Design of Numerical Software

1 Differences Between Numerical Software and Non-numerical Ones

Due to the limitation on the precision of computer numbers approximating real numbers, finite precision computation is different from conventional mathematics. Well-known examples are:

$(a + b) + c = a + (b + c)$?

$x_1 + x_2 + \cdots + x_n = x_n + \cdots + x_2 + x_1$?

If $x > 0$, $1.0 + x > 1.0$?

If $x \neq y$, $x - y \neq 0$?

In this section, we use examples to illustrate two issues: specification and loop termination.

1.1 Specification

A specification of software describes the intended relationship between a program's input and its output. This relationship should be precise and complete to help testing. However in numerical software design it is very hard to write out precise and complete specification for many computational problems. Even when specifications exist, they may be difficult to determine. In this section, we first give an example to illustrate the difficulties in numerical software specification. Then we use another simple example to show how we might go about a specification for numerical software.

This first example is due to [8].

Suppose we use the following FORTRAN 77 program to generate random numbers:

```
FUNCTION RAN()
SAVE K
DATA K / 100001 /
K = MOD(K*125,2796203)
RAN = REAL(K) / 2796203.0
RETURN
END
```

How do we write out a specification for this program? Obviously, any value returned by RAN lies between 0 and 1. But this is not complete and precise. If there is a mistake causing 100001 to be replaced by 10001, can we find

out this mistake by checking the specification and testing the program? A precise and complete specification might exist, it is certainly hard to determine.

Does that mean we have to give up on specifying numerical software? No. In the following example, we show how we might go about a specification regarding numerical issues. Consider the function Sqrt(a) which computes the square root of a nonnegative a. Let us start with a mathematical specification:

$$\text{Sqrt}(a) = \sqrt{a}, \quad a \geq 0.$$

In computer arithmetic, however, a must be a number representable on the machine. Since a can be the result of previous operations, it can be any of ± 0, $\pm\infty$, finite normal floating-point number, and NaN. Also, the result can be any of those values. For $+0$, $+\infty$, any positive finite number, and NaN, the result is obvious. What should the function do when $a < 0$? Clearly, it is undesirable to print an error message and stop execution. Here, NaN can help. What is Sqrt(-0)? We choose -0 because if $a < 0$ and underflowed to -0 in previous operations, then log(Sqrt(a)) will give NaN. If $a > 0$ and underflowed to $+0$ in previous operations, then log(Sqrt(a)) will give $-\infty$. Thus we refine the domain of the function and get the following specification according to the floating-point number system:

				$a > 0$		
	$a < 0$	$a = -0$	$a = 0$	$a \neq \infty$	$a = \infty$	$a = \text{NaN}$
$\text{Sqrt}(a) =$	NaN	-0	0	$\sqrt{a}$	∞	NaN

Now we consider the effect of finite precision computation. The exact $\sqrt{a}$ may require infinite precision, for example $\sqrt{2}$. It is impossible to compute $\sqrt{a}$ exactly in general. The best we can do is to get the floating-point number closest to the exact value. This is what IEEE standard requires. Using the notation fl(x), which denotes the computer number obtained by rounding a real number x. Thus we have the following modified table:

				$a > 0$		
	$a < 0$	$a = -0$	$a = 0$	$a \neq \infty$	$a = \infty$	$a = \text{NaN}$
$\text{Sqrt}(a) =$	NaN	-0	0	$\text{fl}(\sqrt{a})$	∞	NaN

In summary, we start with a mathematical specification. Then we refine the domain based on floating-point number system and consider finite precision computation. In this simple example, we are able to implement the function to satisfy the specification. Specifically, we can compute Sqrt(a) so accurate that it equals fl($\sqrt{a}$). In most cases, however, we are not so lucky.

Variable	Description
$\&a\&$: real	left endpoint of initial interval
$\&b\&$: real	right endpoint of initial interval
$\&F\&$: string	name of a function subprogram defined on the interval $[a, b]$.
$\%z\%$: real	zero of $F(x)$ in $[a, b]$

Table 1: Environment variables in Zeroin function.

Also, sometimes it is unnecessarily expensive to compute the result to such high accuracy. The following example gives a more general case where the error in the result can not be guaranteed to be accurate to the machine precision.

Suppose we are to specify a function Zeroin(a, b, F) which finds a zero of a continuous function $F(x)$ given an interval $[a, b]$ such that $F(a)F(b) \leq 0$. In this example, we use the tabular notation in documentation introduced by Parnas [9, 10, 6]. The documentation includes a description of the environment that identifies a set of quantities of concern to software users and associates each one with a mathematical variable. Table 1 shows the environment variables of the function Zeroin(a, b, F). The notations in Table 1:

- $\&x\&$: x is a monitored variable
- $\%x\%$: x is a controlled variable

As in the previous example, we start with a mathematical description of the function:

$z = \text{Zeroin}(a, b, F)$
$R(,) = (F(a)F(b) \leq 0) \Rightarrow ($

z |	$(a \leq z \leq b) \ \wedge \ (F(z) = 0)$

$) \wedge NC(a, b, F)$

This description says that if $F(a)F(b) \leq 0$ is true, then the returned z satisfies $a \leq z \leq b$ and $F(z) = 0$. If $F(a)F(b) \leq 0$ is false, then the returned z can be anything, but a, b, and F are unchanged after the execution of Zeroin.

Then, we consider the effects of the finite precision computation. Is it alway possible to find the exact zero? Apparently not, because the exact solution may not be representable in the underlying computer number system. So, we require that the computed result be accurate to certain degree,

a specified tolerance. But, if the given tolerance *tol* is too small, it is possible that there are no floating-point numbers in $[z - tol, z + tol]$ where z is the exact solution. Any program attempts to find a computed solution with such small uncertainty *tol* is bound to fail. In that case, the best we can do is to find a floating-point number closest to the exact solution. In particular, if $tol = 0$, the function returns a floating-point number closest to the exact solution z. Introducing variables $\%\widehat{z}\%$, the computed solution, and $\&tol\&$, the tolerance, we refine the mathematical specification and obtain the following specification.

