

Algorithm-Based Fault Tolerance for REE Numerical Applications: Basis and Implementations

Michael Turmon and Robert Granat
{turmon,granat}@aig.jpl.nasa.gov
Machine Learning Systems Group
M/S 126-347
Jet Propulsion Laboratory
Pasadena, CA 91109

5 May 1999

We describe and test the mathematical background for using checksum methods to validate results returned by a numerical subroutine operating in a fault-prone environment that causes unpredictable errors in data. We can treat subroutines that return results satisfying a necessary condition having a linear form — the checksum tests compliance with this necessary condition. These checksum schemes are called algorithm-based fault tolerance (ABFT). Here we discuss the theory and practice of setting numerical tolerances to separate errors caused by a fault from those inherent in finite-precision numerical calculations.

To separate these two classes of errors, we employ well-known bounds on error-propagation within linear algebraic algorithms. These bounds provide a maximum error that is to be expected due to register effects; any error in excess of this is taken to be the product of a fault. Adapting these bounds to the ABFT setting yields a series of tests having different efficiency and accuracy attributes.

Characteristics of a given scheme are concisely expressed using the standard receiver operating characteristic (ROC) curve. For a given error tolerance, a certain proportion of False Alarms (numerical errors tagged as data faults) and Detections (data faults correctly identified) will be observed. The ROC plots these two proportions parametrically as the tolerance is varied; this describes the performance achievable by a certain detection scheme and

provides a basis for choosing one scheme over others. Two series of tests are described here. The first shows the general effectiveness of the linear ABFT schemes we have proposed, and the second verifies the correct behavior of our parallel implementation of them.

We close this introduction by introducing some useful notation. Matrices and vectors are written in uppercase and lowercase roman letters; A^T is the transpose of the matrix A . Any identity matrix is always I ; context provides its dimension. A is *orthogonal* if $AA^T = I$. A square matrix is a *permutation* if it can be obtained by re-ordering the rows of I . The size of a vector v is measured by its p -norm, a non-negative real number $\|v\|_p$; similarly for matrices A . See [3] (hereafter abbreviated GVL), sections 2.2 and 2.3, for the definitions. The *submultiplicative property* of p -norms implies that $\|AB\|_p \leq \|A\|_p \|B\|_p$ and similarly for vectors.

1 General Considerations

In this paper we are concerned with these operations:

- Product: find the product $AB = P$, given A and B .
- LU decomposition: factor A as $A = PLU$ with P a permutation matrix, L unit lower-triangular, and U upper-triangular.
- Singular value decomposition: factor A as $A = UDV^T$, where D is diagonal and U and V are orthogonal matrices.
- System solution: solve for x in $Ax = b$ when given A and b
- Matrix inverse: given A , find B such that $AB = I$.

Although standard linear algebra packages provide many other routines, the ones above were identified by science application teams as the being of the most interest, partly on the basis of amount of time spent within them.

Each of these operations has been written to emphasize that some linear relation holds among the subroutine inputs and its computed outputs; we call this the *postcondition*. For the product, system solution, and inverse, this postcondition is necessary and sufficient, and completely characterizes the subroutine's task. For the other two, the postcondition is only a necessary condition and valid results must enjoy other properties as well. In either case, identifying and checking the postcondition provides a powerful sanity check on the proper functioning of the subroutine.

Before proceeding to examine these operations in detail, we mention two points involved in designing ABFT techniques. Suppose for definiteness that we plan to check one $m \times n$ matrix. Any reasonable checksum scheme must depend on the content of each matrix entry, otherwise some entries would not be checked. This implies that simply computing a checksum requires $O(mn)$ operations. Checksum ABFT schemes thus lose their attractiveness for operations taking $O(mn)$ or fewer operations (e.g. trace, sum, and 1-norm) because it is simpler and more directly informative to achieve fault-tolerance by repeating the computation. The second general point is that, although the postconditions above are linearly-checkable equalities, they need not be. For example, the largest eigenvalue of A is bounded by functions of the 1-norm and the ∞ -norm, both of which are easily computed but not linear. One could thus evaluate the sanity of a computation by checking postconditions that involve such inequalities. None of the operations we consider requires this level of generality.

The postconditions we consider generically involve comparing two linear maps, which are known in factorized form

$$L_1 L_2 \cdots L_p \stackrel{?}{=} R_1 R_2 \cdots R_q \quad . \quad (1)$$

This check can be done exhaustively via n linearly independent probes for an $n \times n$ system. Of course, exhaustive comparison would typically introduce about as much computation as would be required to recompute the answer from scratch. On the other hand, a typical fault to data fans out across the matrix outputs, and a single probe would be enough to catch most errors:

$$L_1 L_2 \cdots L_p w \stackrel{?}{=} R_1 R_2 \cdots R_q w \quad (2)$$

for some probe vector w . This is the approach originally recommended by Abraham and his colleagues [4] to implement ABFT in systolic arrays. It has since been extended and refined by several researchers [1, 2, 5, 7].

There are two designer-selectable choices controlling the numerical properties of such an ABFT system: the checksum weights w and the comparison method indicated above by $\stackrel{?}{=}$. When no assumptions may be made about the operands, the first is relatively straightforward: the elements of w should not vary greatly in magnitude so that results figure essentially equally in the check. (In particular, w should be everywhere nonzero.) In what follows, we let w be the vector of all ones; our implementation allows any w to be supplied by the user.

