REPORT NO: NAWCADPAX--96-194-TR #103

GAUSS ELIMINATION: WORKHORSE OF LINEAR ALGEBRA

Peter R. Turner, Ph.D.
Mathematics Department
U S Naval Academy
Annapolis, MD 21402

5 AUGUST 1995

FINAL REPORT
Period Covering June 1995 to August 1995

Prepared for
OFFICE OF NAVAL RESEARCH
800 N. Quincy Street
Arlington, VA 22217

Reviewed By: [illegible] **Date:** 7/9/96
Author/COTR

Reviewed By: [illegible] **Date:** 7/24/96
LEVEL III Manager

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** 5 August 1995	**3. REPORT TYPE AND DATES COVERED** June 1995 to August 1995

4. TITLE AND SUBTITLE Gauss Elimination: Workhorse of Linear Algebra	**5. FUNDING NUMBERS**
6. AUTHOR(S) Peter R. Turner, Ph.D.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Air Warfare Center Aircraft Division Warminster Code 455100R07 Warminster, PA 18974-0591	**8. PERFORMING ORGANIZATION REPORT NUMBER** NAWCADPAX--96-194-TR
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research 800 N. Quincy Street Arlington, VA 22217	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES
Mathematics Department
U S Naval Academy
Annapolis, MD 21402

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for Public Release; Distribution is Unlimited	**12b. DISTRIBUTION CODE**

13. ABSTRACT *(Maximum 200 words)*

This report brings together many different aspects of Gauss elimination. The basic Gauss elimination (GE) algorithm is a fundamental tool of linear algebra computation for solving systems, computing determinants and determining the rank of matrix. All of these are discussed in varying contexts. These include different arithmetic or algebraic setting such as integer arithmetic or polynomial rings as well as conventional real (floating-point) arithmetic. These have effects on both accuracy and complexity analyses of the algorithm. These, too, are covered here. The impact of modern parallel computer architecture on GE is also included. Finally, GE is considered within the contex of "noisy" matrices. The effect of the noise in matrix entries on the effective rank of the matrix is the central aspect considered here.

14. SUBJECT TERMS Gauss elimination, Polynomial rings, Floating-point	**15. NUMBER OF PAGES** 52
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

Table of Contents

NAWCADPAX--96-194-TR

Gauss Elimination: Workhorse of Linear Algebra

PETER R TURNER
MATHEMATICS DEPARTMENT, U S NAVAL ACADEMY, ANNAPOLIS, MD 21402

ABSTRACT. This report brings together many different aspects of Gauss elimination. The basic Gauss elimination (GE) algorithm is a fundamental tool of linear algebra computation for solving systems, computing determinants and determining the rank of a matrix. All of these are discussed in varying contexts. These include different arithmetic or algebraic settings such as integer arithmetic or polynomial rings as well as conventional real (floating-point) arithmetic. These have effects on both accuracy and complexity analyses of the algorithm. These, too, are covered here. The impact of modern parallel computer architecture on GE is also included. Finally, GE is considered within the context of "noisy" matrices. The effect of the noise in matrix entries on the effective rank of the matrix is the central aspect considered here.

1. INTRODUCTION

In some form Gauss elimination, or *GE* as we shall often abbreviate it here, is probably the most widely used single computational tool in scientific computing for a very wide range of underlying problems. These include solution of partial differential equations, linear programming and least squares approximation in its various guises such as signal processing and linear regression. The basic problems for which it is used are the solution of linear systems of equations (including obtaining the solution space for underdetermined systems), computation of matrix determinants, determination of matrix rank or detection of singularity.

This report is intended to provide a unified treatment of one of the most important tools of linear algebra and of scientific computing. Gauss elimination is usually to be found as a major topic in texts on linear algebra (where the emphasis is on its theoretical basis), numerical analysis (with an emphasis on its practical implementation, paying attention to questions of roundoff error and numerical stability), parallel and vector computing (as an important illustration of the potential power of parallel computers). See [1], [3], [7], [17], [18] for examples.

