accuracy of the result corresponds to the condition number.

For larger dimensions it is very time consuming to compute an accurate solution to test against. But the results of Algorithm 4.20 (`LssIllcoErrBndNear`) are based on an approximate solution computed along the lines of Algorithm 2.1

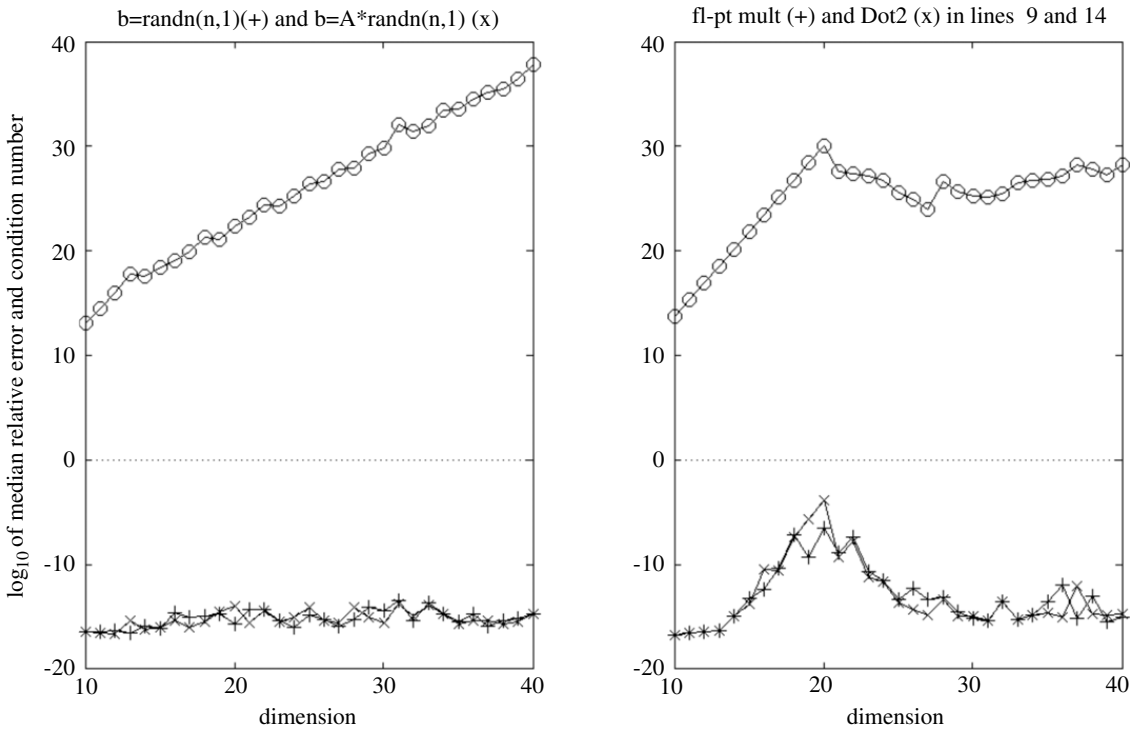

Fig. 5.3. Accuracy of Algorithm 2.1 (`LssIllcoApprox`) for inverse Hilbert and Boothroyd matrices as defined in Table 5.1. The upper parts show the condition number, the lower parts the relative error of the results.

Table 5.3
Median relative error of the bounds computed by Algorithm 4.16 (`LssErrBndNear`) for the matrices in Table 5.1.

Matrix	n	cond(A)	`b=randn(n,1)`	`b=A*randn(n,1)`
Pascal	14	$1.4 \cdot 10^{13}$	$5.1 \cdot 10^{-17}$	$1.2 \cdot 10^{-19}$
	15	$1.6 \cdot 10^{14}$	$3.3 \cdot 10^{-17}$	$1.2 \cdot 10^{-17}$
	16	$1.8 \cdot 10^{15}$	$4.8 \cdot 10^{-17}$	$1.0 \cdot 10^{-17}$
	17	$2.2 \cdot 10^{16}$	$2.0 \cdot 10^{-16}$	$6.3 \cdot 10^{-17}$
	18	$2.5 \cdot 10^{17}$	Failed	Failed
Hilbert	11	$7.4 \cdot 10^{12}$	$4.9 \cdot 10^{-17}$	$4.5 \cdot 10^{-17}$
Inverse Hilbert	11	$1.1 \cdot 10^{13}$	$4.3 \cdot 10^{-17}$	$5.1 \cdot 10^{-17}$
Scaled Hilbert	11	$8.7 \cdot 10^{12}$	$4.3 \cdot 10^{-17}$	$5.2 \cdot 10^{-17}$
Boothroyd	11	$5.0 \cdot 10^{13}$	$6.1 \cdot 10^{-17}$	$1.1 \cdot 10^{-19}$
Vandermonde	13	$3.0 \cdot 10^{14}$	$4.4 \cdot 10^{-17}$	$7.8 \cdot 10^{-17}$

(`LssIllcoApprox`); so from the following results for `LssIllcoErrBndNear` we can deduce that the approximations by Algorithm 2.1 (`LssIllcoApprox`) are at least as good as the computed bounds. Computational results for the latter are given in Section 5.3.

