Robust Triangular Solves for Use in Condition Estimation [*]

Edward Anderson
Cray Research Inc.
655F Lone Oak Drive
Eagan, MN 55121

August 16, 1991

Abstract

Fortran codes are presented for solving a triangular system when the triangular matrix is badly scaled or badly conditioned. These subroutines incorporate scaling to prevent overflow and thus are more robust than their counterparts STRSV, STPSV, and STBSV from the Level 2 BLAS. Solving badly conditioned triangular systems arises in condition estimation when the procedure developed by Hager and Higham is used to estimate the norm of A^{-1} from the triangular factorization of A. We discuss situations in which scaling is necessary to prevent overflow and give an example of how our routines are used in the LAPACK condition estimators.

1 Introduction

The condition estimators in the LAPACK software package are based on the estimate described by Hager [3] and Higham [4]. The norm of the matrix A is computed by conventional means, an estimate is obtained for the 1-norm (or infinity-norm) of its inverse, and the reciprocal condition estimate is computed as RCOND = $1/(\|A\|\|A^{-1}\|)$. When used to estimate $\|A^{-1}\|$, Higham's version of Hager's norm estimation routine is a multi-step procedure that uses a reverse communication interface, asking the higher level routine to supply $A^{-1}x$ or $A^{-T}x$ for a given vector x between steps. This design is very convenient for the development of a software

[*] This work was performed while the author was employed by the University of Tennessee and was supported by NSF Grant No. ASC-8715728.

package like LAPACK, because $A^{-1}x$ is computed by different means in different contexts, depending on the storage used for A and the triangular factorization that has been applied.

When solving linear systems of the form $Ax = b$, we typically begin with a triangular factorization of the form $PAQ = LDU$, where P and Q are permutation matrices, L is lower triangular, D is diagonal, and U is upper triangular. This formulation includes the LU factorization, for which L is unit lower triangular and Q and D are identity matrices, the Cholesky factorization of a symmetric positive definite matrix, for which $P = Q = I$ and $U = L^T$, and the Bunch-Kaufman diagonal pivoting factorization of a symmetric indefinite matrix, for which $Q = P^T$, $U = L^T$, and D is block diagonal with 1×1 and 2×2 diagonal blocks.

For simplicity, we assume the triangular factorization has the form $A = LU$. The norm estimation routine (called SONEST in [4] or SLACON in the LAPACK library) asks that we supply $A^{-1}x$ (if the integer parameter KASE = 1) or $A^{-T}x$ (if KASE = 2) for a given vector x on an intermediate return. If we are computing the norm of A^{-1}, we replace x with $A^{-1}x = U^{-1}L^{-1}x$ by computing

$$x \leftarrow L^{-1}x; \;\; x \leftarrow U^{-1}x$$

or we replace x with $A^{-T}x = U^{-T}L^{-T}x$ by computing

$$x \leftarrow U^{-T}x; \;\; x \leftarrow L^{-T}x.$$

These operations require solving a triangular system involving a unit or non-unit upper or lower triangular matrix.

Triangular solves are available from the Level 2 BLAS routines STRMV, STPMV, and STBMV [2], but we found these routines unsuitable for condition estimation. The condition estimator should logically be one of the most robust pieces of software in a software package, because it will often be called to estimate the condition number of a matrix that is already suspected to be ill-conditioned. Our early efforts using STRSV would cause a floating-point exception to occur if the matrix were badly scaled or badly conditioned, obviously not the best way to indicate ill-conditioning!

In this paper we describe the subroutines we have written to take the place of the Level 2 BLAS solves in the condition estimation routines. These auxiliary routines, named SLATRS, SLATPS, and SLATBS for the three types of triangular matrix storage in the REAL data type, solve the scaled problem $Tx = sb$ or $T^Tx = sb$ for x, avoiding overflow by an appropriate choice of s. We give specific examples illustrating features of the code and analyze the additional overhead in using the robust solves in place of the BLAS. We conclude with an example of how SLATRS is used in one of the LAPACK condition estimation routines.

2 Hager's algorithm for estimating the norm of a matrix

In this section, we briefly describe Hager's method for estimating the 1-norm of a matrix. The main application is in estimating the norm of $B = A^{-1}$ when B has not been explicitly computed. For further details, refer to [3] or [4].

The 1-norm of an n by n matrix B is defined in terms of the vector 1-norm $\|x\|_1 = \sum_{i=1}^{n} |x_i|$ as

$$\|B\|_1 = \max_{x \neq 0} \frac{\|Bx\|_1}{\|x\|_1}.$$

A well-known property is that $\|B\|_1$ is equal to the maximum column sum:

$$\|B\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{n} |b_{i,j}|, \tag{2.1}$$

hence the maximum in (2.1) is attained at $x = \pm e_i$ for some $1 \leq i \leq n$, where e_i is the i^{th} column of the identity matrix. From the optimization point of view, $\|B\|_1$ is the global maximum of the convex function

$$f(x) = \|Bx\|_1$$

over the convex set

$$S = \{x \in R^n : \|x\|_1 \leq 1\}.$$

For a convex function f on a convex set S and a point $x \in S$, a vector z such that

$$f(y) \geq f(x) + z^T(y - x) \tag{2.2}$$

for all $y \in S$ is called a *subgradient* of f at x. Hager constructs a particular subgradient of f at x by defining a vector ξ such that

$$\xi_i = \begin{cases} 1 & \text{if } \sum_{j=1}^{n} b_{i,j} x_j \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

and setting $z = B^T \xi$. This choice of ξ allows us to express $f(x) = \|Bx\|_1$ without using absolute values:

$$f(x) = \sum_{i=1}^{n} \xi_i \left(\sum_{j=1}^{n} b_{i,j} x_j \right).$$

Hence $f(x) = \xi^T Bx = z^T x$, and the proof that z is a subgradient follows by showing

$$f(y) \geq \xi^T By.$$

The subgradient at x is unique only if $f(x)$ is differentiable at x, i.e., if Bx has no zero components. If this is the case, the subgradient we construct is the gradient of f at x.

Hager's algorithm is an iterative procedure which starts at a point on the boundary of S and moves between vertices $\{e_i\}$. First, the subgradient z of f at x is constructed and the maximum element of z is found:

$$|z_j| = \max_{1 \le i \le n} |z_i|.$$

If $|z_j| \le z^T x = \|Bx\|_1$, the procedure stops; if f is differentiable at x, then x is a local maximum of f. If $|z_j| > z^T x$, then one of the points $y = e_j$ or $y = -e_j$ makes the expression $z^T(y - x)$ in (2.2) positive, and therefore $f(y) > f(x)$. Since $f(e_j) = f(-e_j)$, we replace x by e_j and repeat. Since f increases at each step, no vertex e_j is ever repeated, and the algorithm terminates in at most $n + 1$ steps.

Higham [4] demonstrated that Hager's algorithm does much better than $n + 1$ iterations on average, usually converging in two steps. He also constructed several examples for which Hager's algorithm does not return a good estimate, usually because of difficulty in determining if a local maximum is also a global maximum. We have used Higham's subroutines SONEST and CONEST (renamed SLACON and CLACON), which do at least two iterations of Hager's algorithm and never more than five.

In computing the subgradient z, we must be able to multiply by both B and B^T. Multiplication by B is required to form Bx, so that we can then construct ξ. Multiplication by B^T is required to form $z = B^T\xi$. If $B = A^{-1}$, the most important application of this algorithm, then we must compute

$$y = A^{-1}x; \;\; \xi = \text{sign}(y); \;\; z = A^{-T}\xi,$$

which typically involves solving triangular systems for y and z using a previously computed triangular factorization for A.