2 Error Propagation: Theory

After the checksum vector, the second choice is the comparison method. As stated above, we perform comparisons using the corresponding postcondition for each operation. To develop a test that is roughly independent of the matrices at hand, we use the well-known bounds on error propagation in linear operations. In what follows, we develop a test for each operation of interest. For each operation, we cite a theorem bounding the numerical error in the results, and then we use this bound to state a corollary defining a test which is roughly independent of the operands. Throughout, we use \mathbf{u} to represent the numerical precision of the underlying hardware; it is the difference between one and the next floating-point number larger than one.

It is important to understand that the error bounds given in the theorems are *qualitative* and determine the general characteristics of roundoff in our checksum implementations. The estimates we obtain in this section are bounds based on worst-case scenarios, and will typically predict roundoff error larger than practically observed. (See GVL, section 2.4.6, for more on this outlook.) In the ABFT context, using these bounds uncritically would mean setting thresholds too high and missing some fault-induced errors. Their value for us, and it is substantial, is to indicate how roundoff error scales with different inputs. This allows ABFT routines the opportunity to factor out the inputs, yielding performance that is more nearly input-independent. However, some problem-specific tuning will likely improve performance. Our goal is to simplify this tuning process as much as possible.

Theorem 1 *Let $\hat{P} = \text{mult}(A, B)$ be computed using a dot-product, outer-product, or gaxpy-based algorithm. The error matrix $E = \hat{P} - AB$ satisfies*

$$\|E\|_{\infty} \leq n\|A\|_{\infty}\|B\|_{\infty}\mathbf{u} \quad (3)$$

Proof. See GVL, section 2.4.8. □

Corollary 2 *An input-independent checksum test for `mult` is*

$$d = \hat{P}w - ABw \quad (4)$$

$$\|d\|_{\infty}/(\|A\|_{\infty}\|B\|_{\infty}\|w\|_{\infty}) \stackrel{\geq}{\leq} \tau\mathbf{u} \quad (5)$$

where τ is an input-independent threshold.

Remark. The test is expressed as a comparison (indicated by the $\stackrel{\geq}{\leq}$ relation) with a threshold; the latter is a scaled version of the floating-point accuracy. If the discrepancy is larger than $\tau\mathbf{u}$, a fault would be declared, otherwise the error is explainable by roundoff.

Proof. The difference $d = Ew$ so, by the submultiplicative property of norms and theorem 1,

$$\|d\|_\infty \leq \|E\|_\infty \|w\|_\infty \leq n \|A\|_\infty \|B\|_\infty \|w\|_\infty \mathbf{u}$$

and the dependence on A and B is removed by dividing by their norms. The factor of n is unimportant in this calculation, as noted in the remark beginning the section. \square

For the remaining operations, we require the notion of a *numerically realistic* matrix. The reliance of numerical analysts on certain proven algorithms is based on the rarity of certain pathological matrices that cause, for example, pivot elements in decomposition algorithms to grow exponentially. Even algorithms regarded as stable and reliable can be made to misbehave when given such unlikely inputs. Because the underlying routines will fail under such pathological conditions, we may neglect them in designing an ABFT system: such a computation is doomed even without faults. Accordingly, the theorems below must assume that the inputs are numerically realistic to obtain useable error bounds.

Theorem 3 *Let $(\hat{P}, \hat{L}, \hat{U}) = \text{lu}(A)$ be computed using a standard LU decomposition algorithm with partial pivoting. The backward error matrix E defined by $A + E = \hat{P}\hat{L}\hat{U}$ satisfies*

$$\|E\|_\infty \leq 8n^3 \rho \|A\|_\infty \mathbf{u} \tag{6}$$

where the growth factor ρ depends on the size of certain partial results of the calculation, and is bounded by a small constant for numerically realistic matrices.

Proof. See GVL, section 3.4.6. \square

Remark. This is close to the best possible bound for the discrepancy, because the error in simply writing down the matrix A must be of order $\|A\|_\infty \mathbf{u}$. The success of numerical linear algebra is in finding a way to factor realistic matrices A while incurring only a small penalty ρ beyond this lower bound.

Corollary 4 *An input-independent checksum test for lu as applied to numerically realistic matrices is*

$$d = \hat{P}\hat{L}\hat{U}w - Aw \tag{7}$$

$$\|d\|_\infty / (\|A\|_\infty \|w\|_\infty) \gtrsim \tau \mathbf{u} \tag{8}$$

where τ is an input-independent threshold.

Proof. We have $d = Ew$ so, by the submultiplicative property of norms and theorem 3,

$$\|d\|_\infty \leq \|E\|_\infty \|w\|_\infty \leq 8n^3 \rho \|A\|_\infty \|w\|_\infty \mathbf{u} \quad .$$

As before, the factor of $8n^3$ is unimportant in this calculation. For numerically realistic matrices, the growth factor ρ is bounded by a constant, and the indicated test is recovered by dividing by the norm of A . \square

Theorem 5 *Let $(\hat{U}, \hat{D}, \hat{V}) = \text{svd}(A)$ be computed using a standard singular value decomposition algorithm. The forward error matrix E defined by $A + E = \hat{U} \hat{D} \hat{V}^\top$ satisfies*

$$\|E\|_2 \leq \rho \|A\|_2 \mathbf{u} \quad (9)$$

where ρ is a constant not much larger than one for numerically realistic matrices A .

Proof. See GVL, section 5.5.8. \square

Corollary 6 *An input-independent checksum test for svd as applied to numerically realistic matrices is*

$$d = \hat{U} \hat{D} \hat{V}^\top w - Aw \quad (10)$$

$$\|d\|_2 / (\|A\|_2 \|w\|_2) \gtrsim \tau \mathbf{u} \quad (11)$$

where τ is an input-independent threshold.