5.2. Results for *LssErrBndNear*

Next we discuss the results of Algorithm 4.16 (`LssErrBndNear`). Computing times are already given in Table 4.1. Those are for Algorithm 4.11 (`LssErrBndNear0`), but the difference to Algorithm 4.16 (`LssErrBndNear`) is only few $\mathcal{O}(n^2)$ operations and negligible.

Concerning accuracy, we first consider the matrices in Table 5.1. Again we tested linear systems with right hand sides `b=randn(n,1)` and `b=A*randn(n,1)`. As a typical example we display in Table 5.3 the results for Pascal matrices for dimensions 14 to 18. To save space we display for the other matrices from Table 5.1 only the results of Algorithm `LssErrBndNear` for the highest dimension it succeeded.

As can be seen the relative error is of the order $\mathbf{u}$ and better. So the extra-precise residual iteration performs as expected, not only for approximations but also for the rigorous error bound. Also the maximum condition number for which `LssErrBndNear` successfully computes an error bound is not too far from $\mathbf{u}^{-1}$, which is due to the Perron iteration (see also Fig. 4.1). However, this is partly due to the fact that the dimensions are small. Better results in this respect (using directed rounding) are obtained by Algorithm 4.2 (`LssErrBnd`) to be described in Section 2 of this paper. For all matrices except Vandermonde, this algorithm can handle one dimension larger.

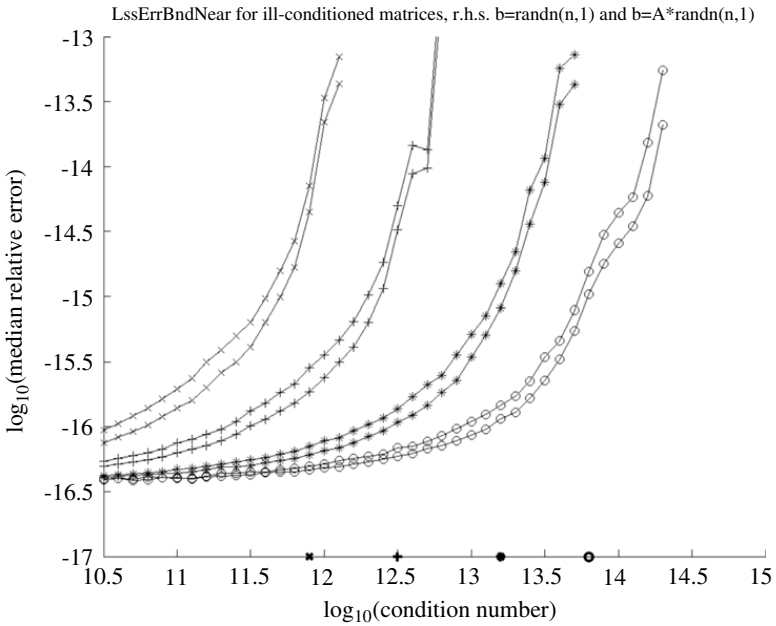

Fig. 5.4. Results of Algorithm 4.16 (`LssErrBndNear`) for ill-conditioned random matrices of dimension $n = 100$ (o), $n = 200$ ($*$), $n = 500$ ($+$) and $n = 1000$ ($\times$).

Next we treat ill-conditioned matrices of larger dimension. For dimensions up to $\mathbf{u}^{-1}$ we may safely use `randsvd(n,cnd)` from the Matlab matrix gallery applying some random orthogonal transformation from the left and right to a diagonal matrix with specified singular values. As mentioned before, this approach is not applicable for condition numbers beyond $\mathbf{u}^{-1}$ because of the inevitable presence of rounding errors.