3 Solving a triangular system with scaling

A robust triangular system solver is needed in the condition estimator when trying to estimate the norm of A^{-1} and in certain other applications as well. It is easy to construct examples where the triangular matrix T and the vector b in the system $Tx = b$ are reasonably well-scaled, but the vector x overflows. Scaling x, or, equivalently, solving $Tx = sb$ or $T^Tx = sb$ for a scaling factor s that is determined dynamically, takes care of most of the difficulties. But we would like to be able to identify when scaling is not required and call the Level 2 BLAS equivalent in these cases, since the BLAS have less overhead and may be much faster if a locally optimized version is available. We therefore begin by computing a bound on the solution x from the components of b, the diagonal of T, and the 1-norm of each column of T, not including the diagonal. Computing the bound is $O(n)$ except for the computation of the column norms, but once the column norms have been computed, they can be reused on subsequent calls.

3.1 The non-transposed problem

For the purpose of discussion, assume $T = L$ is lower triangular. The solution of the triangular system $Lx = b$ generally follows the basic column-oriented algorithm:

```
$x \leftarrow b$
do $i = 1, 2, \ldots, n$
    $x_i \leftarrow x_i / L_{i,i}$
    $\begin{bmatrix} x_{i+1} \\ x_{i+2} \\ \vdots \\ x_n \end{bmatrix} \leftarrow \begin{bmatrix} x_{i+1} \\ x_{i+2} \\ \vdots \\ x_n \end{bmatrix} - x_i \begin{bmatrix} L_{i+1,i} \\ L_{i+2,i} \\ \vdots \\ L_{n,i} \end{bmatrix}$        (3.1)
end do
```

Floating point errors may occur if $L_{i,i} = 0$, if $x_i/L_{i,i}$ overflows, if $x_i L_{j,i}$ overflows for any $i+1 \leq j \leq n$, or if $x_j - x_i L_{j,i}$ overflows for any $i+1 \leq j \leq n$.

The first step in SLATRS is to compute a bound on $x = L^{-1}b$ to see if the Level 2 BLAS routine STRSV can be used. Let Ω denote a machine-dependent constant near overflow which we will define later. We define bounds on the components of x after j iterations of the above loop:

$$M(j) = \text{bound on } x_j$$
$$G(j) = \text{bound on } x_i,\ j+1 <= i <= n$$

Initially, $M(0) = 0$ and $G(0) = \max(x_i,\ i = 1, ..., n)$. Then after j iterations we have

$$\begin{aligned} M(j) &\leq G(j-1)/|L_{j,j}| \\ G(j) &\leq G(j-1) + M(j) \| L_{j+1:n,j} \| \\ &\leq G(j-1)\,(1 + c_j/|L_{j,j}|) \end{aligned}$$

where c_j is greater than or equal to the infinity-norm of column j of L, not counting the diagonal. Hence

$$G(j) \leq G(0) \prod_{i=1}^{j} (1 + c_i/|L_{i,i}|)$$

and

$$M(j) \leq \frac{G(0)}{|L_{j,j}|} \prod_{i=1}^{j-1} (1 + c_i/|L_{i,i}|) .$$

The code actually computes the reciprocals of $G(j)$ and $M(j)$ to avoid overflow in forming these bounds. Since $|x_j| \leq M(j)$, we can safely use the Level 2 BLAS routine STRSV if the reciprocal of the largest $M(j)$ is larger than Ω^{-1}.

Similar bounds are computed if T is upper triangular. If we assume c_j contains the norm of column j of T, not including the diagonal, the remaining discussion applies to either the upper or lower triangular case.

The work to compute the column norms c_j is $O(n^2)$, the same order as the triangular solve, so we would like to be able to re-use this information if possible. In the non-transposed case, c_j need only be as large as the infinity-norm of the offdiagonal part of column j, but in the transposed case the 1-norm is required, and since the 1-norm is always greater than or equal to the infinity-norm, we always compute the 1-norm. In the interest of speed, we assume that the 1-norm of the offdiagonal part of each column does not overflow, and we compute these norms without any scaling if they are not already known. If any of the c_j's is greater than Ω, we compute a scaling factor for the triangular matrix

$$s_t = \Omega / \max(c_j, j = 1, \ldots, n)$$

and multiply each c_j by s_t. This allows us to assume that $c_j \leq \Omega$, which is necessary to prevent overflow in steps of the columnwise method. Note that the c_j's are scaled by s_t but T itself is not altered. Hence when the diagonal elements $T_{i,i}$ are accessed, they must be multiplied by s_t. The norms c_j are multiplied by $1/s_t$ before return and s_t is incorporated into the factor s, so s_t is only used internally.

We now consider the cases that could cause floating point errors in the columnwise method and describe the scaling used to avoid them. First, we scale the right hand side b (contained in the vector X) if the maximum element in b is greater than Ω. Subsequent steps of the solve use additional scaling as needed to guarantee that no element of X exceeds Ω.

If $|T_{i,i}| \geq \Omega^{-1}$, the reciprocal of $T_{i,i}$ can be formed without overflow, but the fraction $x_i/T_{i,i}$ may overflow if $x_i > 1$. To avoid overflow if $|T_{i,i}| > 1$ and $|x_i| > |T_{i,i}|\Omega$, we scale x by $|x_i|$ before dividing by $T_{i,i}$. An example where a simple scaling is used is in solving the equation

$$10^{-20}\, x = 10^{20}$$

The exact solution $x = 10^{40}$ overflows on a Sun ($\Omega = 3.40 \times 10^{38}$), but by scaling the right hand side, we return the solution $s = 10^{-20}$, $x = 10^{20}$.

If $0 < |T_{i,i}| < \Omega^{-1}$, then $1/|T_{i,i}| > \Omega$. Numbers less than Ω^{-1} can occur in IEEE arithmetic because denormalized numbers are permitted near underflow whose reciprocals exceed the overflow threshold. One solution would be to treat this case as if $T_{i,i}$ were zero. But if x_i is also small, $x_i/T_{i,i}$ need not overflow. More precisely, we can divide by $T_{i,i}$ without using the scaling factor s if $|x_i| \leq |T_{i,i}|\Omega$, as in the example

$$10^{-35}\, x = 10^{-20}.$$

On some machines, a divide is implemented as a reciprocal followed by a multiply, so if 10^{-30} were the smallest invertible number, overflow would occur when inverting

10^{-35} even though the result, $x = 10^{15}$, is well within the acceptable range. To get around this problem, we multiply both sides by Ω until the coefficient in front of x is greater than Ω^{-1} (usually, only one such scaling step, consisting of two multiplies, is required). In the above example, both sides are scaled by 10^{30} first, then x is computed as

$$x = 10^{10}/10^{-5} = 10^{15}.$$

If $0 < |T_{i,i}| < \Omega^{-1}$ and $|x_i| > |T_{i,i}|\Omega$, then we scale x by

$$(\min(1, |T_{i,i}|)\Omega/|x_i|)\min(1, 1/c_i),$$

which we know to be less than 1. The first part of the scaling factor assures that $|x_i|/|T_{i,i}| \leq \Omega$, and the second part assures that the SAXPY (3.1) in the basic algorithm can also be done. An example demonstrating this scaling is

$$\begin{bmatrix} 10^{-35} & \\ 10^{20} & 10^{-35} \end{bmatrix} x = \begin{bmatrix} 10 \\ 10 \end{bmatrix}.$$

Assuming $\Omega = 10^{30}$, we scale x (currently holding the right hand side) by

$$10^{-35}10^{30}10^{-1}10^{-20} = 10^{-26}.$$

Then $x_1 = 10^{10}$ and the SAXPY gives a right hand side of -10^{30}. We scale again by $10^{-35}10^{30}10^{-30} = 10^{-35}$ to solve for x_2, and get $x_2 = -10^{30}$ with $s = 10^{-26}10^{-35} = 0$. The result,

$$\begin{bmatrix} 10^{-35} & \\ 10^{20} & 10^{-35} \end{bmatrix} \begin{bmatrix} 10^{-25} \\ -10^{30} \end{bmatrix} = 0 \begin{bmatrix} 10 \\ 10 \end{bmatrix},$$

is correct for x, only the scaling factor has underflowed. The exact solution, including the scaling factor s, would be

$$\begin{bmatrix} 10^{-35} & \\ 10^{20} & 10^{-35} \end{bmatrix} \begin{bmatrix} 10^{-25} \\ 10^{-25} - 10^{30} \end{bmatrix} = 10^{-61} \begin{bmatrix} 10 \\ 10 \end{bmatrix},$$

If $T_{i,i} = 0$, the matrix T is singular, and in applications this information is more important than an inexact solution x. We therefore change the problem, setting $s = 0$ and computing a nontrivial solution to $Tx = 0$. In the basic algorithm, we set $x_i = 1$ and all other x_j's to zero and continue. For example, in the system

$$\begin{bmatrix} 0.0 & & \\ 1.0 & 2.0 & \\ 3.0 & 4.0 & 5.0 \end{bmatrix} x = \begin{bmatrix} 0.0 \\ 3.0 \\ 12.0 \end{bmatrix},$$

a zero diagonal element is immediately encountered at $T_{1,1}$. The fact that an exact (but not unique) solution could still be found because $b_1 = 0$ is of little importance.