Proof. We have $d = Ew$ so, by the submultiplicative property of norms,

$$\|d\|_2 \leq \|E\|_2 \|w\|_2 \leq \rho \|A\|_2 \|w\|_2 \mathbf{u}$$

and the dependence on A is removed by dividing by its norm. The constant ρ is negligible for numerically realistic matrices. \square

Corollary 7 *An input-independent checksum test for svd as applied to numerically realistic matrices is*

$$d = \hat{U} \hat{D} \hat{V}^\top w - Aw \quad (12)$$

$$\|d\|_\infty / (\|A\|_\infty \|w\|_\infty) \gtrsim \tau \mathbf{u} \quad (13)$$

where τ is an input-independent threshold.

Proof. By corollary 6, the check above with the 2-norm in place of the ∞ -norm is an input-independent checksum test. But since these two norms are equivalent in that

$$\begin{aligned} \|w\|_\infty &\leq \|w\|_2 \leq \sqrt{n}\|w\|_\infty \\ (1/\sqrt{n})\|A\|_\infty &\leq \|A\|_2 \leq \sqrt{m}\|A\|_\infty \end{aligned}$$

(see GVL sections 2.2.2 and 2.3.2), the two tests are also equivalent up to negligible constants. \square

Theorem 8 *Let $\hat{B} = \text{inv}(A)$ be computed using gaussian elimination with partial pivoting. The backward error matrix E defined by $(A + E)^{-1} = \hat{B}$ satisfies*

$$\|E\|_\infty \leq 8n^3 \rho \|A\|_\infty \mathbf{u} \quad (14)$$

with ρ as in theorem 3.

Proof. See GVL, section 3.4.6, which defines the backwards error for the linear system solution $Ax = b$. Since A^{-1} is calculated by solving the multiple right-hand-side problem $AA^{-1} = I$, the bound given there on $\|E\|_\infty$ applies here with the same growth factor ρ . (This growth factor depends only on the pivots in the LU factorization which underlies the inverse computation.) \square

Corollary 9 *An input-independent checksum test for inv as applied to numerically realistic matrices is*

$$d = w - A\hat{B}w \quad (15)$$

$$\|d\|_\infty / (\|A\|_\infty \|A^{-1}\|_\infty \|w\|_\infty) \gtrsim \tau \mathbf{u} \quad (16)$$

where τ is an input-independent threshold.

Proof. Note that $d = \Delta w$ where $\Delta = I - A(A + E)^{-1}$. Some algebra is necessary to extract the error E from Δ . Using the Sherman-Morrison formula (GVL section 2.1.3) to rewrite the inverse of $A + E$ we obtain

$$\begin{aligned} \Delta &= I - A[A^{-1} - A^{-1}(I + EA^{-1})^{-1}EA^{-1}] \\ &= (I + EA^{-1})^{-1}EA^{-1} \end{aligned} \quad (17)$$

For numerically realistic matrices, A dominates E and the first factor is negligible. Heuristically, this is because $E \ll A$ implies $EA^{-1} \ll AA^{-1} = I$,

Recommended Checksum Tests					
Algorithm	Δ	σ_1	σ_2	σ_3	Note
<code>mult</code>	$\hat{P} - AB$	$\ A\ \ B\ $	$\ \hat{P}\ $	$\ \hat{P}w\ $	—
<code>lu</code>	$\hat{P}\hat{L}\hat{U} - A$	$\ A\ $	$\ \hat{P}\hat{L}\hat{U}\ $	$\ Aw\ $	σ_0 easier than σ_1
<code>svd</code>	$\hat{U}\hat{D}\hat{V} - A$	$\ A\ $	$\ \hat{U}\hat{D}\hat{V}^\top\ $	$\ Aw\ $	σ_0 easier than σ_1
<code>inv</code>	$I - A\hat{B}$	$\ A\ \ A^{-1}\ $	$\ A\ \ \hat{B}\ $	$\ A\ \ \hat{B}w\ $	$\ A\hat{B}w\ $ useless

Table 1: Algorithms considered here, and recommended checksum tests.

collapsing that factor to I . More formally, inverting a numerically realistic matrix produces an error matrix E such that for any vector v , $\|Ev\| \ll \|Av\|$ otherwise the backward error E would be comparable to A . Since v is arbitrary and A is invertible, we may let $v = A^{-1}u$, obtaining that $\|EA^{-1}u\| \ll \|u\| = \|Iu\|$, showing that the operator EA^{-1} is dominated by I . Therefore we may neglect the first factor and the norm of the error is bounded by

$$\begin{aligned}
\|d\|_\infty &= \|\Delta w\|_\infty \\
&\leq \|E\|_\infty \|A^{-1}\|_\infty \|w\|_\infty \\
&\leq 8n^3 \rho \|A\|_\infty \|A^{-1}\|_\infty \|w\|_\infty \mathbf{u}
\end{aligned} \tag{18}$$

using the submultiplicative property of norms. As before, the factor of $8n^3$ is unimportant in this calculation. Invoking the assumption that A is a numerically realistic matrix allows us to neglect the growth factor ρ , yielding the indicated test. \square

Remarks.

1. The factor $\|A\|\|A^{-1}\|$ is the condition number $\kappa(A)$.
2. This bound on discrepancy, larger than that for `lu`, is the reason matrix inverse is numerically unstable.