In Fig. 5.4 the median relative errors of all solution components of the result of Algorithm `LssErrBndNear` for right hand side `b=randn(n,1)` and `b=A*randn(n,1)` over 100 samples is displayed for dimensions $n = 100$ (o), $n = 200$ ($*$), $n = 500$ ($+$) and $n = 1000$ ($\times$). Note that the maximum relative error over all samples is below 10^{-13}, so that until failure for a condition number of roughly $\mathbf{u}^{-1}/n$ the accuracy of the bounds is not too far from $\mathbf{u}$. For dimensions $n = 100$, $n = 200$, $n = 500$ and $n = 1000$ there is no failure in the 100 samples until condition numbers $7.9 \cdot 10^{13}$, $2.5 \cdot 10^{13}$, $4.0 \cdot 10^{12}$ and $1.6 \cdot 10^{12}$, respectively, as depicted on the x-axis in Fig. 5.4. Again more ill-conditioned matrices can be treated in the same computing time using directed rounding as by Algorithm 4.2 (`LssErrBnd`) presented in Part II of this paper.

Finally we mention that one may compute the error bound of `R * A` by `[aRA,eRA] = Dot2Near(R,A)` rather than `[aRA,eRA] = DotErr(R,A)`. Due to the interpretation overhead this would be costly; however, in all test examples we did not see a significant difference in accuracy.

Next we discuss the quality of rigorous error bounds computed by Algorithm 4.20 (`LssIllcoErrBndNear`) for extremely ill-conditioned matrices.

5.3. Results for *LssIllcoErrBndNear*

A timing comparison has already been shown in Table 5.2. Concerning accuracy we first treat the matrices in Table 5.1. The results for Pascal matrices are already shown in the right graph of Fig. 2.1. As expected the rigorous error bound is weaker than the approximate solution shown in the left graph. In both cases the quality is nicely inverse proportional to the condition number.

The results for Hilbert and scaled Hilbert matrices do not reveal new information, so the graphs are omitted to save space. Similar to Fig. 5.2 the rigorous error bound is weaker, but the behavior is similar. The same holds true for inverse Hilbert and Boothroyd matrices as shown in Fig. 5.5. The results are similar to the approximate solutions computed by Algorithm `LssIllcoApprox` shown in Fig. 5.3; the results for right hand sides `randn(n,1)` and `A*randn(n,1)` are practically identical.

Again the results are surprisingly good for inverse Hilbert matrices: for example, for dimension $n = 40$ the matrix has a condition number larger than 10^{37}, but nevertheless the error bound guarantees more than 10 correct digits of the solution for both right hand sides `b=randn(n,1)` and `b=A*randn(n,1)`. We have no conclusive explanation for that; a similar behavior is observed for Vandermonde matrices and will be discussed in Section 5.4.

Finally we consider extremely ill-conditioned matrices of dimensions up to $n = 1000$. A method how to construct extremely ill-conditioned matrices being exactly representable in floating-point of higher dimension is described in [37]. Moreover, interesting methods can be found in [38,39]. Yet another way, which is used in INTLAB [48] for condition numbers up to about 10^{100}, is to multiply a couple of sparse unit lower triangular matrices with small integer entries and to form A^TA until the desired condition number is achieved. For the following data we tried all methods with similar results.