We set $s = 0$, $x_1 = 1$, and $x_2 = x_3 = 0$, and continue without further trouble. The solution that we compute to $Tx = sb$ is therefore $s = 0$, $x = [1.0, -0.5, -0.2]^T$.

The bounds on x and T are also used to determine when a step in the columnwise method can be performed without fear of overflow. Let $x_{max} = G(0) = \|x\|_\infty$ denote the maximum element currently in x. Adding a multiple of column j of L to x will not overflow if $x_{max} + |x_j|c_j \le \Omega$. If $|x_j| \le 1$ and $|x_j|c_j > \Omega - x_{max}$, then we scale x by $1/2$. If $|x_j| > 1$ and $c_j > (\Omega - x_{max})/|x_j|$, then we scale x by $1/(2|x_j|)$. This choice of scaling factor guarantees that each scaled term is bounded in absolute value by $\Omega/2$, and hence the sum is bounded by Ω. For example, suppose $\Omega = 10^{30}$ and the triangular system is

$$\begin{bmatrix} 1.0 & \\ 10^{30} & 1.0 \end{bmatrix} x = \begin{bmatrix} 10 \\ 10^{30} \end{bmatrix}.$$

After obtaining $x_1 = 10$, we have $c_1 = 10^{30}$ and $x_{max} = 10^{30}$. Since $c_1 > (\Omega - x_{max})/|x_1|$, we scale x by $1/(2|x_1|) = 1/20$, which makes $x_1 c_1 = 0.5 \times 10^{30}$ and $x_{max} = 0.5 \times 10^{29}$. Now the sum is bounded by $0.55 \times 10^{30} < \Omega$, as desired. The solution returned by SLATRS is $s = 0.05$, $x = [0.05, -0.45 \times 10^{30}]^T$.

3.2 The transposed problem

Similarly, a row-wise scheme is used to solve $T^T x = b$. The basic algorithm for T upper triangular ($T = U$) is

```
do $j = 1, \ldots, n$
    $x_j = \left(b_j - U^T_{1:j-1,j} x_{1:j-1}\right) / U_{j,j}$        (3.2)
end do
```

Floating point errors may occur when computing the dotproduct, when subtracting it from b_j, or when dividing by $U_{j,j}$. The algorithm for T lower triangular is similar, with the loop indices in reverse order.

For the transposed problem, we set $M(0) = \max(b_i, i = 1, \ldots, n)$ and compute the single bound $M(j)$, which is a bound on the magnitude of x_1, x_2, …, x_j:

$$\begin{aligned} M(j) &\le M(j-1)(1 + c_j)/|U_{j,j}| \\ &\le M(0) \textstyle\prod_{i=1}^{j} (1 + c_i)/|U_{i,i}|. \end{aligned}$$

$M(i)$ is assumed to be greater than or equal to $M(0)$ for $i \ge 1$, and c_i is greater than or equal to the one-norm of column i of U, not counting the diagonal. The column norms c_i are computed and scaled as in the non-transposed problem. We can safely call STRSV to solve the system if $1/M(n) > \Omega^{-1}$. Otherwise, a Level 1 BLAS version of the above algorithm is used.

Within the Level 1 BLAS solve, we must use in-line code instead of the Level 1 BLAS routines SDOT for the dot product (3.2) if $s_t < 1$, so that we may scale each element of T as it is used. Furthermore, we may need to scale x to prevent overflow in the intermediate results. If possible, we would prefer to scale by $T_{j,j}$, since we must divide by it anyway to form x_j.

If $|b_j| + c_j M(j-1) > \Omega$, then the intermediate results overflow. One solution is to scale x by $1/(2\max(1, M(j-1)))$, which guarantees that each term in the sum is at most $\Omega/2$, and hence that the sum is less than Ω. But this scaling may be more than is necessary if $|T_{j,j}| > 1$, since the sum is divided by $|T_{j,j}|$ before it is stored in x_j. Therefore, if $|T_{j,j}| > 1$, we scale x by $\min(1, |T_{j,j}|/(2M(j-1)))$, and then compute (for T upper triangular)

$$x_j = b_j/U_{j,j} + U^T_{1:j-1,j} x_{1:j-1}/U_{j,j},$$

using the algorithm

```
$s_u = 1/U_{j,j}$
$x_j = b_j/(s_t U_{j,j})$
do $i = 1, \ldots, j-1$
    $x_j = x_j - (s_u U_{i,j}) x_i$
end do
```

If $|T_{j,j}| \leq 1$, we simply scale x by $1/(2\max(1, M(j-1)))$ and determine the scaling necessary to divide by $T_{j,j}$ in a separate step. The three cases for $T_{j,j}$: $|T_{j,j}| > \Omega^{-1}$, $0 < |T_{j,j}| \leq \Omega^{-1}$, and $T_{j,j} = 0$, are the same as in the non-transposed problem.

For example, consider solving the 2 by 2 system $U^T x = b$ for x, where

$$U = \begin{bmatrix} 1.0 & 10^{20} \\ & 10^{20} \end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix} -2.0 \times 10^{18} \\ 2.0 \times 10^{38} \end{bmatrix}.$$

After obtaining $x_1 = -2.0 \times 10^{18}$, we see that

$$|b_1| + c_2 M(1) = 2.0 \times 10^{38} + 10^{20} \cdot 2.0 \times 10^{18} = 4.0 \times 10^{38}$$

which is greater than overflow (3.4×10^{38}) on the Sun. But since $|T_{2,2}|$ is large, we can avoid scaling by computing

$$x_2 = \frac{2.0 \times 10^{38}}{10^{20}} - \frac{10^{20}(-2.0 \times 10^{18})}{10^{20}} = 4.0 \times 10^{18}.$$

4 Implementation details

Three subroutines have been written for each Fortran floating-point data type. For REAL matrices, they are SLATRS for triangular matrices stored in a 2-D array, SLATPS for triangular matrices stored in packed format in a linear array, and SLATBS for triangular band matrices. The calling sequence of these three subroutines is as follows:

```
SLATRS( UPLO, TRANS, DIAG, NORMIN, N, A, LDA, X, SCALE, CNORM, INFO )
SLATPS( UPLO, TRANS, DIAG, NORMIN, N, AP, X, SCALE, CNORM, INFO )
SLATBS( UPLO, TRANS, DIAG, NORMIN, N, KD, AB, LDAB, X, SCALE, CNORM,
        INFO )
```

For a description of the arguments to SLATRS, see the leading comments to the subroutine in Appendix A. The array A is called AP in SLATPS to indicate packed storage, and AB in SLATBS to indicate band storage. For SLATBS, the argument KD is the number of subdiagonals or superdiagonals of the upper or lower triangular band matrix. The complex equivalents of these subroutines have the same calling sequence except that the matrix A and the vector X are COMPLEX instead of REAL.