$\widehat{z} = \text{Zeroin}(a, b, tol, F)$
$R(,) = (F(a)F(b) \leq 0) \wedge (z|\ (F(z) = 0) \wedge (a \leq z \leq b)) \wedge (tol \geq 0) \Rightarrow ($

	$tol > \lvert zu \rvert$	$tol \leq \lvert zu \rvert$
$\widehat{z}'$ |	$(a \leq \widehat{z}' \leq b) \wedge$ $(\lvert\widehat{z}' - z\rvert \leq tol)$	$(a \leq \widehat{z}' \leq b) \wedge$ $(\lvert\widehat{z}' - z\rvert \leq \lvert zu\rvert)$

$) \wedge NC(a, b, tol, F)$

Recall that u is the unit of roundoff, a machine parameter. Is the above specification precise and complete? What if zu is zero or underflows? What if the computed $F(\widehat{z}')$ is zero but $\widehat{z}'$ is not the closest to z which is a real number? Actually, there may be more than one floating-point number satisfying $F(\widehat{z}) = 0$.

1.2 Loop Termination

Loop termination is an important issue in software design. However, a mathematically terminating loop may not terminate in the presence of imperfect arithmetic. The following program finds a zero of a given function $f(x)$ using the bisection method.

```
% This program computes a zero z of a given function f(x)
% continous on the interval [a,b] assuming f(a)*f(b)<0.
% The computed result is accurate to a given tolerance tol.
%
fb = f(b);
while abs(a - b) > tol
   mid = (a + b)/2.0;
   fmid = f(mid);
   if fb*fmid > 0
      b = mid; fb = fmid;
   else
      a = mid;
```

```
   end
end
z = (a + b)/2.0;
```

If $f(x) = (x - 8 \times 10^8)^2 - 2 \times 10^{12}$, an exact zero is

$$z = 8.0141421356237310 \cdots \times 10^8.$$

If we choose

$$a = 8.01 \times 10^8, \quad b = 8.02 \times 10^8, \quad \text{and} \quad tol = 0.5 \times 10^{-7},$$

the while loop in the above program does not terminate in double precision arithmetic. Because the two neighboring floating-point numbers of z are

$$z_a = 8.01414213562373042 \cdots \times 10^8 = 1.7e24e22c7fbd7_{\text{hex}} \times 2^{29}$$

and

$$z_b = 8.01414213562373161 \cdots \times 10^8 = 1.7e24e22c7fbd8_{\text{hex}} \times 2^{29}.$$

The distance between the two consecutive floating-point numbers around the exact solution z is $2^{-23} \approx 1.2 \times 10^{-7}$, which is greater than the tolerance. Specifically, we eventually get $a = z_a$, $b = z_b$ and $mid = (a + b)/2$, which is rounded to b. The while loop never terminates.

On the other hand, a mathematically nonterminating loop may terminate in the presence of imperfect arithmetic. For example, the following program determines the machine precision t (in terms of binary digits).

```
eps = 1.0;
t = 0;
while (1.0 + eps) > 1.0
   eps = eps/2.0;
   t = t + 1;
end
```

Mathematically, the while loop in the above program is nonterminating, but, due to the finite precision, it terminates, even way before `eps` reduces to the smallest positive floating-point number.

2 Guidelines for Numerical Programming

Based on the principles discussed in Chapter 1, we present the following guidelines for writing numerical programs [5].

1. **Do not reinvent wheels.** Use built-in library functions and high quality packages such as LAPACK whenever possible.

2. **Take precautions when checking whether two floating-point numbers are exactly equal.** Two floating-point numbers are rarely equal exactly. Instead, we usually check whether two floating-point numbers are close enough using a tolerance.

3. **Take precautions to avoid unnecessary overflow and underflow.** When applying some straitforward mathmetical formulas in our programs, we often get the situations of overflow or underflow. Actually some of these circumstances can be avoided by changing the algorithm. We regard these kinds of overflow and underflow as unnecessary overflow and underflow. This is what we should take precausion to avoid this circumstance when we design an numerical software.
   There are many different ways to avoid underflow, overflow or both for specific problem. Scaling is the most common way. Here is an example for avoiding unnecessary overflow or underflow by scaling.

   Example 1 *The evaluation of a vector 2-norm,* $||x||_2 = \sum_i (|x_i|^2)^{1/2}$.

   A straightforward implementation would be

   ```
   s = 0;
   for i=1:n
      s = s + x(i)*x(i);
   end
   return sqrt(s).
   ```

   For about half of all machine numbers x, x^2 either underflow or overflow. The overflow or underflow can be avoided by the following algorithm:

   ```
   t = max(|x(i)|);
   s = 0;
   for i=1:n
   ```

```
   s = s + (x(i)/t)*(x(i)/t);
end
return t*sqrt(s).
```

However, this algorithm goes through the array twice and requires more computation than the straightforward version. Especially, division is usually many times more expensive than either addition or multiplication. It is a challenge to develop an algorithm satisfying the following criteria [1]:

(a) *Reliability*: It must compute the answer accurately, i.e., nearly all the computed digits must be correct, unless the answer is outside the range of normalized floating-point numbers.

(b) *Efficiency*: It must be nearly as fast as the straightforward algorithm in most cases.

(c) *Portability*: It must work on any "reasonable" machines, possibly including ones not running IEEE arithmetic. This means it may not cause an error condition, unless $\|x\|_2$ is (nearly) larger than the largest floating-point number.

4. **Try to avoid subtracting quantities contaminated by error**. When two very closed numbers are subtracted, cancellation occurs. There are two kinds of cancellations, one is catastrophic cancellation, and the other is benign cancellation. (see the analysis in 1.6).

5. **Minimize the size of intermediate quantities relative to the final solution**. The large intermediate quantities will lead to the final result with ill-effect of subtractive cancellation and lose some origional information from initial data because the intermediate quantities are so big that the system ignores the relatively small data with information comparision with large intermediate quantities.