3 Error Propagation: Implementation

It is straightforward to transform these results into algorithms for error detection via checksums. The principal issue is computing the desired matrix

norms efficiently from results needed in the root calculation. For example, in the matrix multiply, instead of computing $\|A\|\|B\|$, it is more efficient to compute $\|\hat{C}\|$ which equals $\|AB\|$ under fault-free conditions. By the submultiplicative property of norms, $\|AB\| \leq \|A\|\|B\|$, so this substitution always underestimates the upper bound on roundoff error, leading to false alarms. On the other hand, we must remember that the norm bounds are only general guides anyway. All that is needed is for $\|AB\|$ to scale as does $\|A\|\|B\|$; the unknown scale factor can be absorbed into τ .

Taking this one step farther, we might compute $\|\hat{C}w\|$ as a substitute for $\|A\|\|B\|\|w\|$. Here we run an even greater risk of underestimating the bound, especially if w is nearly orthogonal to the product, so it is wise to use instead $\lambda\|w\| + \|\hat{C}w\|$ for some problem-dependent λ . Extending this reasoning to the other operations yields the comparisons in table 1. The error criterion used there always proceeds from the number $\delta = \|\Delta w\|$ for the indicated difference matrix Δ ; this matrix is of course never explicitly computed. In addition to the obvious

$$T0 : \quad \delta/\|w\| \gtrless \tau \mathbf{u} \quad (\text{trivial test}) \quad (19)$$

we provide three other comparison tests

$$T1 : \quad \delta/(\sigma_1\|w\|) \gtrless \tau \mathbf{u} \quad (\text{ideal test}) \quad (20)$$

$$T2 : \quad \delta/(\sigma_2\|w\|) \gtrless \tau \mathbf{u} \quad (\text{approximate matrix test}) \quad (21)$$

$$T3 : \quad \delta/(\lambda\|w\| + \sigma_3) \gtrless \tau \mathbf{u} \quad (\text{approximate vector test}) \quad (22)$$

The *ideal test* is the one recommended by the theoretical error bounds, and is based on the supplied input arguments, but may not be computable. In contrast, both approximate tests are based on computed quantities, and may also be suggested by the reasoning above. The *matrix test* involves a matrix norm while the *vector test* involves a vector norm and is therefore more subject to false alarms. (Several variants of the matrix tests are available for these operations.) We note that the obvious vector test for `inv` uses $A\hat{B}w$, but since $\hat{B} = \text{inv}(A)$, this test becomes essentially equivalent to the trivial test. We therefore suggest using the vector/matrix test shown in the table.

Clearly the choice of which test to use is based on the interplay of computation time and fault-detection performance for a given population of input matrices. Because of the shortcomings of numerical analysis, we cannot predict definitively that one test will outperform another. The experimental

results reported in the next section are one indicator of real performance, and may motivate more detailed analysis of test behavior. Performance in application codes will be another criterion for choosing tests.

4 Results: Simulated Fault Conditions

In this section we show results of Matlab simulations of the proposed checksum tests. These simulations are intended to verify the essential effectiveness of the checksum technique for ABFT, as well as to sketch the relative behaviors of the tests described above. Due to the special nature of the population of test matrices, and the shortcomings of the fault insertion scheme, these results *must not* be taken as anything but an estimate of relative performance, and a rough estimate of ultimate absolute performance.

We briefly describe the simulation setup. In essence a population of random matrices is used as input to a given computation; faults are injected in half these computations, and a checksum test is used to attempt to identify the affected computations. Random test matrices A of a given condition number κ are generated by the rule

$$A = 10^\alpha U D_\kappa V^T \quad . \quad (23)$$

The random matrices U and V are the orthogonal factors in the QR factorization of two square matrices with normally distributed entries. The diagonal matrix D_κ is filled in by choosing random singular values, such that the largest singular value is unity and the smallest is $1/\kappa$. These matrices all have 2-norm equal to unity; the overall scale is set by α which is chosen uniformly at random between -8 and +8. A total of 800 64×64 matrices (forty applications of the rule (23) for each κ in $\{2^1, \dots, 2^{20}\}$) is processed.

Faults are injected in half of these runs (400 of 800) by first choosing a matrix to affect, and then flipping exactly one bit of its 64-bit representation. For example, if a call to `mult` is to suffer a simulated fault, first A or B is selected at random, and then one bit of A is toggled. If `lu` is to suffer a fault, one of A , L , or U is selected and the fault is injected. If A was selected, one can expect the computed \hat{L} and \hat{U} to have many incorrect elements; if L was selected, only one element of the LU decomposition would be in error. This scheme is intended to simulate errors occurring at various times within the computation.

Next, each of the four tests described above is used to identify faults; for a fixed τ this implies observing a certain false alarm rate and fault-detection