4.1 The parameter Ω

In our software development, we assume that certain machine-dependent parameters, such as the overflow threshold and machine precision, are available. To compute these parameters, we have used a portable Fortran function subprogram SLAMCH, developed at NAG Ltd. for use in LAPACK. SLAMCH and its double-precision counterpart DLAMCH are real functions that accept a single character argument indicating the machine constant to be returned. The input values accepted are

'B': Base of the machine
'E': Epsilon [eps], the relative machine precision
'L': Largest exponent [emax] before overflow
'M': Minimum exponent [emin] before (gradual) underflow
'N': Number of base digits in the mantissa
'O': Overflow threshold = (base**emax)*(1-eps)
'P': Precision = eps*base
'R': 1.0 when rounding occurs in addition, 0.0 otherwise
'S': Safe minimum, such that its reciprocal does not overflow
'U': Underflow threshold = base**(emin-1)

The machine-dependent constant Ω is assigned to the Fortran variable BIGNUM and Ω^{-1} is called SMLNUM. In SLATRS, these constants are set as

```
      UNFL = SLAMCH( 'Safe minimum' )
      OVFL = SLAMCH( 'Overflow' )
      ULP = SLAMCH( 'Epsilon' )*SLAMCH( 'Base' )
      SMLNUM = MAX( UNFL/ULP, ONE/( ULP*OVFL ) )
      BIGNUM = ( ONE-ULP ) / SMLNUM
```

The scaled values for SMLNUM and BIGNUM allow for growth as large as $1/\mu$ when computing intermediate terms, where μ is the machine precision or "unit in the last place".

4.2 Modifications for the COMPLEX data type

The complex version of the triangular solve routines is similar to the real version, but we use the quantity

$$\text{CABS1}(z) = |Re(z)| + |Im(z)|$$

in place of $|z|$, since the absolute value is only used to compute scaling factors. The offdiagonal column norms c_j are computed using the Level 1 BLAS routine SCASUM, so CLATRS, CLATPS, and CLATBS are only guaranteed against overflow if the sum of the 1-norm of the real part of each column and the 1-norm of the imaginary part of each column excluding the diagonal can be computed without overflow. The maximum in b is computed as $\max(Re(b_i)/2 + Im(b_i)/2), i = 1 \ldots n)$ in order to avoid overflow if both the real and imaginary parts of some b_i are close to overflow.

We use the auxiliary routine CLADIV to compute the complex divide $x_i/T_{i,i}$ in real arithmetic, to avoid potential problems when a complex divide y/z is implemented as $(y\bar{z})/(z\bar{z})$. Both $A^{-T}x$ and $A^{-H}x$ are provided, in addition to $A^{-1}x$.

4.3 Operation count

The operation count for the Level 2 BLAS solve STRSV, assuming the matrix is order n, is $n(n+1)/2$ multiplies plus $n(n-1)/2$ adds, for a total of n^2 operations, less n divides if the matrix is unit triangular. Computing the offdiagonal column norms c_j in SLATRS requires $(n-2)(n-1)/2$ adds, but the c_j's can be saved and reused on a subsequent call, and with at least two iterations in the condition estimate we know there will be at least one more call to SLATRS with the same matrix. The additional fixed costs are $2n$ multiplies if the c_j's are scaled and n adds and $3n$ or $4n$ multiplies to compute the bounds to see if STRSV can be used. If the Level 1 BLAS solve is used, there are an additional n multiplies each time the vector x is scaled; in the worst case, where x is rescaled at every step, an extra n^2 multiplies would be needed, doubling the work of the standard triangular solve. If we assume two iterations of the Hager/Higham procedure and no extra scaling in SLATRS, then using the robust triangular solves instead of the BLAS increases the work by about 25%.

4.4 Test software

The subroutines SLATRS, SLATPS, and SLATBS are tested in the LAPACK linear equation test program as part of the test paths STR, STP, and STB, respectively [1]. Special pathological test matrices are generated to exercise all the scaling options to these subroutines, including

- Large right hand side, to test scaling of b

    1. Non-unit triangular with $O(1)$ matrix elements

2. Unit triangular

- Small first diagonal causes immediate overflow
  1. Offdiagonal column norms < 1
  2. Offdiagonal column norms > 1
- Small diagonals cause gradual overflow
- Small diagonals make growth factor underflow, but a small right hand side means that the solution does not overflow
- One zero diagonal element
- Large offdiagonals cause overflow when adding a column

5 Example of a robust condition estimator

We now show how the triangular matrix operations described in this report are used in the subroutine SGECON from LAPACK, which estimate the condition number of a real general matrix from its LU factorization. We assume that the LU factorization of A has been computed and work exclusively with the factors L and U.

SGECON is the high level routine which handles the reverse communication interface to SLACON, the routine to estimate the 1-norm of a matrix. SLACON returns a status value in the integer variable KASE, which expects the following actions when estimating the norm of A^{-1}:

KASE = 0: Done. The algorithm has converged or the maximum number of iterations has been reached. ANORM contains the estimate of the 1-norm.

KASE = 1: Compute $x \leftarrow A^{-1}x$ and call SLACON again with the other parameters unchanged.

KASE = 2: Compute $x \leftarrow A^{-T}x$ and call SLACON again with the other parameters unchanged.

The computations for KASE = 1 or 2 are performed by SLATRS. We call SLATRS twice for KASE = 1, to compute

$$y = L^{-1}s_l b$$

and

$$z = U^{-1}s_u y,$$

where y and z each overwrites x. Since

$$U^{-1}L^{-1}b = U^{-1}y/s_l = z/(s_l s_u),$$

we form $s = s_u s_l$, and if $s > 0$ and $\|x\|/s < \Omega$ then we compute $x \leftarrow x/s$ and call SLACON again, otherwise, we set AINVNM $= 0$ and quit. For further details, refer to the preliminary source code for SGECON in Appendix B.

The scaling step $x \leftarrow x/s$ is done by an LAPACK auxiliary routine SRSCL, instead of SSCAL from the Level 1 BLAS, because of the possibility that s could be less than Ω^{-1}. SRSCL (the RSCL stands for "Reciprocal Scaling") divides a vector x by a scalar s in a way that will not cause overflow, even on a machine for which the divide operation x/s is implemented as $(1/s)x$, unless the final result x/s overflows. First, SRSCL checks if $s \geq \Omega^{-1}$, and if it is, calls SSCAL to do the scaling with $1/s$. Otherwise, if $s < \Omega^{-1}$, both x and s are scaled repeatedly by Ω until s is larger than Ω^{-1}, and at that point SSCAL is called to multiply x by $1/s$.

The result returned for RCOND is usually $1/(\|A\|_1\|A^{-1}\|_1)$, but if AINVNM $= \|A^{-1}\|_1 = 0$, RCOND returns zero. RCOND also returns zero without computing AINVNM if $N = 0$ or if ANORM$= 0$ on entry to SGECON.

6 Conclusions

Higham's reverse communication procedure to implement Hager's method is a convenient tool, and it has been used in the LAPACK condition estimation routines to estimate the norm of the inverse of a matrix from its triangular factorization. The robustness of the Hager/Higham method depends on that of the high-level routine, which must form $A^{-1}x$ and $A^{-T}x$ for a given vector x using the factorization of A. Subroutines that solve a triangular system with scaling have been presented which are more suitable than the Level 2 BLAS for handling the badly conditioned triangular matrices that are likely to arise in condition estimation. Even if the solution of the triangular system overflows, these routines will not overflow as long as the 1-norm of each column of the triangular matrix excluding the diagonal can be computed without overflow. Additional work is required equal to about half of a triangular solve if no scaling is done, but the largest component of this work can be reused when the subroutine is called again, as it will be in Hager and Higham's iterative procedure.

Solving a scaled triangular system is a useful and general operation and is not restricted to condition estimation. The subroutine SLATRS is also used in the inverse iteration procedure in LAPACK, and may find application in other areas as well.

References

[1] E. Anderson, J. Dongarra, and S. Ostrouchov. Implementation guide for LAPACK. LAPACK Working Note 35, Technical Report CS-91-138, University of Tennessee, August 1991.

[2] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[3] W. W. Hager. Condition estimates. *SIAM J. Sci. Stat. Comput.*, 5:311–316, 1984.

[4] N. J. Higham. FORTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation. *ACM Trans. Math. Soft.*, 14(4):381–396, Dec. 1988.