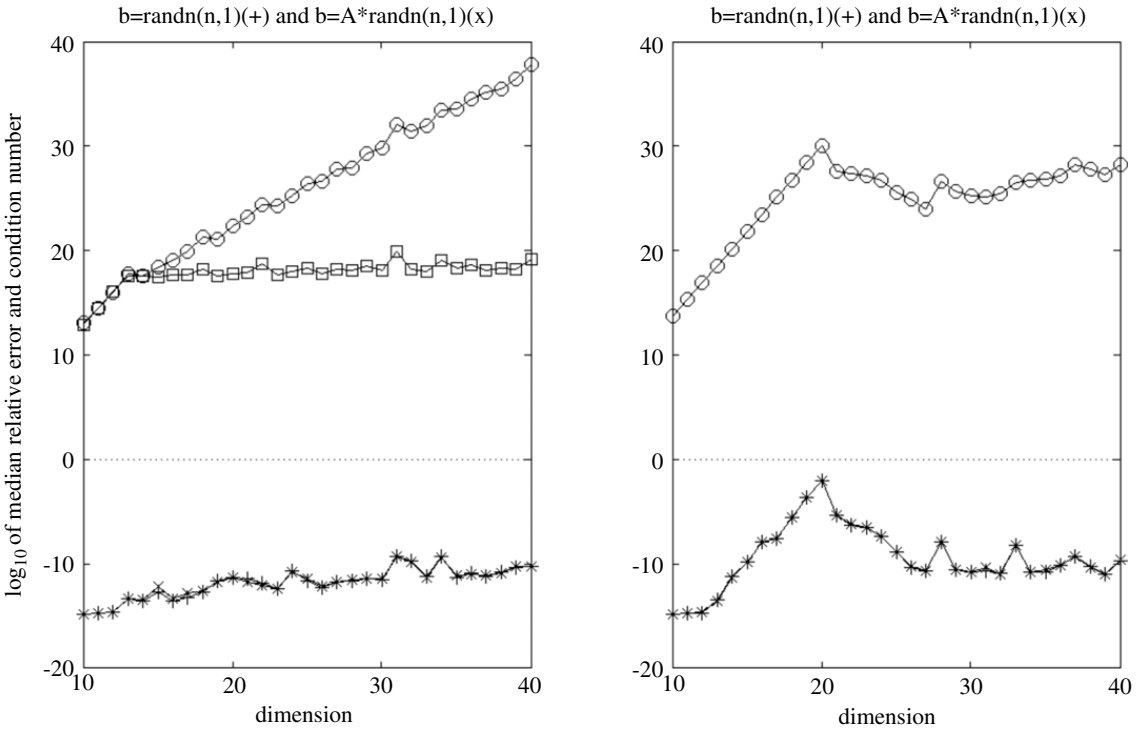

Fig. 5.5. Accuracy of Algorithm 4.20 (`LssIllcoErrBndNear`) for inverse Hilbert and Boothroyd matrices as defined in Table 5.1. The upper parts show the condition number, the lower parts the relative error of the results. In the left graph, "∘" corresponds to the traditional condition number $\|A^{-1}\|_2\|A\|_2$ and "□" to the Bauer–Skeel condition number $\| \,|A^{-1}| \cdot |A| \,\|$.

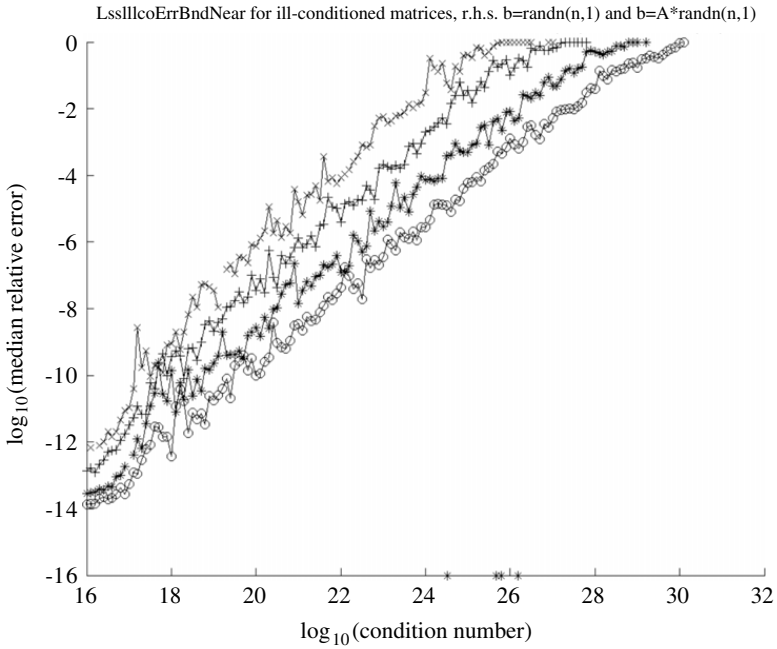

Fig. 5.6. Results of Algorithm 4.16 (`LssErrBndNear`) for ill-conditioned random matrices of dimension $n = 100$ (o), $n = 200$ ($*$), $n = 500$ ($+$) and $n = 1000$ ($\times$).

In Fig. 5.6 the results of Algorithm 4.20 (`LssIllcoErrBndNear`) for dimensions $n \in \{100, 200, 500, 1000\}$ and for right hand sides `b=randn(n,1)` and `b=A*randn(n,1)` are shown in one graph. With increasing condition number the quality of the error bounds decreases. The maximum treatable condition number is of the order $\mathbf{u}^{-2}/n^2$. In contrast, the results of Algorithm 4.16 (`LssErrBndNear`) as shown in Fig. 5.4 are always of high accuracy — until the algorithm fails. This is due to the extra-precise residual iteration in `LssErrBndNear`, which could not be used in `LssIllcoErrBndNear`. For dimensions $n = 100$, $n = 200$, $n = 500$ and $n = 1000$ there is no failure of `LssIllcoErrBndNear` in the 100 samples until condition numbers $6.2 \cdot 10^{25}$, $1.5 \cdot 10^{26}$, $4.7 \cdot 10^{25}$ and $3.3 \cdot 10^{24}$, respectively, as depicted on the x-axis in Fig. 5.6. Accidentally the number for dimension $n = 200$ is larger than for $n = 100$ due to the randomness of the examples.