   The following example makes this situation clear to us.

   Example 2 *Given numbers $x_1, x_2, ..., x_n$, this algorithm computes $S_n = \sum_{i=1}^{n} x_i$.*

   Let $S = x_1, x_2, ..., x_n$.
   while S contains more than one element
   Remove two numbers x and y from S;

and add their sum $x + y$ to S

end

Assign the remaining element of S to S_n.

We express the ith execution of the repeat loop as $T_i = x_{i_1} + y_{i_1}$. The computed sums satisfy

$$\widehat{T}_i = \frac{x_{i_1} + y_{i_1}}{1 + \delta_i}, \quad |\delta_i| \le u, \ i = 1 : n - 1.$$

So overall we have the error

$$E_n := S_n - \widehat{S}_n = \sum_{i=1}^{n-1} \delta_i \widehat{T}_i \tag{1}$$

The smallest possible error bound is therefore

$$|E_n| \le u \sum_{i=1}^{n-1} |\widehat{T}_i| \tag{2}$$

So from (1) and (2) we can indicate that minimization of intermediate result will help reduce the error.

6. **It is advantageous to express update formulae as**

   $$\text{new\_value} = \text{old\_value} + \text{small\_correction}$$

   if the small correction can be computed with many correct signigicant figures. A classic example is the iterative refinement for improving the computed solution to a linear system $Ax = b$. By computing residual $r = b - Ax$ in extended precision and solving $A\Delta x = r$, a more accurate solution $x + \Delta x$ can be obtained. This procedure can be repeated to compute highly accurate solution.

   Example 3 *Estimating π by computing the perimeters of regular polygons (square, octagon, hexagon, etc.) inscribed inside a circle.*

7. **Reformulate an unstable algorithm into a stable one.** For example, Gaussian elimination without pivoting is backward unstable (ref 1.7), but pivoting produces a stable algorithm.

8. **Use well-conditioned transformations of the problem**. Similar to the sensitivity of the problem of solving linear system (ref 1.8), the sensitivity of multiplying matrix-vector depends on the condition number of the matrix. When we apply a transformation (multiply a matrix), where possible, we use well-conditioned transformation such as orthogonal transformation.

3 Exploiting IEEE Arithmetic

Although IEEE standards 754 and 854 have been established for long time, commercial language and compilers generally provide poor support compared with the most commercial floating point processors conforming to the IEEE standards. The establishment of IEEE standards yields many benefits in numerical software design, such as the uses of NaN, $\pm\infty$, the exception handling, etc.. In this part, we present some techniques of exploiting IEEE arithmetic in numerical software development. The first example illustrates how to make use of the special values. The second example shows how to design efficient numerical software by using exception handling.

The special values NaN and $\pm\infty$ defined in IEEE standard arithmetic have been intensively used in real numerical software design for solving the practical problems. Here is a simple example.

Example 4 *Suppose we wish to evaluate the dual of the vector p-norm where* $1 \leq p \leq \infty$*, that is , the q-norm, where* $p^{-1} + q^{-1} = 1$*. In MATLAB we use the expression:* $norm(x, 1/(1 - 1/p))$*, and at extreme cases of* $p = 1$ *and* ∞ *we get* $q = 1/(1 - 1/p) = \infty$ *and* 1 *correctly in IEEE arithmetic. Note that the formula* $q = p/(p-1)$ *would not work, because* ∞/∞ *evaluates to a NaN.*

Now we look at the use of exception handling in numerical software design.

There is a fact that some numerical algorithms run quickly and give right answers usually, and some algorithms run slowly and always give right answers. The right answers in our context are that algorithms are stable, or that they compute the exact answer for a problem that is a slight perturbation of its input. We will fully take advantage of this feature to achieve fast but occasionally unstable algorithms. The following paradigm will be used in our design:

1. Use the fast algorithm to compute an answer; this will usually be done stably.

2. Quickly and reliably assess the accuracy of the computed answer by using exception handling.

3. In the unlikely event the answer is not accurate enough, recompute it slowly but accurately.

There are some conditions for successfully applying this approach:

- A large difference in speed between the fast and slow algorithms
- Being able to measure the accuracy of the answer quickly and reliably
- Floating point exceptions not causing the unstable algorithm to abort or run very slowly

The most important condition is the last one which means that the system must either continue past exceptions and later permit the program to determine whether an exception occurred, or else support user-level trap handling. The default response to five exceptions in IEEE is to proceed without a trap and deliver to the destination an appropriate default value. The IEEE standard defines these default values clearly. Detail about exception handling and default values see 1.3.4 .

When adopting the fast algorithm, the speed of the fast algorithm is determined by the relative speeds of

- conventional, unexceptional floating point arithmetic;
- arithmetic with NaN and $\pm\infty$ as arguments;
- testing sticky flags;
- trap handling.

In the worst case all the things are very slow except conventional, unexceptional floating point arithmetic, we have to change the implementation to avoid all exceptions which would be terribly slow. In our discussion we assume that

1. user-defined trap handlers are not available,
2. testing sticky flags is expensive that should avoid to test it frequently,
3. arithmetic with NaN and $\pm\infty$ is reasonably fast.

Now we concentrate on how IEEE exception handling can be used to design a fast algorithm. We will use an example of computing a condition estimation to illustrate the ideas from [2]. When solving an n-by-n linear system $Ax = b$, we wish to computer a bound on the error $x_{\text{computed}} - x_{\text{true}}$. We will measure the error using either the one-norm $||x||_1 = \sum_{i=1}^{n} |x_i|$, or the infinity norm $||x||_\infty = \max_i |x_i|$. Then the usual error bound is

$$||x_{\text{computed}} - x_{\text{true}}||_1 \leq k_1(A)p(n)\epsilon\rho||x_{\text{true}}||_1$$

where $p(n)$ is a slowly growing function of n (usually about n), ϵ is the machine precision, $k_1(A)$ is the condition number of A, and ρ is the pivot growth factor, see detail in [5]. The condition number is defined as $k_1(A) = ||A||_1 \, ||A^{-1}||_1$, where $||B||_1 \equiv \max_{1 \le j \le n} \sum_{i=1}^{n} |b_{ij}|$. Since computing A^{-1} costs more than solving $Ax = b$, we prefer to estimate $||A^{-1}||_1$ inexpensively from A's LU factorization; this is called condition estimation.