Another aspect which is usually ignored in those particular texts is the interplay between GE and the computer arithmetic being used. Roundoff error analysis covers traditional floating-point computation for real (or complex) systems. For integer systems, however, the problem is not roundoff error but the growth in the matrix elements as the elimination proceeds. Recently this aspect must be extended beyond computer *arithmetic* to include questions related to the computer *algebra* system. In addition to the texts mentioned above, GE is included as a symbolic routine in computer algebra systems such as Maple[1] and Mathematica[2]. In this context, GE is not necessarily restricted to matrices with numerical entries, but includes matrices with entries in other algebraic rings such as algebraic polynomials. See [2], [27] for example. (Also note Maple linear algebra library functions such as pffge which performs "fraction-free" GE over polynomial rings.)

[1] Maple is a registered trademark of Maple Waterloo Software, Inc.
[2] Mathematica is a registered trademark of Wolfram Research, Inc.

In this report we begin with a description of the basic GE algorithm and some variations of this. Section 2 also contains a brief review of the main applications and of the complexity and error analyses for these. Section 3 is concerned with integer GE. The requirements of integer arithmetic raises some different issues. Roundoff error is no longer a problem, of course. Since the integers are not closed under division, we describe first the division-free form of the algorithm and then the questions raised by this –most notably, the growth of the dynamic range that is needed and what if any pivoting strategy should be used in an integer-arithmetic setting. Some of these issues were previously discussed for the specific context of residue number systems in the series of reports and papers [10], [11], [21], [22], [23].

Sections 4 and 5 deal with GE for matrices with entries in other fields or rings. Specifically, Section 4 discusses the use of rational arithmetic while Section 5 is concerned with more general rings such as rings of polynomials. The question of solvability is addressed.

Section 6 is devoted to the impact of various forms of parallel computer architectures on the implementation and efficiency of GE. Vector and array architectures are considered. The use of such systems for performing very long integer arithmetic for standard integer GE is also considered here. In Section 7, we consider GE as a tool for a specific practical problem - namely the determination of the effective rank of a matrix whose entries are contaminated by noise. One of the key aspects here is setting the appropriate tolerance level. This question is discussed in some detail.

1.1. Problem statement and notation. Except where specifically stated we shall consider a square $n \times n$ matrix A although some of the problems and solutions are similarly valid for rectangular matrices. Elements of the matrix A will be denoted by a_{ij}. Its determinant is denoted $\det A$ and its rank by $\mathrm{r}(A)$. The three basic problems we are concerned with here are the solution of systems of equations, computing $\det A$ and determining $\mathrm{r}(A)$. For definitions of any of these, see a standard text such as [1].

In the case of systems of equations, we use the notation

$$A\mathbf{x} = \mathbf{b}$$

and the elements of the unknown and right-hand side vectors are denoted x_i, b_i.

The matrices $L = (l_{ij})$ and $U = (u_{ij})$ denote lower and upper triangular matrices respectively. Typically they will be lower and upper triangular factors of A so that $A = LU$. In cases where any pivoting has been used L, U are the factors of a permuted version of A. That is $LU = PA$ where P is a permutation matrix indicating the order in which the rows of A have been used. (In a practical implementation of GE, the *matrix* P would usually be stored in the form of a permutation or *pivot* vector. We use the notation P for either form.)

In the context of either error analysis or matrices contaminated with noise we shall use E for the error or noise matrix. In case we wish the noise level to be explicit, we use E_σ to denote a matrix whose elements are (independent, identically distributed) normally distributed random variables with mean 0 and standard deviation σ. That is each element of E_σ is drawn from $N(0,\sigma)$.

The various arithmetic and algebraic systems in which computation is taking place are:

$\mathbf{R}$	the real numbers
$\mathbf{Z}$	the integers
$\mathbf{Q}$	the rationals
$\mathbf{F}$	the floating-point, fl.p., number system
$\mathbf{R}[x]$	algebraic polynomials over the reals
$\mathcal{R}$	a general algebraic ring

Clearly, $\mathbf{F}$ covers a wide range of different wordlengths, precisions and implementations of floating-point arithmetic. For most of this work the details of the particular fl.p. system being used are not central since the general properties are similar. Specific fl.p. systems will be detailed as needed.

2. The basic algorithms

The underlying principle of GE is that multiples of the first equation (or row of the matrix) are subtracted from all subsequent equations (rows) to eliminate the first unknown (element) from each of these. The process is then repeated to eliminate entries below the diagonal of each column in turn. The system of equations is then solved by substitution in the resulting triangular system. Alternatively the determinant of the original matrix is given by the product of the diagonal entries in the resulting matrix. The rank is given by counting the number of nonzero elements on the diagonal of the final array. In the case of solving a system of equations, whatever operations are performed on the matrix must also be performed on the right-hand side.