Average-case Matrices, All Faults

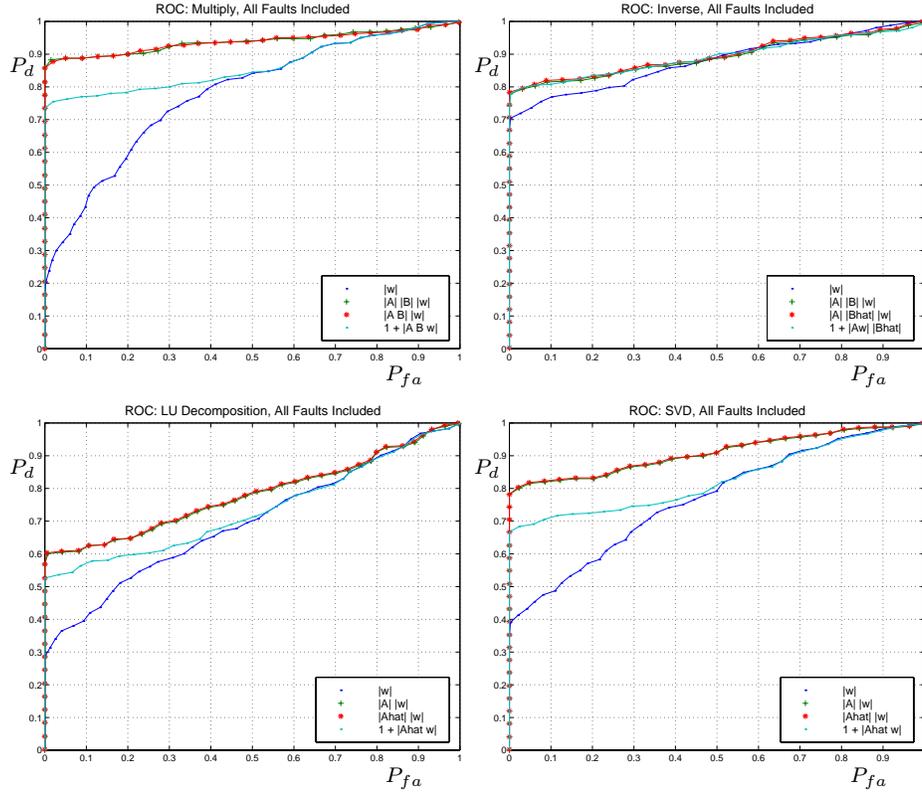


Figure 1: ROC for random matrices of bounded condition number, including all faults.

rate. The pair (P_{fa}, P_d) may be plotted parametrically versus τ to obtain an ROC curve which illustrates the performance achievable by a given test. See figure 1. In these figures, T_0 and T_3 are the solid blue lines with dots, with T_0 in dark blue lying below T_3 . T_2 is shown in red asterisks, and T_1 , the optimal test, in green crosses.

Of course, some missed fault detections are worse than others since many faults occur in the low-order bits of the mantissa and cause very minor changes in the matrix. Accordingly, a second set of ROCs is shown in figure 2. In this set, faults causing a minute perturbation (less than one part in 10^{-8} , about the accuracy of single-precision floating point) are screened from the results entirely. This curve is more realistic for our applications.

Average-case Matrices, Significant Faults

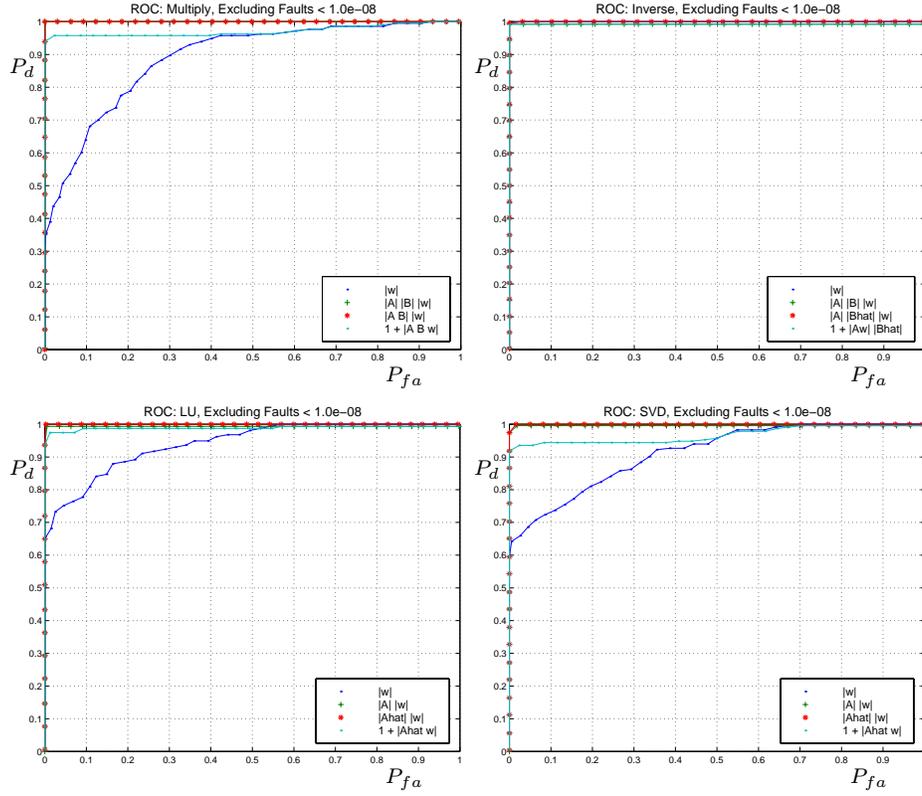


Figure 2: ROC for random matrices of bounded condition number, excluding faults of relative size less than 10^{-8} .

P^* Across Experiments

	Average-Case		Worst-Case	
	All	Sig.	All	Sig.
mult	0.86	1.00	0.63	0.92
inv	0.78	1.00	0.32	0.50
lu	0.60	1.00	0.43	0.90
svd	0.78	0.97	0.60	0.87
Mean	0.76	0.99	0.50	0.80

Table 2: Fault detection probability when no false alarms are permitted. Worst-case results are taken from the appendix. Results when all faults must be detected, and when only significant faults must be detected, are shown.