Appendix A: SLATRS

```
      SUBROUTINE SLATRS( UPLO, TRANS, DIAG, NORMIN, N, A, LDA, X, SCALE,
     $                   CNORM, INFO )
*
*  -- LAPACK auxiliary routine (preliminary version) --
*     Univ. of Tennessee, Univ. of California Berkeley, NAG Ltd.,
*     Courant Institute, Argonne National Lab, and Rice University
*     August 15, 1991
*
*     .. Scalar Arguments ..
      CHARACTER          DIAG, NORMIN, TRANS, UPLO
      INTEGER            INFO, LDA, N
      REAL               SCALE
*     ..
*     .. Array Arguments ..
      REAL               A( LDA, * ), CNORM( * ), X( * )
*     ..
*
*  Purpose
*  =======
*
*  SLATRS solves one of the triangular systems
*
*     A *x = s*b  or  A'*x = s*b
*
*  with scaling to prevent overflow.  Here A is an upper or lower
*  triangular matrix, A' denotes the transpose of A, x and b are
*  n-element vectors, and s is a scaling factor, usually less than
*  or equal to 1, chosen so that the components of x will be less than
*  the overflow threshold.  If the unscaled problem will not cause
*  overflow, the Level 2 BLAS routine STRSV is called.  If the matrix A
*  is singular (A(j,j) = 0 for some j), then s is set to 0 and a
*  non-trivial solution to A*x = 0 is returned.
*
*  Arguments
*  =========
*
*  UPLO    (input) CHARACTER*1
*          Specifies whether the matrix A is upper or lower triangular.
*          = 'U':  Upper triangular
*          = 'L':  Lower triangular
*
*  TRANS   (input) CHARACTER*1
*          Specifies the operation applied to A.
*           = 'N':  Solve A * x = s*b  (No transpose)
*           = 'T':  Solve A'* x = s*b  (Transpose)
*           = 'C':  Solve A'* x = s*b  (Conjugate transpose = Transpose)
*
*  DIAG    (input) CHARACTER*1
*          Specifies whether or not the matrix A is unit triangular.
*          = 'N':  Non-unit triangular
*          = 'U':  Unit triangular
*
*  NORMIN  (input) CHARACTER*1
*          Specifies whether CNORM has been set or not.
*          = 'Y':  CNORM contains the column norms on entry
*          = 'N':  CNORM is not set on entry.  On exit, the norms will
*                  be computed and stored in CNORM.
*
*  N       (input) INTEGER
*          The order of the matrix A.  N >= 0.
*
*  A       (input) REAL array, dimension (LDA,N)
*          The triangular matrix A.  If UPLO = 'U', the leading n by n
*          upper triangular part of the array A contains the upper
*          triangular matrix, and the strictly lower triangular part of
*          A is not referenced.  If UPLO = 'L', the leading n by n lower
*          triangular part of the array A contains the lower triangular
*          matrix, and the strictly upper triangular part of A is not
*          referenced.  If DIAG = 'U', the diagonal elements of A are
*          also not referenced and are assumed to be 1.
*
*  LDA     (input) INTEGER
*          The leading dimension of the array A.  LDA >= max (1,N).
*
*  X       (input/output) REAL array, dimension (N)
*          On entry, the right hand side b of the triangular system.
*          On exit, X is overwritten by the solution vector x.
*
*  SCALE   (output) REAL
*          The scaling factor s for the triangular system
*             A * x = s*b  or  A'* x = s*b.
*          If SCALE = 0, the matrix A is singular or badly scaled, and
*          the vector x is an exact or approximate solution to A*x = 0.
*
*  CNORM   (input or output) REAL array, dimension (N)
*
*          If NORMIN = 'Y', CNORM is an input variable and CNORM(j)
*          contains the norm of the off-diagonal part of the j-th column
*          of A.  If TRANS = 'N', CNORM(j) must be greater than or equal
*          to the infinity-norm, and if TRANS = 'T' or 'C', CNORM(j)
*          must be greater than or equal to the 1-norm.
*
*          If NORMIN = 'N', CNORM is an output variable and CNORM(j)
*          returns the 1-norm of the offdiagonal part of the j-th column
*          of A.
*
*  INFO    (output) INTEGER
*          = 0:  successful exit
*          < 0:  if INFO = -k, the k-th argument had an illegal value
*
*  Further Details
*  ======= =======
*
*  A rough bound on x is computed; if that is less than overflow, STRSV
*  is called, otherwise, specific code is used which checks for possible
*  overflow or divide-by-zero at every operation.
*
*  A columnwise scheme is used for solving A*x = b.  The basic algorithm
*  if A is lower triangular is
*
*       x[1:n] := b[1:n]
*       for j = 1, ..., n
*            x(j) := x(j) / A(j,j)
*            x[j+1:n] := x[j+1:n] - x(j) * A[j+1:n,j]
*       end
*
*  Define bounds on the components of x after j iterations of the loop:
*     M(j) = bound on x[1:j]
*     G(j) = bound on x[j+1:n]
*  Initially, let M(0) = 0 and G(0) = max{x(i), i=1,...,n}.
*
*  Then for iteration j+1 we have
*     M(j+1) <= G(j) / | A(j+1,j+1) |
*     G(j+1) <= G(j) + M(j+1) * | A[j+2:n,j+1] |
*            <= G(j) ( 1 + CNORM(j+1) / | A(j+1,j+1) | )
*
*  where CNORM(j+1) is greater than or equal to the infinity-norm of
*  column j+1 of A, not counting the diagonal.  Hence
*
*     G(j) <= G(0) product ( 1 + CNORM(i) / | A(i,i) | )
*                  1<=i<=j
*  and
*
```

```
*     |x(j)| <= ( G(0) / |A(j,j)| ) product ( 1 + CNORM(i) / |A(i,i)| )
*                                   1<=i< j
*
*  Since |x(j)| <= M(j), we use the Level 2 BLAS routine STRSV if the
*  reciprocal of the largest M(j), j=1,..,n, is larger than
*  max(underflow, 1/overflow).
*
*  The bound on x(j) is also used to determine when a step in the
*  columnwise method can be performed without fear of overflow.  If
*  the computed bound is greater than a large constant, x is scaled to
*  prevent overflow, but if the bound overflows, x is set to 0, x(j) to
*  1, and scale to 0, and a non-trivial solution to A*x = 0 is found.
*
*  Similarly, a row-wise scheme is used to solve A'*x = b.  The basic
*  algorithm for A upper triangular is
*
*       for j = 1, ..., n
*            x(j) := ( b(j) - A[1:j-1,j]' * x[1:j-1] ) / A(j,j)
*       end
*
*  We simultaneously compute two bounds
*       G(j) = bound on ( b(i) - A[1:i-1,i]' * x[1:i-1] ), 1<=i<=j
*       M(j) = bound on x(i), 1<=i<=j
*
*  The initial values are G(0) = 0, M(0) = max{b(i), i=1,..,n}, and we
*  add the constraint G(j) >= G(j-1) and M(j) >= M(j-1) for j >= 1.
*  Then the bound on x(j) is
*
*       M(j) <= M(j-1) * ( 1 + CNORM(j) ) / | A(j,j) |
*
*            <= M(0) * product ( ( 1 + CNORM(i) ) / |A(i,i)| )
*                      1<=i<=j
*
*  and we can safely call STRSV if 1/M(n) and 1/G(n) are both greater
*  than max(underflow, 1/overflow).
*
*  ======================================================================
*
*     .. Parameters ..
      REAL               ZERO, HALF, ONE
      PARAMETER          ( ZERO = 0.0E+0, HALF = 0.5E+0, ONE = 1.0E+0 )
*     ..
*     .. Local Scalars ..
      LOGICAL            NOTRAN, NOUNIT, UPPER
      INTEGER            I, IMAX, J, JFIRST, JINC, JLAST
      REAL               BIGNUM, GROW, OVFL, REC, SMLNUM, SUMJ, TJJ,
     $                   TJJS, TMAX, TSCAL, ULP, UNFL, USCAL, XBND, XJ,
     $                   XMAX
*     ..
*     .. External Functions ..
      LOGICAL            LSAME
      INTEGER            ISAMAX
      REAL               SASUM, SDOT, SLAMCH
      EXTERNAL           LSAME, ISAMAX, SASUM, SDOT, SLAMCH
*     ..
*     .. External Subroutines ..
      EXTERNAL           SAXPY, SSCAL, STRSV, XERBLA
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          ABS, MAX, MIN