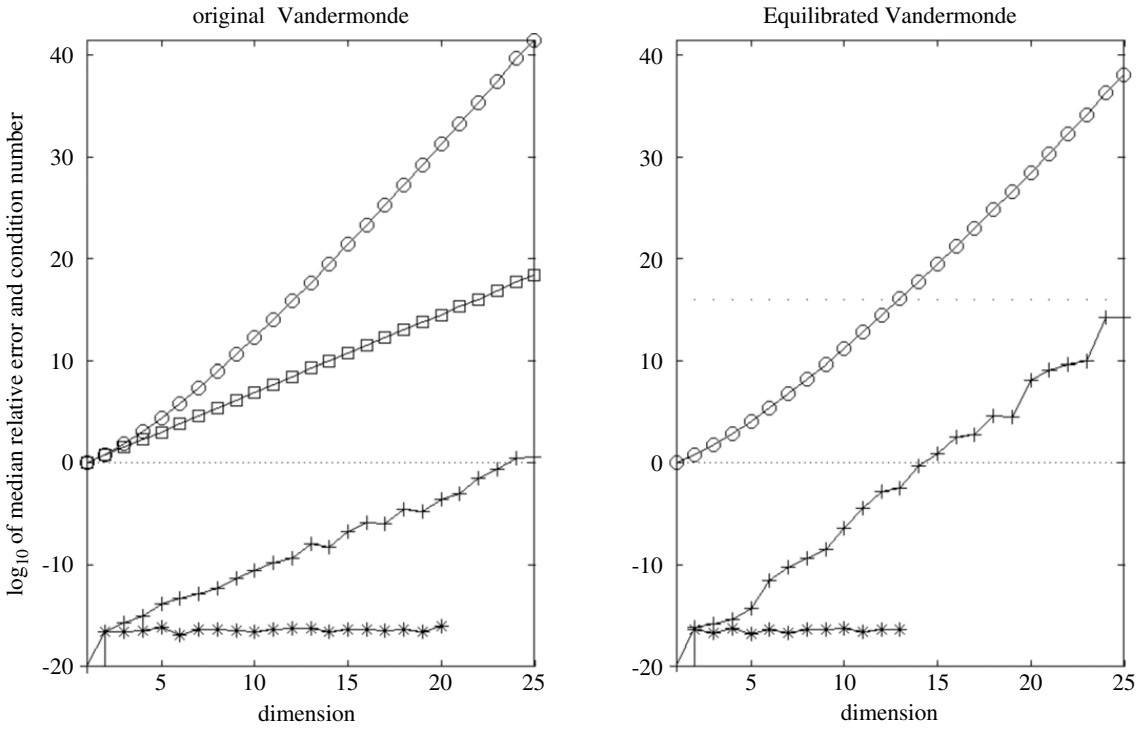

Fig. 5.7. Accuracy of Algorithm 4.20 (`LssIllcoErrBndNear`) ($*$) and Matlab built-in $A \setminus b$ ($+$) for original (left graph) and equilibrated Vandermonde matrices (right graph) as defined in Table 5.1. The upper parts show the condition number, the lower parts the relative error of the results. In both cases the right hand side is `b=randn(n,1)`. Condition numbers of original and equilibrated Vandermonde matrices (o) and of optimally scaled matrices ($\square$) are shown.

5.4. Vandermonde matrices

We observed in Fig. 5.5 that the quality of the error bounds for the inverse Hilbert matrices was considerably better than one would expect from the condition number. For example, the condition number for $n = 40$ is $2.3 \cdot 10^{37}$, but nevertheless the rigorous bounds by Algorithm 4.20 (`LssIllcoErrBndNear`) are accurate to 10 decimal digits.

A similar behavior can be observed for the notoriously ill-conditioned Vandermonde matrices [56,57]. In Fig. 5.7 we display the results of `LssIllcoErrBndNear` for the original Vandermonde matrix as defined in Table 5.1 and of its equilibrated version. In both cases the right hand side is `b=randn(n,1)`; the results for `b=A*randn(n,1)` are completely similar. The circles in the upper halves denote the traditional condition number $\|A^{-1}\|_2\|A\|_2$, increasing rapidly beyond 10^{40}. In the lower halves the median relative error over all solution components for the built-in Matlab call $A \setminus b$ is given by "$+$". For comparison also the median relative error of `LssIllcoErrBndNear` is depicted by "$*$". For small dimension sometimes the solution is exactly representable; in that case an error zero is replaced by 10^{-20}.