Of course, these statements are oversimplifications of the true situation but they contain the essence of the approach.

Example 1. *We illustrate the basic idea of GE for a* 3×3 *matrix.*

The initial matrix

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

is modified by subtracting d/a times the first row from the second and, similarly, g/a times the first row from the third to yield

$$\begin{bmatrix} a & b & c \\ 0 & e - \frac{d}{a}b & f - \frac{d}{a}c \\ 0 & h - \frac{g}{a}b & i - \frac{g}{a}c \end{bmatrix} = \begin{bmatrix} a & b & c \\ 0 & e' & f' \\ 0 & h' & i' \end{bmatrix}$$

say. Next h'/e' times the second row of the modified matrix is subtracted from the third row to give

$$\begin{bmatrix} a & b & c \\ 0 & e' & f' \\ 0 & 0 & i' - \frac{h'}{e'}f' \end{bmatrix} = \begin{bmatrix} a & b & c \\ 0 & e' & f' \\ 0 & 0 & i'' \end{bmatrix}$$

from which we obtain, for example,

$$\det A = ae'i''$$

Clearly there are difficulties in the event that either of a, e' are zero. If we had been solving a system of equations then the same multipliers would have been used to adjust the right-hand side vector so that the original system

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} u \\ v \\ w \end{bmatrix}$$

would be reduced to the triangular system

$$\begin{bmatrix} a & b & c \\ 0 & e' & f' \\ 0 & 0 & i'' \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} u \\ v' \\ w'' \end{bmatrix}.$$

In the rest of this section, we present the general version of this simple GE procedure and discuss some of the difficulties that can arise.

2.1. The ijk-form.

The "ijk-form" of GE is just the general $n \times n$ version of the algorithm used in Example 1 above.

Algorithm 1 Basic GE ijk form

```
Input    $n \times n$ matrix $A$ (and right-hand side $\mathbf{b}$ if solving a system)
Compute
        for $i = 1$ to $n-1$
            for $j = i+1$ to $n$
                $m := a_{ji}/a_{ii}$
                $a_{ji} := 0$
                $b_j := b_j - mb_i$        (if solving a system)
                for $k = i+1$ to $n$
                    $a_{jk} := a_{jk} - ma_{ik}$
Output (modified) matrix $A$ (and $\mathbf{b}$ if solving a system)
```

Remark 1. *The first operation in the middle, j- loop is the computation of the appropriate multiplier so that the apparent repetition of the division d/a in Example 1 is avoided. Clearly the algorithm fails if $a_{ii} = 0$. This is the extreme case of the need for pivoting which is discussed below.*

Remark 2. *The second line of the j-loop sets the element being "zeroed" directly rather than performing the arithmetic which should yield that 0.*

Pivoting. Pivoting is the name given to altering the order in which the rows are used in the GE algorithm. The simplest cause for the need for pivoting is the occurrence of a 0 in the appropriate diagonal position so that Algorithm 1 breaks down. In the floating-point environment an almost equally difficult situation is created by a very small diagonal entry which may be just the result of roundoff error in a quantity which would be 0 if *exact* arithmetic were being used. Such a pivot element can result in large errors in the computed solution.

The usual pivoting strategy is *partial pivoting* in which the current i-th column is searched for its element of largest magnitude on or below the diagonal. The row in which this occurs is then interchanged with the i-th row before the elimination continues. This form of GE is described in Algorithm 2.

Algorithm 2 GE with Partial Pivoting *ijk* form

```
Input     $n \times n$ matrix $A$ (and right-hand side b if solving a system)
Compute
          for $i = 1$ to $n - 1$
              find $p \geq i$ such that
                  $|a_{pi}| = \max\{|a_{ji}| : i \leq j \leq n\}$
              interchange rows $i$ and $p$ (including b if solving a system)
              for $j = i + 1$ to $n$
                  $m := a_{ji}/a_{ii}$
                  $a_{ji} := 0$
                  $b_j := b_j - mb_i$      (if solving a system)
                  for $k = i + 1$ to $n$
                      $a_{jk} := a_{jk} - ma_{ik}$
Output (modified) matrix $A$ (and b if solving a system)
```