*     ..
*     .. Executable Statements ..
*
      INFO = 0
      UPPER = LSAME( UPLO, 'U' )
      NOTRAN = LSAME( TRANS, 'N' )
      NOUNIT = LSAME( DIAG, 'N' )
*
*     Test the input parameters.
*
      IF( .NOT.UPPER .AND. .NOT.LSAME( UPLO, 'L' ) ) THEN
         INFO = -1
      ELSE IF( .NOT.NOTRAN .AND. .NOT.LSAME( TRANS, 'T' ) .AND. .NOT.
     $         LSAME( TRANS, 'C' ) ) THEN
         INFO = -2
      ELSE IF( .NOT.NOUNIT .AND. .NOT.LSAME( DIAG, 'U' ) ) THEN
         INFO = -3
      ELSE IF( .NOT.LSAME( NORMIN, 'Y' ) .AND. .NOT.
     $         LSAME( NORMIN, 'N' ) ) THEN
         INFO = -4
      ELSE IF( N.LT.0 ) THEN
         INFO = -5
      ELSE IF( LDA.LT.MAX( 1, N ) ) THEN
         INFO = -7
      END IF
      IF( INFO.NE.0 ) THEN
         CALL XERBLA( 'SLATRS', -INFO )
         RETURN
      END IF
*
```

```
*     Quick return if possible
*
      IF( N.EQ.0 )
     $   RETURN
*
*     Determine machine dependent parameters to control overflow.
*
      UNFL = SLAMCH( 'Safe minimum' )
      OVFL = SLAMCH( 'Overflow' )
      ULP = SLAMCH( 'Epsilon' )*SLAMCH( 'Base' )
      SMLNUM = MAX( UNFL / ULP, ONE / ( ULP*OVFL ) )
      BIGNUM = ( ONE-ULP ) / SMLNUM
      SCALE = ONE
*
      IF( LSAME( NORMIN, 'N' ) ) THEN
*
*        Compute the 1-norm of each column, not including the diagonal.
*
         IF( UPPER ) THEN
*
*           A is upper triangular.
*
            DO 10 J = 1, N
               CNORM( J ) = SASUM( J-1, A( 1, J ), 1 )
   10       CONTINUE
         ELSE
*
*           A is lower triangular.
*
            DO 20 J = 1, N-1
               CNORM( J ) = SASUM( N-J, A( J+1, J ), 1 )
   20       CONTINUE
         END IF
      END IF
*
*     Scale the column norms by TSCAL if the maximum entry in CNORM is
*     greater than BIGNUM.
*
      IMAX = ISAMAX( N, CNORM, 1 )
      TMAX = CNORM( IMAX )
      IF( TMAX.LE.BIGNUM ) THEN
         TSCAL = ONE
      ELSE
         TSCAL = BIGNUM / TMAX
         CALL SSCAL( N, TSCAL, CNORM, 1 )
      END IF
*
*     Compute a bound on the computed solution vector to see if the
*     Level 2 BLAS routine STRSV can be used.
*
      J = ISAMAX( N, X, 1 )
      XMAX = ABS( X( J ) )
      XBND = XMAX
      IF( NOTRAN ) THEN
*
*        Compute the growth in A * x = b.
*
         IF( UPPER ) THEN
            JFIRST = N
            JLAST = 1
            JINC = -1
         ELSE
            JFIRST = 1
            JLAST = N
            JINC = 1
         END IF
         IF( NOUNIT ) THEN
*
*           A is non-unit triangular.
*
*           Compute GROW = 1/G(j) and XBND = 1/M(j).
*           Initially, G(0) = max{x(i), i=1,...,n}.
*
            GROW = ONE / MAX( XBND, SMLNUM )
            XBND = GROW
            DO 30 J = JFIRST, JLAST, JINC
*
*              Exit the loop if the growth factor is too small.
*
               IF( GROW.LE.SMLNUM )
     $            GO TO 50
*
*              M(j) = G(j-1) / abs(A(j,j))
*
               TJJ = ABS( A( J, J ) )*TSCAL
               XBND = MIN( XBND, MIN( ONE, TJJ )*GROW )
               IF( TJJ+CNORM( J ).GE.SMLNUM ) THEN
*
*                 G(j) = G(j-1)*( 1 + CNORM(j) / abs(A(j,j)) )
*
```

```
                  GROW = GROW*( TJJ / ( TJJ+CNORM( J ) ) )
               ELSE
*
*                 G(j) could overflow, set GROW to 0.
*
                  GROW = ZERO
               END IF
   30       CONTINUE
            GROW = XBND
         ELSE
*
*           A is unit triangular.
*
*           Compute GROW = 1/G(j), where G(0) = max{x(i), i=1,...,n}.
*
            GROW = MIN( ONE, ONE / MAX( XBND, SMLNUM ) )
            DO 40 J = JFIRST, JLAST, JINC
*
*              Exit the loop if the growth factor is too small.
*
               IF( GROW.LE.SMLNUM )
     $            GO TO 50
*
*              G(j) = G(j-1)*( 1 + CNORM(j) )
*
               GROW = GROW*( ONE / ( ONE+CNORM( J ) ) )
   40       CONTINUE
         END IF
   50    CONTINUE
*
      ELSE
*
*        Compute the growth in A' * x = b.
*
         IF( UPPER ) THEN
            JFIRST = 1
            JLAST = N
            JINC = 1
         ELSE
            JFIRST = N
            JLAST = 1
            JINC = -1
         END IF
         IF( NOUNIT ) THEN
*
```

```
*           A is non-unit triangular.
*
*           Compute GROW = 1/G(j) and XBND = 1/M(j).
*           Initially, M(0) = max{x(i), i=1,...,n}.
*
            GROW = ONE / MAX( XBND, SMLNUM )
            XBND = GROW
            DO 60 J = JFIRST, JLAST, JINC
*
*              Exit the loop if the growth factor is too small.
*
               IF( GROW.LE.SMLNUM )
     $            GO TO 80
*
*              G(j) = max( G(j-1), M(j-1)*( 1 + CNORM(j) ) )
*
               XJ = ONE + CNORM( J )
               GROW = MIN( GROW, XBND / XJ )
*
*              M(j) = M(j-1)*( 1 + CNORM(j) ) / abs(A(j,j))
*
               TJJ = ABS( A( J, J ) )*TSCAL
               IF( XJ.GT.TJJ )
     $            XBND = XBND*( TJJ / XJ )
   60       CONTINUE
            GROW = MIN( GROW, XBND )
         ELSE
*
*           A is unit triangular.
*
*           Compute GROW = 1/G(j), where G(0) = max{x(i), i=1,...,n}.
*
            GROW = MIN( ONE, ONE / MAX( XBND, SMLNUM ) )
            DO 70 J = JFIRST, JLAST, JINC
*
*              Exit the loop if the growth factor is too small.
*
               IF( GROW.LE.SMLNUM )
     $            GO TO 80
*
*              G(j) = ( 1 + CNORM(j) )*G(j-1)
*
               XJ = ONE + CNORM( J )
               GROW = GROW / XJ
   70       CONTINUE
```

```
         END IF
   80    CONTINUE
      END IF
*
      IF( ( GROW*TSCAL ).GT.SMLNUM ) THEN
*
*        Use the Level 2 BLAS solve if the reciprocal of the bound on
*        elements of X is not too small.
*
         CALL STRSV( UPLO, TRANS, DIAG, N, A, LDA, X, 1 )
      ELSE
*
*        Use a Level 1 BLAS solve, scaling intermediate results.
*
         IF( XMAX.GT.BIGNUM ) THEN
*
*           Scale X so that its components are less than or equal to
*           BIGNUM in absolute value.
*
            SCALE = BIGNUM / XMAX
            CALL SSCAL( N, SCALE, X, 1 )
            XMAX = BIGNUM
         END IF
*
         IF( NOTRAN ) THEN
*
*           Solve A * x = b
*
            DO 100 J = JFIRST, JLAST, JINC
*
*              Compute x(j) = b(j) / A(j,j), scaling x if necessary.
*
               XJ = ABS( X( J ) )
               IF( NOUNIT ) THEN
                  TJJS = A( J, J )*TSCAL
                  TJJ = ABS( TJJS )
                  IF( TJJ.GT.SMLNUM ) THEN
*
*                    abs(A(j,j)) > SMLNUM:
*
                     IF( TJJ.LT.ONE ) THEN
                        IF( XJ.GT.TJJ*BIGNUM ) THEN
*
*                          Scale x by 1/b(j).
*
```

```
                           REC = ONE / XJ
                           CALL SSCAL( N, REC, X, 1 )
                           SCALE = SCALE*REC
                           XMAX = XMAX*REC
                        END IF
                     END IF
                     X( J ) = X( J ) / TJJS
                     XJ = ABS( X( J ) )
                  ELSE IF( TJJ.GT.ZERO ) THEN
*
*                    0 < abs(A(j,j)) <= SMLNUM:
*
                     IF( XJ.GT.TJJ*BIGNUM ) THEN
*
*                       Scale x by (1/abs(x(j)))*abs(A(j,j))*BIGNUM
*                       to avoid overflow when dividing by A(j,j).
*
                        REC = ( TJJ*BIGNUM ) / XJ
                        IF( CNORM( J ).GT.ONE ) THEN
*
*                          Scale by 1/CNORM(j) to avoid overflow when
*                          multiplying x(j) times column j.
*