The algorithm is derived from a convex optimization approach, and is based on the observation that the maximal value of the function $f(x) = ||Bx||_1 / ||x||_1$ equals $||B||_1$ and is attained at one of the vectors e_j, for $j = 1, ..., n$, where e_j is the jth column of the n-by-n identity matrix.

First we introduce two algorithms for solving triangle systems of equations. The first one is simple, fast and disregards the possibility of overflow and underflow. The second scales carefully to avoid over/underflow and is used in our fast condition estimation algorithm.

Algorithm 1 *Solve a lower triangular system* $Lx = b$.

$x(1:n) = b(1:n)$
for $i = 1$ *to* n
  $x(i) = x(i)/L(i,i)$
  $x(i+1:n) = x(i+1:n) - x(i) * L(i+1:n, i)$
endfor

note: $L(i:j, k:l)$ *indicate the submatrix of* L *lying in rows* i *through* j *and columns* k *through* l *of* L. *Similarly* $L(i, k:l)$ *is the same as* $L(i:i; k:l)$.

This is a much common operation and has been standardized as subroutine STRSV, one of the BLAS [3, 4, 7].

Due to the possibilities of overflow, division by zero, and invalid exceptions caused by the ill-conditioning or bad scaling of the linear systems, the LAPACK routine SGECON uses Algorithm 2 below instead of Algorithm 1 to solve the triangular systems.

Here is a brief outline of the scaling algorithm. Coarse bounds on the solution size are computed as follows. The algorithm begins by computing $c_j = \sum_{i=j+1}^{n} |L_{ij}|$, $G_0 = 1/\max_i |b_i|$, a lower bound G_i on the values of x_{i+1}^{-1} through x_n^{-1} after step i of Algorithm 1:

$$G_i = G_0 \prod_{j=1}^{i} \frac{|L_{jj}|}{|L_{ij}| + c_j},$$

and finally a lower bound g on the reciprocal of the largest intermediate or final values computed anywhere in Algorithm 1:

$$g = \min_{1 \le i \le n} (G_0, G_{i-1} \cdot \min(1, |L_{ii}|)).$$

Lower bounds on x_j^{-1} are computed instead of upper bounds on x_j to avoid the possibility of overflow in the upper bounds.

Let $UN = 1/OV$ be smallest floating point number that can safely be inverted. If $g \ge UN$, this means the solution can be computed without danger of overflow, so we can simply call the BLAS.

Algorithm 2 *Solve a lower triangular system $Lx = sb$ with scale factor $0 \le s \le 1$.*

compute g and $c_1, ..., c_{n-1}$ as described above
if $(g \ge UN)$ *then*
  call the BLAS routine STRSV
else
  $s = 1$
  $x(1:n) = b(1:n)$
  $x_{\max} = \max_{1 \le i \le n} |x(i)|$
  for $i = 1$ *to* n
    if $(UN \le L(i,i) < 1$ *and* $|x(i)| > |L(i,i)| \cdot OV)$ *then*
      $scale = 1/|x(i)|$
      $s = s \cdot scale; x(1:n) = x(1:n) \cdot scale; x_{max} = x_{max} \cdot scale$
    else if $(0 < |L(i,i)| < UN$ *and* $|x(i)| > |L(i,i)| \cdot OV)$ *then*
      $scale = ((|L(i,i)| \cdot OV)/|x(i)|)/max(1, c_i)$
      $s = s \cdot scale; x(1:n) = x(1:n) \cdot scale; x_{max} = x_{max} \cdot scale$
    else if $(L(i,i) = 0)$ *then ... compute a null vector* $x : Lx = 0$
      $s = 0$
      $x(1:n) = 0; x(i) = 1; x_{max} = 0$
    endif
    $x(i) = x(i)/L(i,i)$
    if $(|x(i)| > 1)$ *and* $c(i) > (OV - x_{max})/|x(i)|)$ *then*
      $scale = 1/(2 \cdot |x(i)|)$
      $s = s \cdot scale; x(1:n) = x(1:n) \cdot scale$
    else if $(|x(i)| \le 1$ *and* $|x(i)| \cdot c(i) > (OV - x_{max}))$ *then*
      $scale = 1/2$
      $s = s \cdot scale; x(1:n) = x(1:n) \cdot scale$
    endif

$x(i+1:n) = x(i+1:n) - x(i) \cdot L(i+1:n, i)$
$x_{max} = max_{i<j<n}|x(j)|$
endfor
endif

This algorithm 2 named SLATRS is provided in LAPACK. Now we compare the costs of Algorithm 1 and 2. Algorithm 1 costs about n^2 flops, half additions and half multiplies. In the first step of algorithm 2, computing the c_i costs $n^2/2 + O(n)$ flops. In the best case, when $g \leq UN$, it makes the overall operation count about $1.5n^2$. In the worst (and very rare) case the inner loop of Algorithm 2 will scale at each step, increasing the operation count by n^2 again, for a total of $2.5n^2$. Updating $x_{\max}$ costs another $n^2/2$ data access and comparision, which may not be cheaper than the same number of floating point operations. More important than these operation counts is that Algorithm 2 has many data dependent branches, which makes it harder to optimize on pipelined or parallel architectures than the much simple Algorithm 1.

We give Algorithm 3 which uses Algorithm 2 and is very slow because it is designed carefully to avoid over/underflow.