In the left graph of Fig. 5.7 we see that for condition numbers way beyond 10^{20} the results of the built-in Matlab approximation $A \setminus b$ still maintain some accuracy, apparently contradicting the well-accepted rule of thumb that in IEEE 754 double precision for condition number 10^k about $16-k$ correct digits can be expected. In contrast, the linear system with the equilibrated matrix in the right graph shows the expected behavior: the relative error surpasses 1 at the condition number 10^{16} (the dotted line in the upper half). Here is yet another example that equilibration need not to improve the quality of the result.

We do not have a convincing explanation for that. Similar contradictions to the mentioned rule of thumb have been observed in [3]. A reason might be the following. The Bauer–Skeel condition number $\kappa := \| \, |A^{-1}| \cdot |A| \, \|_\infty$ is displayed by "$\square$" in the left graph in Figs. 5.5 and 5.7. This is the optimal normwise condition number achievable by left diagonal scaling, and it is also equal to the componentwise condition number with respect to relative perturbations of the matrix entries.

The accuracy of the Matlab-approximation $A \setminus b$ satisfies the well-known rule of thumb with respect to the Bauer–Skeel condition number for the original Vandermonde matrix, see Fig. 5.7; the same observation applies to inverse Hilbert matrices as in Fig. 5.5. In any case the computational results for inverse Hilbert and Vandermonde matrices are nice but seem to be a kind of artifact.

6. Conclusion

In this Part I of the article all algorithms use solely ordinary floating-point arithmetic in rounding to nearest. One purpose of the paper is to show that it is fairly simple to obtain rigorous error bounds subject to that constraint.

An algorithm for computing approximations of reasonable quality for extremely ill-conditioned matrices with condition number $\gg \mathbf{u}^{-1}$ was given. Moreover, algorithms computing rigorous error bounds (including possible underflow), also for extremely ill-conditioned matrices, have been presented. All algorithms are given in executable Matlab-code. All algorithms use only the basic floating-point operations in rounding to nearest. For the extra-precise dot product, i.e. products accumulated in double the working precision with result rounded into working precision, an algorithm using only the basic floating-point operations in rounding to nearest was given as well.

The error bounds are of high quality, however, certain estimates in rounding to nearest seem improvable. Considerably sharper error bounds for ill-conditioned matrices and also for extremely ill-conditioned matrices are presented in Part II of this paper. They are a little bit more involved; of course, they can be computed in rounding to nearest, but are better and easier to discuss using directed rounding.

Acknowledgment

The author wishes to thank an anonymous referee for helpful comments, in particular for pointing to accurate algorithms to solve extremely ill-conditioned linear systems with special matrices.