                           REC = REC / CNORM( J )
                        END IF
                        CALL SSCAL( N, REC, X, 1 )
                        SCALE = SCALE*REC
                        XMAX = XMAX*REC
                     END IF
*
*                    Scale both x(j) and A(j,j) until A(j,j) >= SMLNUM,
*                    then divide.
*
   85                CONTINUE
                     IF( ABS( TJJS ).LT.SMLNUM ) THEN
                        X( J ) = X( J )*BIGNUM
                        TJJS = TJJS*BIGNUM
                        GO TO 85
                     END IF
                     X( J ) = X( J ) / TJJS
                     XJ = ABS( X( J ) )
                  ELSE
*
*                    A(j,j) = 0:  Set x(1:n) = 0, x(j) = 1, and
*                    scale = 0, and compute a solution to A*x = 0.
```

```
*
                     DO 90 I = 1, N
                        X( I ) = ZERO
   90                CONTINUE
                     X( J ) = ONE
                     XJ = ONE
                     SCALE = ZERO
                     XMAX = ZERO
                  END IF
               END IF
*
*              Scale x if necessary to avoid overflow when adding a
*              multiple of column j of A.
*
               IF( XJ.GT.ONE ) THEN
                  REC = ONE / XJ
                  IF( CNORM( J ).GT.( BIGNUM-XMAX )*REC ) THEN
*
*                    Scale x by 1/(2*abs(x(j))).
*
                     REC = REC*HALF
                     CALL SSCAL( N, REC, X, 1 )
                     SCALE = SCALE*REC
                  END IF
               ELSE IF( XJ*CNORM( J ).GT.( BIGNUM-XMAX ) ) THEN
*
*                 Scale x by 1/2.
*
                  CALL SSCAL( N, HALF, X, 1 )
                  SCALE = SCALE*HALF
               END IF
*
               IF( UPPER ) THEN
                  IF( J.GT.1 ) THEN
*
*                    Compute the update
*                       x(1:j-1) := x(1:j-1) - x(j) * A(1:j-1,j)
*
                     CALL SAXPY( J-1, -X( J )*TSCAL, A( 1, J ), 1, X,
     $                           1 )
                     I = ISAMAX( J-1, X, 1 )
                     XMAX = ABS( X( I ) )
                  END IF
               ELSE
                  IF( J.LT.N ) THEN
```

```
*
*                    Compute the update
*                       x(j+1:n) := x(j+1:n) - x(j) * A(j+1:n,j)
*
                     CALL SAXPY( N-J, -X( J )*TSCAL, A( J+1, J ), 1,
     $                           X( J+1 ), 1 )
                     I = J + ISAMAX( N-J, X( J+1 ), 1 )
                     XMAX = ABS( X( I ) )
                  END IF
               END IF
  100       CONTINUE
*
         ELSE
*
*           Solve A’ * x = b
*
            DO 140 J = JFIRST, JLAST, JINC
*
*              Compute x(j) = b(j) - sum A(k,j)*x(k).
*                                    k<>j
*
               XJ = ABS( X( J ) )
               USCAL = TSCAL
               REC = ONE / MAX( XMAX, ONE )
               IF( CNORM( J ).GT.( BIGNUM-XJ )*REC ) THEN
*
*                 If x(j) could overflow, scale x by 1/(2*XMAX).
*
                  REC = REC*HALF
                  IF( NOUNIT ) THEN
                     TJJS = A( J, J )*TSCAL
                     TJJ = ABS( TJJS )
                     IF( TJJ.GT.ONE ) THEN
*
*                       Divide by A(j,j) when scaling x if A(j,j) > 1.
*
                        REC = MIN( ONE, REC*TJJ )
                        USCAL = USCAL / TJJS
                     END IF
                  END IF
                  IF( REC.LT.ONE ) THEN
                     CALL SSCAL( N, REC, X, 1 )
                     SCALE = SCALE*REC
                     XMAX = XMAX*REC
                  END IF
               END IF
*
               SUMJ = ZERO
               IF( USCAL.EQ.ONE ) THEN
*
*                 If the scaling needed for A in the dot product is 1,
*                 call SDOT to perform the dot product.
*
                  IF( UPPER ) THEN
                     SUMJ = SDOT( J-1, A( 1, J ), 1, X, 1 )
                  ELSE IF( J.LT.N ) THEN
                     SUMJ = SDOT( N-J, A( J+1, J ), 1, X( J+1 ), 1 )
                  END IF
               ELSE
*
*                 Otherwise, use in-line code for the dot product.
*
                  IF( UPPER ) THEN
                     DO 110 I = 1, J - 1
                        SUMJ = SUMJ + ( A( I, J )*USCAL )*X( I )
  110                CONTINUE
                  ELSE IF( J.LT.N ) THEN
                     DO 120 I = J + 1, N
                        SUMJ = SUMJ + ( A( I, J )*USCAL )*X( I )
  120                CONTINUE
                  END IF
               END IF
*
               IF( USCAL.EQ.TSCAL ) THEN
*
*                 Compute x(j) := ( x(j) - sumj ) / A(j,j) if 1/A(j,j)
*                 was not used to scale the dotproduct.
*
                  X( J ) = X( J ) - SUMJ
                  IF( NOUNIT ) THEN
*
*                    Compute x(j) = x(j) / A(j,j), scaling if necessary.
*
                     XJ = ABS( X( J ) )
                     TJJS = A( J, J )*TSCAL
                     TJJ = ABS( TJJS )
                     IF( TJJ.GT.SMLNUM ) THEN
*
*                       abs(A(j,j)) > SMLNUM:
*
```

```
                        IF( TJJ.LT.ONE ) THEN
                           IF( XJ.GT.TJJ*BIGNUM ) THEN
*
*                             Scale X by 1/abs(x(j)).
*
                              REC = ONE / XJ
                              CALL SSCAL( N, REC, X, 1 )
                              SCALE = SCALE*REC
                              XMAX = XMAX*REC
                           END IF
                        END IF
                        X( J ) = X( J ) / TJJS
                     ELSE IF( TJJ.GT.ZERO ) THEN
*
*                       0 < abs(A(j,j)) <= SMLNUM:
*
                        IF( XJ.GT.TJJ*BIGNUM ) THEN
*
*                          Scale x by (1/abs(x(j)))*abs(A(j,j))*BIGNUM.
*
                           REC = ( TJJ*BIGNUM ) / XJ
                           CALL SSCAL( N, REC, X, 1 )
                           SCALE = SCALE*REC
                           XMAX = XMAX*REC
                        END IF
*
*                       Scale both x(j) and A(j,j) until
*                       abs(A(j,j)) >= SMLNUM, then divide.
*
  125                   CONTINUE
                        IF( ABS( TJJS ).LT.SMLNUM ) THEN
                           X( J ) = X( J )*BIGNUM
                           TJJS = TJJS*BIGNUM
                           GO TO 125
                        END IF
                        X( J ) = X( J ) / TJJS
                     ELSE
*
*                       A(j,j) = 0:  Set x(1:n) = 0, x(j) = 1, and
*                       scale = 0, and compute a solution to A'*x = 0.
*
                        DO 130 I = 1, N
                           X( I ) = ZERO
  130                   CONTINUE
                        X( J ) = ONE
                        SCALE = ZERO
                        XMAX = ZERO
                     END IF
                  END IF
               ELSE
*
*                 Compute x(j) := x(j) / A(j,j)  - sumj if the dot
*                 product has already been divided by 1/A(j,j).
*
                  X( J ) = X( J ) / TJJS - SUMJ
               END IF
               XMAX = MAX( XMAX, ABS( X( J ) ) )
  140       CONTINUE
         END IF
         SCALE = SCALE / TSCAL
      END IF
*
*     Scale the column norms by 1/TSCAL for return.
*
      IF( TSCAL.NE.ONE ) THEN
         CALL SSCAL( N, ONE / TSCAL, CNORM, 1 )
      END IF
*
      RETURN
*
*     End of SLATRS
*
      END
```