Remark 3. *In practice it is usually simpler to keep a record of the order in which the rows are used by storing a permutation vector (which is equivalent to storing the permutation matrix P). If the interchange is performed, it is of course only necessary to interchange the entries for columns i through n since the first $i-1$ elements of both rows are already zero.*

LU factorization. For many purposes, it is desirable to make better use of the work involved in GE. By storing the multipliers used, we obtain the LU factorization of the original matrix A. That is we find a lower triangular factor L and an upper triangular one U such that

$$A = LU$$

or, in the case of pivoting, L, U are factors of a permuted version of the matrix A:

$$PA = LU$$

Algorithm 3 Basic LU Factorization *ijk* form

```
Input     $n \times n$ matrix $A$
Compute
          for $i = 1$ to $n - 1$
              for $j = i + 1$ to $n$
                  $a_{ji} := a_{ji}/a_{ii}$
                  for $k = i + 1$ to $n$
                      $a_{jk} := a_{jk} - a_{ji}a_{ik}$
Output (modified) matrix $A$
```

Remark 4. *For this form of the algorithm, the factors L and U are stored in the same locations as the original matrix A. The lower factor L has unit diagonal entries and so these need not be stored explicitly.*

Remark 5. *This choice of a unit lower triangular factor is convenient but the two factors can be scaled in a variety of ways. This choice is known as Doolittle factorization. Modifying the algorithm to yield a unit upper triangular factor is the Crout reduction.*

Remark 6. *If A is symmetric the factors can be scaled to share the same diagonal entries - this is then called the Cholesky factorization.*

Reasons and advantages. The principal advantages of the LU factorization lie in the fact that if multiple systems are to be solved with the same coefficient matrix then the factorization can be done just once and each system solved using forward and back substitution loops. This is then *much* cheaper computationally than, for example, inverting the matrix. This is detailed in Section 2.4 below.

On a serial computer, inverting a matrix (of dimension greater than 2×2) is *almost never* the right approach to a problem.

Pivoting. The LU factorization has the same need for pivoting as does the basic GE algorithm. The determination of the appropriate pivot element is just the same as for GE. It is necessary to keep a record of the interchanges in order to make the corresponding changes to a right-hand side vector or to obtain the correct sign for the determinant.

For these reasons it is more convenient to simply store the permutation matrix (or vector) P for subsequent use in the solution process. This is the form described in the following algorithm.

Algorithm 4 LU Factorization with Partial Pivoting ijk form

```
Input    $n \times n$ matrix $A$
Initialize permutation vector
        for $i = 1$ to $n$
            $P[i] := i$
Compute
        for $i = 1$ to $n-1$
            find $p \geq i$ such that
                $|a_{P[p]i}| = \max\{|a_{P[j]i}| : i \leq j \leq n\}$
            interchange $P[i]$ and $P[p]$
            for $j = i+1$ to $n$
                $a_{P[j]i} := a_{P[j]i}/a_{P[i]i}$
                for $k = i+1$ to $n$
                    $a_{P[j]k} := a_{P[j]k} - a_{P[j]i}a_{P[i]k}$
Output (modified) matrix $A$
```

Remark 7. *The "lower" factor has elements $l_{ij} = a_{P[i]j}$ for $i > j$. The "upper" factor has $u_{ij} = a_{P[i]j}$ for $i \leq j$. The product of these two is PA where P is now regarded as the permutation matrix which has 0's everywhere except for 1's in the positions $i, P[i]$ or, equivalently, $p_{ij} = \delta_{P[i]j}$.*

Example 2. *Consider the 4×4 matrix* $\begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 2 & 3 & 4 \\ 3 & 3 & 3 & 4 \\ 4 & 4 & 4 & 4 \end{bmatrix}$.

The initial permutation vector is $P = (1, 2, 3, 4)$. Clearly the pivot element in the first column is $a_{41} = 4$. So P is now $(4, 2, 3, 1)$ and the matrix resulting from the first step of the elimination process is $\begin{bmatrix} 1/4 & 1 & 2 & 3 \\ 1/2 & 0 & 1 & 2 \\ 3/4 & 0 & 0 & 1 \\ 4 & 4 & 4 & 4 \end{bmatrix}$. None of the subsequent elimination steps alter this matrix - but they do change the permutation vector to (4, 1, 3, 2) and finally to (4, 1, 2, 3) indicating the order in which the rows were used.