We may make some general observations about the results. Clearly $T0$, the un-normalized test, fares poorly in all experiments. This illustrates the value of the results on error propagation that form the basis for the normalized tests. Generally speaking,

$$T0 \ll T3 < T2 \approx T1 \quad . \quad (24)$$

This confirms theory, in which $T1$ is the ideal test and the others approximate it. In particular, $T1$ and $T2$ are quite similar because generally only an enormous fault can change the norm of a matrix — these cases are easy to detect.

Further, we note that the most relevant part of the ROC curve is when $P_{fa} \approx 0$; we may in fact be interested in the value P^* , defined to be P_d when $P_{fa} = 0$. P^* is the detection rate when no false alarms are permitted; it is summarized for these experiments in table 2. The first two columns of this table come from the data in figures 1 and 2; the other columns are from a “worst-case” matrix population which is described in the appendix. Under the average-case test conditions, about 99% of faults could be detected with no false alarms; this level of performance would seem adequate for REE purposes. In worst-case — and no science application should be in this regime — effectiveness drops to about 80%. This shows that fault detection will be more effective for numerically well-posed applications.

5 Results: Fault-free parallel operation

We briefly examine some characteristics of our parallel implementation of the checksum procedures described here. Shown in figure 3 are certain ROC curves for the four operations we have considered. In contrast to the results just reviewed, these curves were generated by checking Scalapack computations with Matlab.¹ In this test we use randomly perturbed matrices from the Matlab gallery selection (see equation (25) in the appendix). These matrices are generally ill-conditioned or poorly scaled, but serve as a demanding test set to check our routines against a known standard. In this case, for simplicity, the overall scale parameter $\alpha = 0$ and a fixed perturbation scale $\epsilon = 10^{-10}$ was used.

This time, identical matrix operands are given to Matlab and to our Scalapack implementation. Faults are not injected by modifying operands

¹Matlab uses LINPACK ZGEDI/ZGEFA for `inv` and `lu`, and ZSVDC for `svd`. For `mult`, Matlab uses a straightforward inner product implementation with nested loops. [6]

ROC: Testing Parallel Operation

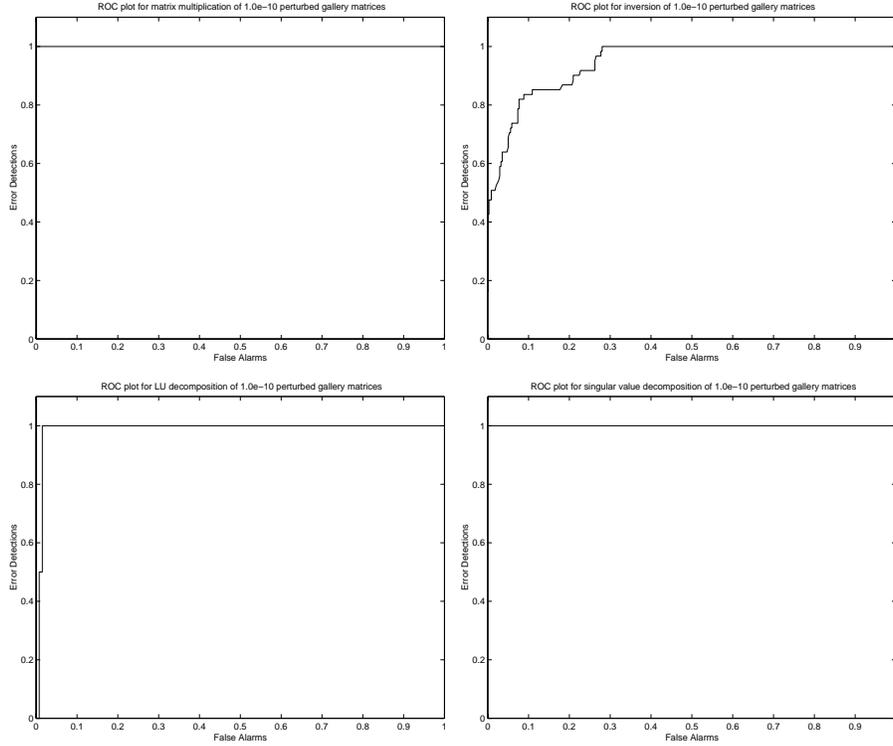


Figure 3: Parallel implementation checked by Matlab computation.

because our objective is to verify the correct numerical operation of our sub-routines. Each system computes the full result matrix; these are combined with the Scalapack checksum comparison to form an ROC as follows. We declare that an error has occurred when the two full results differ by more than a fixed tolerance (10^{-10} in these experiments). An error is declared to have been *detected* or not according to whether a checksum discrepancy was found by the Scalapack implementation. (The Matlab implementation does not compute a checksum; it is used only to find the full result matrix.) With these definitions, a false alarm, for example, means that Scalapack found a checksum discrepancy, but no significant discrepancy was present in the result of the computation. The ROC thus serves as a check, via Matlab, on the numerical characteristics of our Scalapack implementation. In doing these tests, the comparison rule $T0$ was used; this has smaller consequences

than in the previous section because most of the perturbed gallery matrices have roughly unit norm.

These curves were generated by sweeping the threshold τ used in the Scalapack *T0* test from 0 to ∞ . It is clear from the curves that there is excellent agreement between the Scalapack and Matlab versions of `mult`, `lu`, and `svd`. Indeed, when the matrix is badly scaled, ill-conditioned, or numerically unrealistic — causing Scalapack and Matlab to differ according to the full answer — Scalapack finds the error in the checksum calculation also. In essence, the message is: if the computation did not succeed, the checksum test discovers it. Because of the additional instability of the inverse algorithm, its results are less definite. One explanation is that the checksum test is missing some errors that occur in the computed inverse; this needs further investigation.