Appendix B: SGECON

```
      SUBROUTINE SGECON( NORM, N, A, LDA, ANORM, RCOND, WORK, IWORK,
     $                   INFO )
*
*  -- LAPACK routine (preliminary version) --
*     Univ. of Tennessee, Univ. of California Berkeley, NAG Ltd.,
*     Courant Institute, Argonne National Lab, and Rice University
*     August 6, 1991
*
*     .. Scalar Arguments ..
      CHARACTER          NORM
      INTEGER            INFO, LDA, N
      REAL               ANORM, RCOND
*     ..
*     .. Array Arguments ..
      INTEGER            IWORK( * )
      REAL               A( LDA, * ), WORK( * )
*     ..
*
*  Purpose
*  =======
*
*  SGECON estimates the reciprocal of the condition number of a real
*  general matrix A, in either the 1-norm or the infinity-norm, using
*  the LU factorization computed by SGETRF.
*
*  An estimate is obtained for norm(inv(A)), and the reciprocal of the
*  condition number is computed as
*     RCOND = 1 / ( norm(A) * norm(inv(A)) ).
*
*  Arguments
*  =========
*
*  NORM    (input) CHARACTER*1
*          Specifies whether the 1-norm condition number or the
*          infinity-norm condition number is required:
*          = '1' or 'O':  1-norm
*          = 'I':         Infinity-norm
*
*  N       (input) INTEGER
*          The order of the matrix A.  N >= 0.
*
*  A       (input) REAL array, dimension (LDA,N)
*          The factors L and U from the factorization A = P*L*U
*          as computed by SGETRF.
*
*  LDA     (input) INTEGER
*          The leading dimension of the array A.  LDA >= max(1,N).
*
*  ANORM   (input) REAL
*          If NORM = '1' or 'O', the 1-norm of the original matrix A.
*          If NORM = 'I', the infinity-norm of the original matrix A.
*
*  RCOND   (output) REAL
*          The reciprocal of the condition number of the matrix A,
*          computed as RCOND = 1/(norm(A) * norm(inv(A))).
*
*  WORK    (workspace) REAL array, dimension (4*N)
*
*  IWORK   (workspace) INTEGER array, dimension (N)
*
*  INFO    (output) INTEGER
*          = 0:  successful exit
*          < 0: if INFO = -k, the k-th argument had an illegal value
*
*  =====================================================================
*
*     .. Parameters ..
      REAL               ONE, ZERO
      PARAMETER          ( ONE = 1.0E+0, ZERO = 0.0E+0 )
*     ..
*     .. Local Scalars ..
      LOGICAL            ONENRM
      CHARACTER          NORMIN
      INTEGER            IX, KASE, KASE1
      REAL               AINVNM, SCALE, SL, SMLNUM, SU
*     ..
*     .. External Functions ..
      LOGICAL            LSAME
      INTEGER            ISAMAX
      REAL               SLAMCH
      EXTERNAL           LSAME, ISAMAX, SLAMCH
*     ..
*     .. External Subroutines ..
      EXTERNAL           SLACON, SLATRS, SRSCL, XERBLA
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          ABS, MAX
*     ..
*     .. Executable Statements ..
*
*     Test the input parameters.
*
      INFO = 0
      ONENRM = NORM.EQ.'1' .OR. LSAME( NORM, 'O' )
      IF( .NOT.ONENRM .AND. .NOT.LSAME( NORM, 'I' ) ) THEN
         INFO = -1
      ELSE IF( N.LT.0 ) THEN
         INFO = -2
      ELSE IF( LDA.LT.MAX( 1, N ) ) THEN
         INFO = -4
      ELSE IF( ANORM.LT.ZERO ) THEN
         INFO = -5
      END IF
      IF( INFO.NE.0 ) THEN
         CALL XERBLA( 'SGECON', -INFO )
         RETURN
      END IF
*
*     Quick return if possible
*
      RCOND = ZERO
      IF( N.EQ.0 ) THEN
         RCOND = ONE
         RETURN
      ELSE IF( ANORM.EQ.ZERO ) THEN
         RETURN
      END IF
*
      SMLNUM = SLAMCH( 'Safe minimum' )
*
*     Estimate the norm of inv(A).
*
      AINVNM = ZERO
      NORMIN = 'N'
      IF( ONENRM ) THEN
         KASE1 = 1
      ELSE
         KASE1 = 2
      END IF
      KASE = 0
   10 CONTINUE
      CALL SLACON( N, WORK( N+1 ), WORK, IWORK, AINVNM, KASE )
      IF( KASE.NE.0 ) THEN
```

```
         IF( KASE.EQ.KASE1 ) THEN
*
*           Multiply by inv(L).
*
            CALL SLATRS( 'Lower', 'No transpose', 'Unit', NORMIN, N, A,
     $                   LDA, WORK, SL, WORK( 2*N+1 ), INFO )
*
*           Multiply by inv(U).
*
            CALL SLATRS( 'Upper', 'No transpose', 'Non-unit', NORMIN, N,
     $                   A, LDA, WORK, SU, WORK( 3*N+1 ), INFO )
         ELSE
*
*           Multiply by inv(U').
*
            CALL SLATRS( 'Upper', 'Transpose', 'Non-unit', NORMIN, N, A,
     $                   LDA, WORK, SU, WORK( 3*N+1 ), INFO )
*
*           Multiply by inv(L').
*
            CALL SLATRS( 'Lower', 'Transpose', 'Unit', NORMIN, N, A,
     $                   LDA, WORK, SL, WORK( 2*N+1 ), INFO )
         END IF
*
*        Divide X by 1/(SL*SU) if doing so will not cause overflow.
*
         SCALE = SL*SU
         NORMIN = 'Y'
         IF( SCALE.NE.ONE ) THEN
            IX = ISAMAX( N, WORK, 1 )
            IF( SCALE.LT.ABS( WORK( IX ) )*SMLNUM .OR. SCALE.EQ.ZERO )
     $         GO TO 20
            CALL SRSCL( N, SCALE, WORK, 1 )
         END IF
         GO TO 10
      END IF
*
*     Compute the estimate of the reciprocal condition number.
*
      IF( AINVNM.NE.ZERO )
     $   RCOND = ( ONE / AINVNM ) / ANORM
*
   20 CONTINUE
      RETURN
*
```

```
*     End of SGECON
*
      END
```