References

[1] N.J. Higham, Accuracy and Stability of Numerical Algorithms, second ed., SIAM Publications, Philadelphia, 2002.
[2] R. Skeel, Iterative refinement implies numerical stability for gaussian elimination, Math. Comp. 35 (151) (1980) 817–832.
[3] J.B. Demmel, Y. Hida, W. Kahan, X.S. Li, S. Mukherjee, E.J. Riedy, Error bounds from extra precise iterative refinement, ACM Tran. Math. Software (TOMS) 32 (2) (2006) 325–351.
[4] G.H. Golub, Ch. Van Loan, Matrix Computations, third ed., Johns Hopkins University Press, Baltimore, 1996.
[5] M. La Porte, J. Vignes, Étude statistique des erreurs dans l'arithmétique des ordinateurs; application au controle des resultats d'algorithmes numériques, Numer. Math. 23 (1974) 63–72.
[6] Jean Vignes, Algorithmes numériques, analyse et mise en œuvre. 2. Éditions Technip, Paris, 1980. Équations et systèmes non linéaires. [Nonlinear equations and systems], With the collaboration of René Alt and Michèle Pichat, Collection Langages et Algorithmes de l'Informatique.
[7] A. Frommer, Proving conjectures by use of interval arithmetic, in: U. Kulisch, et al. (Eds.), Perspectives on Enclosure Methods. SCAN 2000, GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics, Univ. Karlsruhe, Germany, September 19–22, 2000, Springer, Wien, 2001.
[8] W. Tucker, The Lorenz attractor exists, C. R. Acad. Sci., Paris, Sér. I, Math. 328 (12) (1999) 1197–1202.
[9] J.M. Muller, N. Brisebarre, F. de Dinechin, C.P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, S. Torres, Handbook of Floating-Point Arithmetic, Birkhäuser, Boston, 2010.
[10] ANSI/IEEE 754-1985: IEEE standard for binary floating-point arithmetic. New York, 1985.
[11] ANSI/IEEE 754-2008: IEEE standard for floating-point arithmetic. New York, 2008.
[12] T. Ogita, S.M. Rump, S. Oishi, Verified solution of linear systems without directed rounding, Technical Report 2005-04, Advanced Research Institute for Science and Engineering, Waseda University, Tokyo, Japan, 2005.
[13] K. Ozaki, T. Ogita, S. Miyajima, S. Oishi, S.M. Rump, A method of obtaining verified solutions for linear systems suited for Java, J. Comput. Appl. Math. (JCAM) 199 (2) (2006) 337–344. (Special issue on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN 2004)).
[14] S.M. Rump, Error estimation of floating-point summation and dot product, BIT 51 (1) (2012) 201–220.
[15] J. Demmel, Y. Hida, Accurate and efficient floating point summation, SIAM J. Sci. Comput. (SISC) 25 (2003) 1214–1248.
[16] N.J. Higham, The accuracy of floating point summation, SIAM J. Sci. Comput. 14 (1993) 783–799.
[17] M. Malcolm, On accurate floating-point summation, Comm. ACM 14 (11) (1971) 731–736.
[18] A. Neumaier, Rundungsfehleranalyse einiger Verfahren zur Summation endlicher Summen, Zeitschrift für Angew. Math. Mech. (ZAMM) 54 (1974) 39–51.
[19] D.M. Priest, On properties of floating-point arithmetics: numerical stability and the cost of accurate computations, Ph.D. Thesis, Mathematics Department, University of California at Berkeley, CA, 1992. ftp://ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z.
[20] K. Ozaki, T. Ogita, S.M. Rump, S. Oishi, Accurate matrix multiplication by using level 3 BLAS operation, in: Proceedings of the 2008 International Symposium on Nonlinear Theory and its Applications, NOLTA'08, IEICE, Budapest, Hungary, 2008, pp. 508–511.
[21] S.M. Rump, T. Ogita, S. Oishi, Accurate floating-point summation part I: faithful rounding, SIAM J. Sci. Comput. 31 (1) (2008) 189–224.
[22] S.M. Rump, Ultimately fast accurate summation, SIAM J. Sci. Comput. (SISC) 31 (5) (2009) 3466–3502.
[23] J.R. Shewchuk, Adaptive precision floating-point arithmetic and fast robust geometric predicates, Discrete Comput. Geom. 18 (3) (1997) 305–363.
[24] Y.-K. Zhu, W. Hayes, Fast, guaranteed-accurate sums of many floating-point numbers, in: G. Hanrot, P. Zimmermann (Eds.), Proceedings of the RNC7 Conference on Real Numbers and Computers, Loria, Nancy, France, 2006, pp. 11–22.
[25] Y.-K. Zhu, J.-H. Yong, G.-Q. Zheng, A new distillation algorithm for floating-point summation, SIAM J. Sci. Comput. 26 (6) (2005) 2066–2078.
[26] G. Zielke, V. Drygalla, Genaue Lösung linearer Gleichungssysteme, GAMM Mitt. Ges. Angew. Math. Mech. 26 (2003) 7–108.
[27] X. Li, J. Demmel, D. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Kang, A. Kapur, M. Martin, B. Thompson, T. Tung, D. Yoo, Design, implementation and testing of extended and mixed precision BLAS, ACM Trans. Math. Software 28 (2) (2002) 152–205.
[28] XBLAS: a reference implementation for extended and mixed precision BLAS. http://crd.lbl.gov/~xiaoye/XBLAS/.
[29] D.H. Bailey, H. Yozo, X.S. Li, B. Thompson, ARPREC: an arbitrary precision computation package, Technical Report LBNL-53651, Lawrence Berkeley National Laboratory, Berkeley, CA, 2002.
[30] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, P. Zimmermann, MPFR: a multiple-precision binary floating-point library with correct rounding, ACM Trans. Math. Software 33 (2) (2007) article 13.
[31] G.E. Forsythe, M.A. Malcolm, C.B. Moler, Computer Methods for Mathematical Computations, Prentice Hall, Englewood Cliffs, NJ, 1977.
[32] N.J. Higham, Experience with a matrix norm estimator, SIAM J. Sci. Statist. Comput. (SISC) 11 (1990) 804–809.
[33] L.N. Trefethen, R. Schreiber, Average-case stability of gaussian elimination, SIAM J. Matrix Anal. Appl. 11 (3) (1990) 335–360.
[34] D. Viswanath, L.N. Trefethen, Condition numbers of random triangular matrices, SIAM J. Matrix Anal. Appl. 19 (2) (1998) 564–581.
[35] S. Oishi, K. Tanabe, T. Ogita, S.M. Rump, Convergence of Rump's method for inverting arbitrarily ill-conditioned matrices, J. Comput. Appl. Math. 205 (1) (2007) 533–544.
[36] S.M. Rump, Inversion of extremely ill-conditioned matrices in floating-point, Japan J. Indust. Appl. Math. (JJIAM) 26 (2009) 1–29.
[37] S.M. Rump, A class of arbitrarily ill-conditioned floating-point matrices, SIAM J. Matrix Anal. Appl. (SIMAX) 12 (4) (1991) 645–653.
[38] T. Nishi, T. Ogita, S. Oishi, S.M. Rump, A method for the generation of a class of ill-conditioned matrices. in: 2008 International Symposium on Nonlinear Theory and its Applications, NOLTA'08, Budapest, Hungary, September 7–10, 2008, pp. 53–56.
[39] T. Nishi, S.M. Rump, S. Oishi, On the generation of very ill-conditioned integer matrices, Nonlinear Theory Appl. (NOLTA), IEICE 2 (2) (2011) 226–245.