Then, we have $L = \begin{bmatrix} 1 & & & \\ 1/4 & 1 & & \\ 1/2 & 0 & 1 & \\ 3/4 & 0 & 0 & 1 \end{bmatrix}$, $U = \begin{bmatrix} 4 & 4 & 4 & 4 \\ & 1 & 2 & 3 \\ & & 1 & 2 \\ & & & 1 \end{bmatrix}$ and then $LU = \begin{bmatrix} 4 & 4 & 4 & 4 \\ 1 & 2 & 3 & 4 \\ 2 & 2 & 3 & 4 \\ 3 & 3 & 3 & 4 \end{bmatrix}$ which is also $PA = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 2 & 3 & 4 \\ 3 & 3 & 3 & 4 \\ 4 & 4 & 4 & 4 \end{bmatrix}$

2.2. Algorithmic variations. There are several variations on the basic ijk-form of GE which have merits for different computing environments. We concentrate here on just two of these variations. For simplicity we take no account of pivoting here, and we restrict to Gauss elimination rather than the factorization of the matrix A. (The modifications for these are straightforward.) The ijk-form in Algorithm 1 is well-suited to a conventional serial computer architecture with matrices stored by rows. For other storage schemes or processor architectures, alternatives may be preferred. A much more detailed discussion of these aspects is included in [18]. (The reader should be aware of a difference in notation between this report and [18]: there the pivot row is denoted by k and the current elimination row by i so that our ijk-form is denoted the kij-form in [18].)

The ikj-form for column vector operations. The algorithm for the ikj-form is as follows.

Algorithm 5 Basic GE ikj form

```
Input    $n \times n$ matrix $A$ (and right-hand side $\mathbf{b}$ if solving a system)
Compute
        for $i = 1$ to $n-1$
            for $j = i+1$ to $n$
                $m_j := a_{ji}/a_{ii}$;  $a_{ji} := 0$
                $b_j := b_j - m_j b_i$        (if solving a system)
            for $k = i+1$ to $n$
                for $j = i+1$ to $n$
                    $a_{jk} := a_{jk} - m_j a_{ik}$
Output (modified) matrix $A$ (and $\mathbf{b}$ if solving a system)
```

Remark 8. *Now the innermost loop is performing a "column + scalar×column" operation where the scalar is the element a_{ik} of the pivot row and the column it multiplies is the column of multipliers m_j computed by the initial j-loop. The corresponding operation in the ijk-form is a "row + scalar×row" operation.*

Both of these are examples of (in the terminology of [7]) *saxpy* operations. The ikj-form is better for matrices that are stored by columns. The choice of optimal algorithm is therefore system dependent.

Scalar product forms. For processors which are designed for rapid execution of a multiply-accumulate (MAC) operation such as $s := s + a \times b$ it is desirable to organize an algorithm to utilize this. The basic linear algebra operation which is particularly intense in its use of MAC operations is the formation of a scalar product. Further variations of GE are therefore designed to use scalar product operations as much as possible.

To see how this fits into the general pattern, it is easiest to look at the innermost operation of the LU factorization Algorithm 3. If the order of the loops is altered so that the i-loop is the innermost one then the operation being performed would indeed be a scalar product. The details are more complicated and again the reader is referred to [18] (Appendix 1) for a full explanation.

Algorithm 6 Basic LU factorization jki (scalar product) form

```
Input    $n \times n$ matrix $A$
Compute
    for $j = 2$ to $n$
        for $k = 2$ to $j$
            $a_{j,k-1} := a_{j,k-1}/a_{k-1,k-1}$
            for $i = 1$ to $k - 1$
                $a_{jk} := a_{jk} - a_{ji}a_{ik}$
        for $k = j + 1$ to $n$
            for $i = 1$ to $j - 1$
                $a_{jk} := a_{jk} - a_{ji}a_{ik}$
Output (modified) matrix $A$
```

Remark 9. *As in the case of Algorithm 3, the output matrix contains the Doolittle factors of the original matrix.*

Remark 10. *Although the loop structure of Algorithm 6 is more complicated, the arithmetic operation count is unchanged. The k-loop is separated into two parts corresponding to the lower and upper triangles of the matrix. The innermost loop in each of the parts is using MAC operations to update matrix entries.*