Algorithm 3 *This algorithm computes a lower bound γ for $||A^{-1}||_1$.*

Choose x with $||x||_1 = 1$ (e.g., $x = \frac{(1,1,\ldots,1)^{\mathrm{T}}}{n}$)
Do
solve $Ay = x$ (by solving $Lw = x$ and $Uy = w$ using Algorithm 2)
form $\xi := sign(y)$
solve $A^{\mathrm{T}}z = \xi$ (by solving $U^{\mathrm{T}}w = \xi$ and $L^{\mathrm{T}}z = w$ using Algorithm 2)
if $||z||_\infty \leq z^T x$ then
$\gamma := ||y||_1$
quit
else $x := e_j$, for that j where $|z_j| = ||z||_\infty$
while TRUE

We are going to avoid the slower Algorithm 2 by using exception handling (IEEE standard facility) to deal with ill-conditioned or bad-scaled matrices. In our algorithm we just call the BLAS routine STRSV. It has the property that overflow occurs only if the matrix is extremely ill-conditioned. In this case we detect the exceptions using the sticky exception flags, we can immediately terminate with a well-deserved estimate $RCOND = 0$. Merely replacing the triangular solver used in Algorithm 3 and inserting tests for overflow does not work, as can be seen by choosing a moderately ill-conditioned

matrix of norm near the underflow threshold; this will cause overflow while solving $Uy = w$ even though A is only moderately ill-conditioned. Therefore we modify the logic of the algorithm as follows.

Algorithm 4 *This algorithm estimates the reciprocal of the condition number* $k_1(A) = ||A||_1||A^{-1}||_1$.

Let $\alpha = ||A||_1$
RCOND is the estimated reciprocal of condition number $k_1(A)$
Call exceptionreset()
Choose x *with* $||x||_1 = 1$ $(e.g., x := \frac{(1,1...,1)^T}{n})$
Repeat
  solve $Lw = x$ *by calling STRSV*
  if (except()) then $RCOND := 0$*;quit /*$k_1(A) \geq OV/\rho$*/*
  if $(\alpha > 1)$ *then*
    if $(||w||_\infty \geq OV/\alpha)$ *then*
      solve $Uy = w$ *by calling STRSV*
      if(except()) then RCOND:=0;quit /$k_1(A) \geq OV$ */*
      else $y := y \cdot \alpha$
        if(execpt()) then $RCOND := 0$*;quit /*$k_1(A) \geq OV$ */*
      endif
    else solve $Uy = w \cdot \alpha$ *by call STRSV*
      if (except()) then $RCOND := 0$*;quit /*$k_1(A) \geq OV$ */*
    endif
  else solve $Uy = w \cdot \alpha$ *by calling STRSV*
    if (except()) then $RCOND := 0$*;quit /*$k_1(A) \geq OV$ */*
  endif
  form $\xi := sign(y)$
  solve $U^T w = \xi \cdot \alpha$ *by calling STRSV*
  if (except()) then $RCOND := 0$*,quit /*$k_1(A) \geq \frac{OV}{n^3}$ */*
  else solve $L^T z = w$ *by calling STRSV*
    if (except()) then $RCOND := 0$*;quit /*$k_1(A) \geq \frac{OV}{n^2}$ */*
  endif
  if $||z||_\infty \leq z^T x$ *then*
    $RCOND := 1/||y||_1$*;quit*
  else $x := e_j$*, where* $|z_j| = ||z||_\infty$
  endif

The behavior of Algorithm 4 is described by the following: If Algorithm 4 stops early because of an exception, then the "true rounded" reciprocal of

the condition number satisfies $RCOND \leq \frac{max(n^3,\rho)}{OV}$, where $\rho = \frac{||U||_1}{||A||_1}$ is the pivot growth factor.

In the algorithm there are seven places where exceptions may occur. We will analyze them one by one. That x is chosen such that $||x||_1 = 1$, and $||\xi||_1 = n$.

1. An excetion occurs when computing $L^{-1}x$. Since $A = LU$, $L^{-1} = UA^{-1}$, this implies

   $$OV \leq ||L^{-1}x||_1 \leq ||U||_1||A^{-1}||_1||x||_1 = \frac{||U||_1}{||A||_1}||A||_1||A^{-1}||_1 = \rho \cdot k_1(A).$$

   Therefore, $k_1(A) \geq OV/\rho$, i.e., $RCOND \leq \rho/OV$

2. An exception occurs when computing $U^{-1}L^{-1}x$ with $\alpha > 1$. Then

   $$OV \leq ||U^{-1}L^{-1}x||_1 \leq ||A^{-1}||_1 < ||A^{-1}||_1\alpha = k_1(A),$$

   so $RCOND \leq 1/OV$.

3. An exception occurs when computing $\alpha \cdot U^{-1}L^{-1}x$ with $\alpha > 1$. Then

   $$OV \leq ||\alpha U^{-1}L^{-1}x||_1 \leq k_1(A),$$

   so $RCOND \leq 1/OV$.

4. An exception occurs when computing $U^{-1}\alpha L^{-1}x$, with $\alpha > 1$ and $||L^{-1}x||_1 < \frac{OV}{\alpha}$, then

   $$OV \leq ||U^{-1}\alpha L^{-1}x||_1 \leq ||A^{-1}||_1\alpha = k_1(A),$$

   so $k_1(A) \geq OV$, i.e., $RCOND \leq 1/OV$.

5. An exception occurs when computing $U^{-1}\alpha L^{-1}x$ with $\alpha \leq 1$. Then

   $$OV \leq ||U^{-1}\alpha L^{-1}x||_1 \leq ||A^{-1}||_1\alpha = k_1(A),$$

   so $k_1(A) \geq OV$, i.e., $RCOND \leq 1/OV$.