6 Conclusions

Theoretical results bounding the expected roundoff error in a given computation provide several types of input-independent threshold tests for checksum differences. The observed behavior of these tests is in good general agreement with theory, and readily computable tests are easy to define. All operations considered here (`mult`, `lu inv`, and `svd`) admit tests that are effective in detecting faults at the 99% level on typical matrix inputs. The base computations have been successfully implemented in parallel using Scalapack.

7 Appendix: Worst-case testing

Forty matrices from the Matlab “gallery” command (see table 3) are the basis for worst-case testing; these matrices are often ill-conditioned and badly scaled. Moreover, some members of this population deliberately violate the “numerically realistic” condition (see section 2) on which our tests are based. Given a matrix G from this set, a test matrix A is generated by the rule

$$A = 10^\alpha(G + \epsilon R) \tag{25}$$

where R is a square perturbation matrix of uniform random numbers between -1 and $+1$. The perturbation adjustment ϵ is stepped logarithmically from 10^{-10} to 10^{-4} , providing matrices progressively farther from the gallery set. The overall scale factor compensates for the fact that many of the gallery matrices are themselves scaled to have roughly unit norm; α is

Worst-case Matrices, All Faults

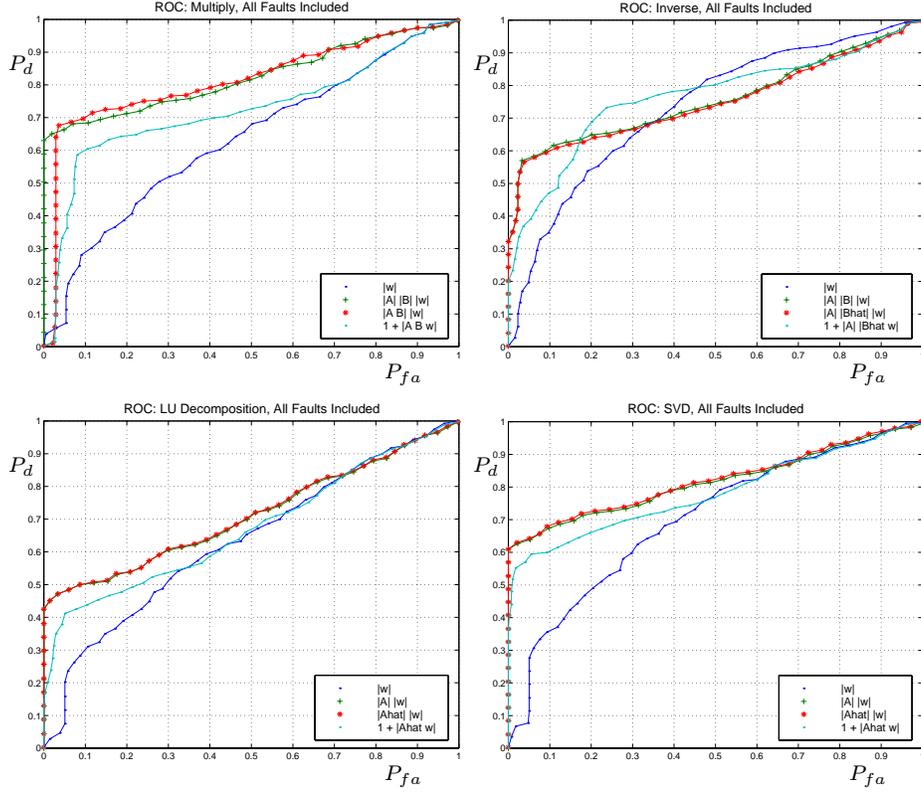


Figure 4: ROC for random worst-case matrices, including all faults

chosen uniformly at random between -8 and $+8$. A total of 800 64×64 matrices (20 perturbations of each of 40 gallery base matrices) is processed. Apart from this modified population, the experimental setup is just as described in section 4; the ROC curves in this appendix (figures 4 and 5) are directly comparable with the earlier ones.

We first note the anticipated loss in fault-detection performance, roughly of about 10-50%. The primary cause of this effect is probably the amplification of some floating-point roundoff errors into the status of fault-induced errors; another factor may be a breakdown of our error bounds, which are only approximate, for this difficult class of matrices. We note the superiority of the normalized tests, and that the ranking of tests observed in the average case, equation (24), is also seen here. We also see a decisive superiority of