[40] P. Langlois, Accurate algorithms in floating-point arithmetic, in: Lecture at the 12th GAMM-IMACS International Symposion on Scientific Computing, SCAN, in: Computer Arithmetic and Validated Numerics, IEEE, Duisburg, 2006.
[41] T. Ogita, S.M. Rump, S. Oishi, Accurate sum and dot product, SIAM J. Sci. Comput. (SISC) 26 (6) (2005) 1955–1988.
[42] D.E. Knuth, The Art of Computer Programming: Seminumerical Algorithms, Vol. 2, Addison Wesley, Reading, Massachusetts, 1969.
[43] T.J. Dekker, A floating-point technique for extending the available precision, Numer. Math. 18 (1971) 224–242.
[44] I. Babuśka, Numerical stability in mathematical analysis, Inform. Process. 68 (1969) 11–23.
[45] A. Neumaier, Introduction to Numerical Analysis, Cambridge University Press, 2001.
[46] T. Yamamoto, Error bounds for approximate solutions of systems of equations, Japan J. Appl. Math. 1 (1984) 157–171.
[47] L. Collatz, Einschließungssatz für die charakteristischen Zahlen von Matrizen, Math. Z. 48 (1942) 221–226.
[48] S.M. Rump, INTLAB - Interval Laboratory, Version 6, pp. 1998–2011. http://www.ti3.tu-harburg.de/rump.
[49] P. Koev, Accurate computations with totally nonnegative matrices, SIAM J. Matrix Anal. Appl. 29 (2007) 731–751.
[50] P. Alonso, J. Delgado, R. Gallego, J.M. Peña, Growth factors of pivoting strategies associated to Neville Elimination, J. Comput. Appl. Math. 235 (7) (2011) 1755–1762.
[51] F.M. Dopico, J.M. Molera, Accurate solution of structured linear systems via rank-revealing decompositions, IMA J. Numer. Anal. 32 (2012) 1096–1116.
[52] N. Castro-González, J. Ceballos, F. Dopico, J. Molera, Multiplicative perturbation theory and accurate solution of least squares problems, 2012 (submitted for publication).
[53] J. Demmel, Accurate SVDs of structured matrices, SIAM J. Math. Anal. (SIMA) 21 (2) (1999) 562–580.
[54] J.B. Demmel, I. Dumitriu, O. Holtz, P. Koev, Accurate and efficient expression evaluation and linear algebra, Acta Numer. 2008 (2008) 87–145.
[55] J. Boothroyd, Algorithm 274: generation of Hilbert derived test matrix, Comm. ACM 9 (1) (1966) 11.
[56] B. Beckermann, The condition number of real Vandermode, Krylov and positive definite Hankel matrices, Numer. Math. 85 (4) (2000) 553–577.
[57] W. Gautschi, Optimally scaled and optimally conditioned Vandermonde and Vandermonde-like matrices, BIT 51 (1) (2011) 103–125.