6. An exception occurs when computing $U^{-T}\alpha\xi$. Since $A^T = U^TL^T$, $U^{-T} = L^TA^{-T}$, and $||B^T||_1 \leq n||B||_1$, we get

   $$\begin{aligned} OV &\leq ||U^{-T}\alpha\xi||_1 \leq ||L^T||_1||A^{-T}||_1\alpha||\xi||_1 \\ &\leq ||L^T||_1 \cdot n||A^{-1}||_1 \cdot \alpha \cdot ||\xi||_1 \\ &= ||L^T||_1 \cdot n \cdot k_1(A) \cdot n \\ &\leq n^3k_1(A). \end{aligned}$$

   Therefore, $k_1(A) \geq \frac{OV}{n^3}$, i.e, $RCOND \leq \frac{n^3}{OV}$.

7. An exception occurs when computing $L^{-T}U^{-T}\alpha\xi$, so

   $$OV \leq ||L^{-T}U^{-T}\alpha\xi||_1 \leq ||A^{-T}||_1\alpha||\xi||_1 \leq n||A^{-1}||_1 \cdot \alpha \cdot n = n^2k_1(A).$$

   Therefore, $RCOND \leq \frac{n^2}{OV}$.

Combining the above seven cases, we have shown that $RCOND \leq \frac{max(n^3,\rho)}{OV}$ when an exception occurs.

To compare the efficiencies of Algorithm 3 and Algorithm 4, Demmel and Li [2] rewrite several condition estimation routines in LAPACK using Algorithm 4, including SGECON for general dense matrices, SPOCON for dense symmetric positive definite matrices, SGBCON for general band matrices, and STRCON for triangular matrices, all in IEEE single precision.

The performance results are on a "fast" DECstation 5000 and "slow" DECstation 5000 (both have a MIPS R3000 chips as CPU), a Sun 4/260 (which has a SPARC chip as CPU), a DEC Alpha and CRAY-C90. The "slow" DEC 5000 correctly implements IEEE arithmetic, but does arithmetic with NaN about 80 times slower than normal arithmetic. The "fast" DEC 5000 implements IEEE arithmetic incorrectly, when the operands involve denormals or NaNs, does so at the same speed as normal arithmetic. The CRAY does not have exception handling, but the speeds can still be compared in the most common cases where no exceptions occur to see what speedup there could be if exception handling were available.

Demmel and Li reported that they ran Algorithm 3 and Algorithm 4 on a suite of well-conditioned random matrices where no exceptions occur, and no scaling is necessary in Algorithm 2. They also compared Algorithm 3 and Algorithm 4 on several intentionally ill-scaled linear systems for which some of the scalings inside Algorithm 2 have to be invoked, but whose condition numbers are still finite. The numbers in the following tables [2] are the ratios of the time spent by the old LAPACK routines using Algorithm 3 to the time spent by the new routines using Algorithm 4.

Machine	Matrix dimension n	100	200	300	400	500
DEC 5000	SGBCON $sbw = 4$	3.00	4.25	5.33	6.50	6.45
	$sbw = 0.8n$	1.57	1.46	1.55	1.56	1.67
	SGECON	2.00	1.52	1.46	1.44	1.43
	SPOCON	2.83	1.92	1.71	1.55	1.52
	STRCON	3.33	1.78	1.60	1.54	1.52
SUN 4/260	SGBCON	2.00	2.20	2.11	2.77	2.71
	SGECON	3.02	2.14	1.88	1.63	1.62
	SPOCON	5.00	2.56	2.27	2.22	2.17
	STRCON	1.50	2.00	2.30	2.17	2.18
DEC Alpha	SGBCON $sbw = 3$	2.00	2.00	8.67	8.40	9.28
	$sbw = 0.8n$	2.67	2.63	2.78	2.89	3.23
	SGECON	2.66	2.01	1.85	1.78	1.66
	SPOCON	2.25	2.46	2.52	2.42	2.35
	STRCON	3.00	2.33	2.28	2.18	2.07
CRAY-C90	SGECON	4.21	3.48	3.05	2.76	2.55

Speedups on DEC 5000/Sun 5-260/ DEC Alpha/CRAY-C90. No exceptions nor scaling occur. *sbw* stands for semi-bandwidth.

	Example 1	Example 2	Example 3
"fast" DEC 5000 speedup	2.15	2.32	2.00
"slow" DEC 5000 slowdown	11.67	13.49	9.00
SPARCstation 10 speedup	3.12	3.92	3.24

The speeds of some examples with exceptions. Matrix dimensions are 500.

The most important lesson is that well-designed exception handling permits the most common cases, where no exceptions occur, to be implemented much more quickly. This alone makes exception handling worth implementing well.

4 Estimating Errors

Since errors are unavoidable, numerical software should specify error bounds for the computed results. In this section, we discuss two methods for estimating errors in addition to the forward error analysis: running error analysis and interval analysis.

4.1 Running Error Analysis

Due to computational errors, it is necessary for numerical software to specify tolerances for the computed answers. How do we get error bounds? In Section 1.6, we described forward error analysis. In this section, we present two automated error analysis methods: Running error analysis and interval analysis.

We will use inner product to illustrate the running error analysis. First we introduce the standard models. Recall that IEEE standards require that for operations $+$, $*$, $/$, and $\sqrt{}$ the computed result be the same as if the operation were carried out exactly and then rounded. Thus we have

$$\begin{aligned} fl(x \text{ op } y) &= (x \text{ op } y)(1+\delta), \text{ for op} = +, *, / \\ \text{sqrt}(x) &= \sqrt{x}\,(1+\delta), \quad |\delta| \le u. \end{aligned}$$

Alternatively,

$$\begin{aligned} fl(x \text{ op } y) &= \frac{x \text{ op } y}{1+\delta}, \text{ for op} = +, *, / \\ \text{sqrt}(x) &= \frac{\sqrt{x}}{1+\delta}, \quad |\delta| \le u. \end{aligned}$$

Consider the inner product $s_n = x^T y$, where $x,\ y \in \mathcal{R}^n$. Let $s_i = x_1y_1 + ... + x_iy_i$ denote the ith partial sum.