2.3. Applications.

Solving linear systems. The most frequent application of GE and LU factorization is to the solution of a linear system

$$A\mathbf{x} = \mathbf{b} \tag{1}$$

Then the GE algorithms, Algorithms 1 and 2, result in an equivalent upper triangular system

$$U\mathbf{x} = \widehat{\mathbf{b}} \tag{2}$$

whose solution is the same as that of (1). This system is then solved using *back substitution.*

Algorithm 7 Back substitution — Row form

```
Input    $n \times n$ upper triangular matrix $U$, $n$-vector $\mathbf{b}$
Initialize solution   $\mathbf{x} := 0$
Compute
```

```
        for $i = n$ downto 1
            for $j = i + 1$ to $n$
                $b_i := b_i - u_{ij}x_j$
            $x_i := b_i/u_{ii}$
    Output solution $\mathbf{x}$
```

Remark 11. *Note that* $\mathbf{b}$ *has been used (in place of* $\widehat{\mathbf{b}}$*) to denote the right hand side of (2) in this algorithm.*

Remark 12. *This version of back substitution has been arranged in such a way that the outer loop results in obtaining one further component of the solution each time. The inner loop of this algorithm subtracts the scalar product of the i-th row of U and the current solution vector from the i-th component of the right -hand side. It is thus well-suited to a computational environment which is designed for efficient multiply-accumulate operations and the matrix is stored by rows.*

For architectures which are well-suited to "vector plus scalar$\times$vector" operations — these are the *saxpy*'s of [7] — a column-oriented version of this algorithm is probably to be preferred.

Algorithm 8 Back substitution — Column form

```
Input      $n \times n$ upper triangular matrix $U$, $n$-vector $\mathbf{b}$
Initialize solution   $\mathbf{x} := \mathbf{0}$
Compute
        for $j = n$ downto 1
            $x_j := b_j/u_{jj}$
            for $i = 1$ to $j - 1$
                $b_i := b_i - u_{ij}x_j$
Output solution $\mathbf{x}$
```

Remark 13. *Here the inner loop subtracts* $x_j\times$ *the j-th column of U from the entire right-hand side vector as soon as it is available.*

In the event that pivoting has been achieved by using a pivot vector (or matrix) such as in Algorithms 3 and 4 then that pivot information must be carried through to the back substitution phase. Essentially, this means replacing all *row* labels in these algorithms by $P[i]$ or $P[j]$. For the column-form of the algorithm this may result in some loss of efficiency since data will no longer be accessed in a natural order. For this reason it may be worth performing the row interchanges for architectures where this is the preferred algorithm. These last comments apply equally to the LU factorization solution of the system (1).

For the LU factorization, Algorithms 3 and 4 result in (a permuted version of) the system

$$LU\mathbf{x} = \mathbf{b} \tag{3}$$

For simplicity we ignore the effect of pivoting here. Solving (3) is equivalent to solving $L\mathbf{y} = \mathbf{b}$ and then $U\mathbf{x} = \mathbf{y}$. The first of these requires a *forward elimination* process which is (almost) entirely equivalent to the back substitution used for the second stage. The only differences are that the loops run in the opposite directions and, because L is a *unit* lower triangular matrix the division by the diagonal element is not needed.

Remark 14. *We can now see the advantage of LU factorization for solving multiple systems with the same coefficient matrix. To solve a second or subsequent system, would require only forward and back substitution phases to be repeated since the factorization would be identical. For the elementary form of GE, the elimination stage would also need to be repeated. As we see in Section 2.4, this is the most expensive part of the whole process.*

Determinant evaluation. Whether we use GE or LU, the evaluation of $\det A$ is extremely simple once the elimination phase is complete. (We are taking no account of any questions of error analysis or stability at this stage.)

In the absence of pivoting the determinant is simply the product of the diagonal elements of U. We describe the algorithm in terms of the LU factorization.