Worst-case Matrices, Significant Faults

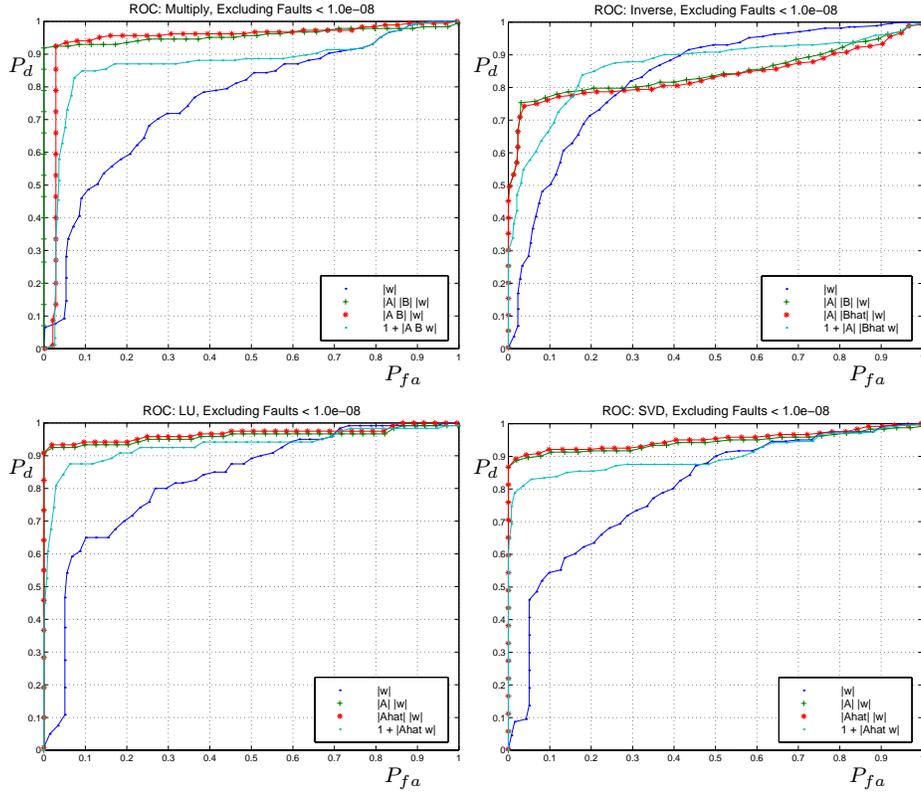


Figure 5: ROC for random worst-case matrices, excluding faults of relative size less than 10^{-8} .

T_1 over T_2 in the mult curves (upper-left panel of both figures).

Referring to the results for significant faults only, multiply, LU decomposition, and SVD have $P^* \approx 0.9$, which is quite favorable. The inverse, however, has $P^* \approx 0.5$, which is no doubt partly due to the numerical instability of inversion relative to the factorizations. Nonetheless, it would be of interest to locate the source of these missed faults, and perhaps to develop an improved test.

Name	$\log_{10} \kappa$	Description
cauchy	19.5	Cauchy matrix
chebpec	15.4	Chebyshev spectral differentiation matrix
chebvand	17.4	Vandermonde-like matrix for Chebyshev polynomials
chow	∞	Chow matrix (singular Toeplitz lower Hessenberg)
circul	1.8	Circulant matrix
clement	10.0	Clement matrix
condex	2.0	Counter-examples to condition number estimators
cycol	49.7	Matrix whose columns repeat cyclically
dramadah	16.1	Zero/one matrix whose inverse has large integers
fiedler	3.5	Fiedler matrix
forsythe	7.8	Forsythe matrix (perturbed Jordan block)
frank	18.3	Frank matrix
gearmat	17.4	Gear matrix
grcar	0.5	Grcar matrix
hanowa	1.4	Matrix whose eigenvalues lie on a vertical line
invhess	2.9	Inverse of an upper Hessenberg matrix
invol	34.7	Involutory matrix
jordbloc	1.9	Jordan block
kahan	11.0	Kahan matrix
kms	1.0	Kac-Murdock-Szego Toeplitz matrix
krylov	∞	Krylov matrix
lehmer	3.6	Lehmer matrix
lesp	1.6	Tridiagonal matrix with real, sensitive eigenvalues
lotkin	20.4	Lotkin matrix
minij	3.8	Symmetric positive definite matrix $\min(i, j)$
moler	17.1	Moler matrix (symmetric positive definite)
neumann	16.0	Singular matrix from the discrete Neumann problem
orthog	0.0	Orthogonal and nearly orthogonal matrices
parter	0.5	Parter matrix (Toeplitz with singular values $\approx \pi$)
pei	1.8	Pei matrix
prolate	16.8	Prolate matrix (symmetric, ill-conditioned Toeplitz)
rando	2.6	Random matrix with elements -1, 0 or 1
randhess	0.0	Random, orthogonal upper Hessenberg matrix
randsvd	7.8	Random matrix with pre-assigned singular values
redheff	2.8	Redheffer matrix
riemann	2.4	Matrix associated with the Riemann hypothesis
ris	0.5	Symmetric Hankel matrix
toeppe	4.2	Symmetric positive definite Toeplitz matrix
tridiag	3.2	Tridiagonal matrix (sparse)
triv	18.6	Upper triangular matrix of Wilkinson and others

Table 3: Matrices from the Matlab gallery used in testing.

References

- [1] D. L. Boley, R. P. Brent, G. H. Golub, and F. T. Luk. Algorithmic fault tolerance using the Lanczos method. *SIAM J. Matrix Anal. Appl.*, 13(1):312–332, 1992.
- [2] M. P. Connolly and P. Fitzpatrick. Fault-tolerant QRD recursive least squares. *IEE Proc. Comput. Digit. Tech.*, 143(2):137–144, 1996. IEE, not IEEE.
- [3] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins Univ., Baltimore, second edition, 1989.
- [4] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computing*, 33(6):518–528, 1984.
- [5] F. T. Luk and H. Park. An analysis of algorithm-based fault tolerance techniques. *Journal of Parallel and Distributed Computing*, 5:172–184, 1988.
- [6] MathWorks, 14 April 1999. Personal communication via e-mail from J. Regensburger.
- [7] Wang and Jah. Algorithm-based fault tolerance for FFT networks. *IEEE Transactions on Computing*, 43(7):849–854, 1994.