Algorithm 5 *Compute inner product $x^T y$.*

```
s = 0;
for i=1 to n
  s = s + x(i)*y(i);
end
```

Writing the computed partial sums as $\hat{s}_i =: s_i + e_i$ and $\hat{z}_i = fl(x_iy_i)$, we have, using the alternative standard model,

$$\hat{z}_i = \frac{x_iy_i}{1+\delta_i} \text{ or } \hat{z}_i = x_iy_i - \delta_i\hat{z}_i, \quad |\delta_i| \le u.$$

Similarly, $(1+\epsilon_i)\hat{s}_i = \hat{s}_{i-1} + \hat{z}_i$, where $|\epsilon_i| \le u$. Consequently,

$$s_i + e_i = \hat{s}_i = \hat{s}_{i-1} + \hat{z}_i - \epsilon_i\hat{s}_i = s_{i-1} + e_{i-1} + x_iy_i - \epsilon_i\hat{s}_i - \delta_i\hat{z}_i.$$

Hence $e_i = e_{i-1} - \epsilon_i\hat{s}_i - \delta_i\hat{z}_i$, which gives

$$|e_i| \le |e_{i-1}| + u|\hat{s}_i| + u|\hat{z}_i|.$$

Since $e_0 = 0$, we have $|e_n| \le u\mu_n$, where $\mu_i = \mu_{i-1} + |\hat{s}_i| + |\hat{z}_i|$ with $\mu_0 = 0$. According to the above formula, we introduce the following algorithm

Algorithm 6 *Given $x, y \in \mathcal{R}^n$ this algorithm computes $s = fl(x^T y)$ and a number μ such that $|s - x^T y| \leq \mu$*

```
s = 0;
mu = 0;
for i=0:n
    z = x(i)*y(i);
    s = s + z;
    mu = mu + abs(s) + abs(z);
end
mu = mu*u.
```

This type of computation, where an error bound is computed concurrently with the solution, is called *running error analysis*. The paradigm is: For each step, we use the rounding error model to write

$$|(x \text{ op } y) - fl(x \text{ op } y)| \leq u\,|fl(x \text{ op } y)|,$$

which gives a bound for the error in $(x \text{ op } y)$ that is easily computed, since $fl(x \text{ op } y)$ is stored on the computer. Key features of a running error analysis are that few inequalities are involved in the derivation of the bound and that the actual computed intermediate quantities are used, enabling advantage to be taken of cancellation and zero operands. A running error bound is a posteriori and usually sharper than an a priori one. The disadvantage of running error analysis is that it introduces additional computation. Nowadays the running error analysis is a somewhat negligible practice, but it is applicable to almost any numerical algorithm.

Now we apply the forward error analysis to the inner product and compare the error bounds. Using the standard model, we have

$$\begin{aligned} \hat{s}_1 &= fl(x_1 y_1) = x_1 y_1 (1 + \delta_1) \\ \hat{s}_2 &= fl(\hat{s}_1 + x_2 y_2) = x_1 y_1 (1 + \delta_1)(1 + \delta_3) + x_2 y_2 (1 + \delta_2)(1 + \delta_3). \end{aligned}$$

To simplify the expressions, let us drop the subscripts on the δ_i and write $1 + \delta_i \equiv 1 + \delta$, we get

$$\hat{s}_3 = fl(\hat{s}_2 + x_3 y_3) = x_1 y_1 (1 + \delta)^3 + x_2 y_2 (1 + \delta)^3 + x_3 y_3 (1 + \delta)^2.$$

So overall we have

$$\hat{s}_n = x_1 y_1 (1 + \delta)^n + x_2 y_2 (1 + \delta)^n + x_3 y_3 (1 + \delta)^{n-1} + \cdots + x_n y_n (1 + \delta)^2.$$

To simplify the expressions further, we use the following result: If $|\delta_i| \leq u$ and $\rho_i = \pm 1$ for $i = 1 : n$, and $nu < 1$, then

$$\prod_{i=1}^{n}(1+\delta_i)^{\rho_i} = 1 + \theta_n \quad \text{where} \quad |\theta_n| \leq \frac{nu}{1-nu} =: \gamma_n.$$

So we obtain

$$\hat{s}_n = x_1y_1(1+\theta_n) + x_2y_2(1+\theta_n') + x_3y_3(1+\theta_{n-1}) + ... + x_ny_n(1+\theta_2).$$

The forward error bound is

$$|x^Ty - fl(x^Ty)| \leq \gamma_n \sum_{i=1}^{n} |x_iy_i| = \gamma_n |x|^T|y|.$$

This forward error bound is an a priori bound that does not depend on the actual rounding errors committed.

Example 5 *We computed 100 inner products of random vectors of size 1,500. The entries of vectors were uniformly distributed over* $[-1, 1]$. *Both running error analysis and forward error analysis were used to estimate the errors. The inner products were computed in single precision. Another set of inner products of same vectors were computed in double precision used as accurate results. Figure 1 plots three errors.*

4.2 Interval Error Analysis

Most practical problems requiring extensive numerical computation involve quantities determined experimentally by approximate measurements– very often with some estimate of the accuracy of the measured values. So a typical calculation will begin with some numbers known only to a certain number of significant digits. The uncertainty in a number can be represented by an interval.

Example 6 *If* $x = 1.234$ *is accurate to three decimal digits, then the exact value lies in the interval* $[1.225, 1.235]$.