Algorithm 9 Determinant evaluation - without pivoting

```
Input    $n \times n$ matrix $A$
Compute  LU factorization by Algorithm 3 (or any of its equivalent variations)
    initialize  $D := 1$
        for $i = 1$ to $n$
            $D := D * a_{ii}$
Output   det $A$ is $D$.
```

The only additional difficulty if pivoting is used is keeping track of the interchanges that have been made (whether explicitly or implicitly) in order to get the correct sign for the determinant. It suffices to maintain a variable s, say in the pivoting algorithm which is initialized to $s := 1$ and is multiplied by -1 whenever $i \neq p$ (in Algorithm 2) or $P[i] \neq P[p]$ in Algorithm 4. In the latter case the multiplication loop in Algorithm 9 would also be replaced by

```
for $i = 1$ to $n$
    $D := D * a_{P[i]i}$
```

Rank and singularity detection. Again, neglecting any problems created by roundoff errors or ill-conditioning in the matrix, once either GE or LU (with pivoting) has been completed, singularity is detected simply by seeking a 0 entry in a pivot position. In fact it is sufficient to examine a_{nn} since, if any pivot element is zero, then necessarily $a_{nn} = 0$.

Again note we are assuming both pivoting and exact arithmetic or algebra in making this statement. In practice, for floating-point arithmetic at least, this is not sufficient and more care must be taken to test for *near-singularity*.

Extending our ideal world analysis, by counting the number of zero pivots we obtain the rank-deficiency of the matrix. Equivalently the number of nonzero pivots yields the rank of A. This is a much more optimistic claim than even the singularity statement. Within a computer algebra system or with exact arithmetic such statements are correct.

In our ideal world, the rank algorithm is simple:

Algorithm 10 Rank detection — for *exact* arithmetic or algebra systems

```
Input    $n \times n$ matrix $A$
Compute  LU factorization with pivoting by Algorithm 4 (or any of its equivalent variations)
```

```
        initialize   $r := 0$
            for $i = 1$ to $n$
                if $a_{P[i]i} \neq 0$ then $r := r + 1$
    Output       rank $A$ is $r$.
```

In the domain of real numerical computation, the question of rank determination is more difficult. The effect of roundoff error on GE has been thoroughly analysed. Some of that analysis is summarized in Section 2.5. Packages such as MATLAB[3] (see [16], for example) include rank as a function in their computational library and use a carefully computed tolerance dependent on parameters of the computer system to determine the "true" rank of the original matrix from its Singular Value Decomposition (SVD) [7]. In the event that the matrix entries are themselves subject to error — perhaps experimental error or the effect of noisy data, for instance — the problem becomes more difficult to handle reliably. This is discussed further in Section 2.5 and then in Section 7.

The solution space for underdetermined systems. In the case of underdetermined systems of equations, whether this is the result of having fewer equations than unknowns or of rank deficiency in the matrix, a set of vectors spanning the solution space is easy to obtain from the LU factorization of the matrix.

The backward substitution must be performed the same number of times as the rank deficiency. Thus if the $n \times n$ matrix has rank r, we solve the smaller system $n - r$ times each time with r of the unknowns specified. To guarantee linearly independent solutions, it suffices to use the standard basis vectors $\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_{n-r}$ in turn to specify the values of $x_{r+1}, x_{r+2}, \ldots, x_n$. That is, we first set $(x_{r+1}, x_{r+2}, \ldots, x_n) = (1, 0, \ldots, 0)$ and solve for the remaining unknowns. The process is repeated for $(x_{r+1}, x_{r+2}, \ldots, x_n) = (0, 1, 0, \ldots, 0)$ and so on until we have used $(x_{r+1}, x_{r+2}, \ldots, x_n) = (0, \ldots, 0, 1)$.

The algorithms are simple extensions of those outlined earlier. Similar basis vectors could be used in more abstract settings with symbolic computer algebra systems.

2.4. Elementary complexity analysis.

The computational complexity of floating-point algorithms is often measured by the number of floating-point arithmetic operations that are required. Traditionally, the relative efficiency of algorithms was measured by counting multiplications and divisions and neglecting addition and subtraction operations. For modern processors the time required for floating-point multiplication is not much greater than that for addition and so it is sensible to consider all arithmetic operations. Division still costs substantially more and any elementary function evaluations are typically yet more expensive. The floating-point operation counts for GE are well-known. They can be found in almost any standard text on numerical analysis or computational linear algebra such as [3] or [7].

Operation counts.

TABLE 1 Floating-point operation counts for GE solution of an $n \times n$ linear system $A\mathbf{x} = \mathbf{b}$.