Thus when we operate on x and y represented by $[a, b]$ and $[c, d]$ respectively, we would like to operate on the intervals and get another interval $[u, v]$ so that x op y lies in this interval. We need to define operations on intervals $[a, b]$ op $[c, d] = [u, v]$. For elementary functions, we have

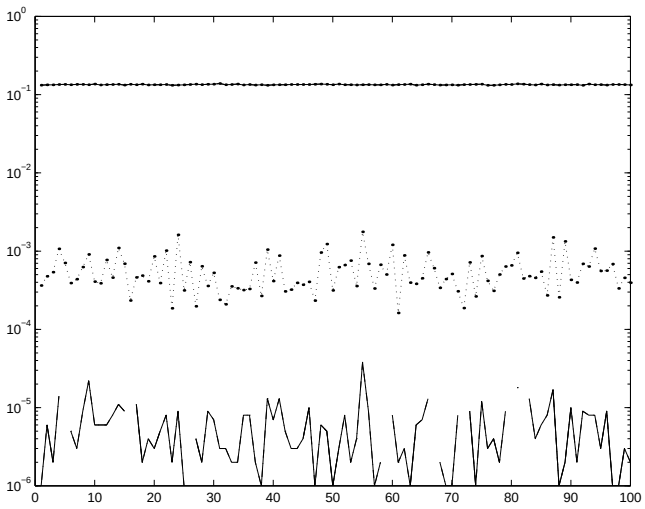

Figure 1: The top curve shows the errors estimated by forward error analysis, the midle curve shows the errors estimated by running error analysis, and the bottom curve is the actual errors

$[a,b]+[c,d]=[a+c,b+d]$,
$[a,b]-[c,d]=[a-d,b-c]$,
$[a,b]*[c,d]=[\min(ac,ad,bc,bd),\max(ac,ad,bc,bd)]$,
$[a,b]/[c,d]=[a,b]*[1/d,1/c],\quad 0\notin[c,d]$.

How can interval arithmetic be used in estimating rounding errors? Suppose we begin with two floating-point numbers x and y. Denote $z=x+y$ and $\hat{z}=fl(x+y)$. We know that the exact result lies in the interval $[\hat{z}-u,\hat{z}+u]$ where u is the unit of round off. Then in the subsequent operations on $\hat{z}$, we apply the operations on the interval $[\hat{z}-u,\hat{z}+u]$. Finally, we get an interval and guarantee that the final results lies in the interval. Interval analysis is controversial because narrow intervals can not be guaranteed. One reason is that when dependencies occur in a calculation, in the sense that a variable appears more than once, final interval lengths can be pessimestic.

Example 7 *Suppose* $x\in[-2,1]$*, then we know that* $x*x\in[1,4]$*. However, the interval arithmetic gives*

$$[-2,1]*[-2,1]=[\min(4,-2,1),\max(4,-2,1)]=[-2,4]$$

which is much wider than $[1,4]$*.*

Consider the inner product example. The above example shows that interval analysis would give large error estimate, especially when x_i are around zero. In general, suppose $x_i \in [\widehat{x}_i - \epsilon_i, \widehat{x}_i + \epsilon_i]$, for $\epsilon_i > 0$. The interval arithmetic model gives $x_i^2 \in [m_i, M_i]$ where $m_i = \min((\widehat{x}_i - \epsilon_i)^2, \widehat{x}_i^2 - \epsilon_i^2, (\widehat{x}_i + \epsilon_i)^2)$ and $M_i = \max((\widehat{x}_i - \epsilon_i)^2, \widehat{x}_i^2 - \epsilon_i^2, (\widehat{x}_i + \epsilon_i)^2)$. However, we know that $x_i^2 \in [\min((\widehat{x}_i - \epsilon_i)^2, (\widehat{x}_i + \epsilon_i)^2), \max((\widehat{x}_i - \epsilon_i)^2, (\widehat{x}_i + \epsilon_i)^2)]$. If $\widehat{x}_i^2 - \epsilon_i^2 < (\widehat{x}_i - \epsilon_i)^2$, equivalently, $x_i < \epsilon_i$, the interval arithmetic model gives $\sum x_i^2 \in [-(\sum \epsilon_i^2 - \sum x_i^2), \sum(\widehat{x}_i + \epsilon_i)^2]$, which is wider than $[\sum(\widehat{x}_i - \epsilon_i)^2, \sum(\widehat{x}_i + \epsilon_i)^2]$.

4.3 Conclusion

About error estimate, Wilkinson [11, Page 567] says

> There is still a tendency to attach too much importance to the precise error bounds obtained by an a priori error analysis. In my opinion, the bound itself is usually the least important part of it. The main object of such an analysis is to expose the potential instabilities, if any, of an algorithm so that hopefully from the insight thus obtained one may be led to improved algorithms. Usually the bound itself is weaker than it might have been because of the necessity imposed by expressing the errors in terms of matrix norms. A priori bounds are not, in general, quantities that should be used in practice. Practical error bounds should usually be determined by some form of a posteriori error analysis, since this takes full advantage of the statistical distribution of rounding errors and of any special features, such as sparseness, in the matrix.

References

[1] James W. Demmel. *Applied Numerica Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.

[2] James W. Demmel and Xiaoye Li. Fast Numerical Algorithms via Exception Handling. *IEEE Trans. Comput.*, 43(8):983-992, 1994.

[3] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Maths. Soft.*, 16(1):1-17, March 1990.

[4] J. Dongarra, J. Du Croz, S. Hammarling, and Richard J. Hanson. An Extended Set of Fortran Basic Linear Algebra Subroutines. *ACM Trans. Math. Soft.*, 14(1):1-17, March 1988.

[5] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithm.* Society for Industrial and Applied Mathematics, 1996.

[6] M. Iglewski, J. Madey and D.L. Parnas. Documentation Paradigms. *CRL Report 270*, Mcmaster University, CRL, TRIO (Telecommunications Research Institute of Ontario), July 1993.

[7] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5:308-323, 1979.

[8] Webb Miller. *The Engineering of Numerical Software.* Prentice-Hall, NJ, 1984.

[9] David Lorge Parnas. Tabular representation of relations. *CRL Report 260*, Mcmaster University, TRIO (Telecommunications Research Institute of Ontario), Oct. 1992.

[10] D.L. Parnas and J. Madey. Functional documentation for computer systems enginerring (version 2). *CRL Report 237*, Mcmaster University, TRIO (Telecommunications Research Institute of Ontario), Sept. 1991

[11] J.H. Wilkinson. Modern Error Analysis. *SIAM Review.* 13:4, 548–568.