Operation	Forward elimination	Back substitution	TOTAL
+ or −	$\frac{1}{3}n\left(n^2-1\right)$	$\frac{1}{2}n\,(n-1)$	$\frac{1}{6}n\,(n-1)\,(2n+5)$
×	$\frac{1}{3}n\left(n^2-1\right)$	$\frac{1}{2}n\,(n-1)$	$\frac{1}{6}n\,(n-1)\,(2n+5)$
/	$\frac{1}{2}n\,(n-1)$	n	$\frac{1}{2}n\,(n+1)$

[3]MATLAB is a registered trademark of The MathWorks, Inc

If multiple systems with the same coefficient matrix are to be solved then the overall operation count for GE is obtained by multiplying these totals by the number of systems. In particular the inversion of the matrix A would therefore result in multiplying all these totals by n so that matrix inversion by GE is an $O(n^4)$ operation.

The corresponding table of operation counts, Table 2, for Doolittle factorization makes it easy to see the practical advantage of using the LU factorization whenever multiple systems are to be solved.

TABLE 2 Floating-point operation counts for solution of $A\mathbf{x} = \mathbf{b}$ using Doolittle LU factorization.

Operation	Factorization	Forward sub	Back sub	TOTAL
+ or −	$\frac{1}{6}n(n-1)(2n-1)$	$\frac{1}{2}n(n-1)$	$\frac{1}{2}n(n-1)$	$\frac{1}{6}n(n-1)(2n+5)$
×	$\frac{1}{6}n(n-1)(2n-1)$	$\frac{1}{2}n(n-1)$	$\frac{1}{2}n(n-1)$	$\frac{1}{6}n(n-1)(2n+5)$
/	$\frac{1}{2}n(n-1)$	0	n	$\frac{1}{2}n(n+1)$

Now in order to solve several systems, the factors of the original matrix are already known and so only the operation counts for the forward and backward substitution phases need to be repeated.

TABLE 3 Floating-point operation counts for solution of m systems using LU factorization and GE

Op	Factorization	1 system	TOTAL for LU	TOTAL for GE
+ or −	$\frac{1}{6}n(n-1)(2n-1)$	$n(n-1)$	$\frac{1}{6}n(n-1)(2n-1)$ $+mn(n-1)$	$\frac{m}{6}n(n-1)(2n+5)$
×	$\frac{1}{6}n(n-1)(2n-1)$	$n(n-1)$	$\frac{1}{6}n(n-1)(2n-1)$ $+mn(n-1)$	$\frac{m}{6}n(n-1)(2n+5)$
/	$\frac{1}{2}n(n-1)$	n	$\frac{1}{2}n(n-1)+mn$	$\frac{m}{2}n(n+1)$

It is immediately apparent that the savings here are potentially great for large systems with multiple right-hand sides. For inversion of a large matrix, GE requires approximately $2n^4/3$ flops compared to $8n^3/3$ for LU factorization.

2.5. Elementary error analysis.

In this section, we summarize briefly the well-known error-analysis of GE and LU factorization. For a more complete description see [3] or [7]. For all the usual error bounds, we need the notation associated with the standard vector and matrix norms and the condition number of a matrix.

Condition numbers and norms. There are three important vector norms that are commonly used in anlaysing numerical methods for matrix problems. These are the L_1, L_2 and L_∞ norms which are defined for $\mathbf{x} \in \mathbf{R}^n$ by

$$||\mathbf{x}||_1 = \sum_{i=1}^{n} |x_i| \tag{4}$$

$$||\mathbf{x}||_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2} \tag{5}$$

$$||\mathbf{x}||_\infty = \max_{1\le i\le n} |x_i| \tag{6}$$

which are often known as the *taxicab*, *Euclidean* and *maximum* norms respectively.

Any vector norm has an associated matrix norm given by

$$||A|| = \max\{||A\mathbf{x}||/||\mathbf{x}||\}$$

In particular the L_1 and L_∞ norms can be obtained by

$$||A||_1 = \max_j \sum_i |a_{ij}| \tag{7}$$

$$||A||_\infty = \max_i \sum_j |a_{ij}| \tag{8}$$

which are respectively the maximum column (absolute) sum and maximum row (absolute) sum.

The *condition number* of a matrix (relative to whichever norm is being used) is $\kappa(A) = ||A|| \cdot ||A^{-1}||$. It is used as a measure of the inherent difficulty of a linear algebra problem. The larger the condition number the more difficult it is to get a numerically stable solution. Typically, a large condition number implies that A has at least one very large or at least one very small eigenvalue. This will often result in large accumulated roundoff errors in the solution of a system. However, if the solution to such a system has a very small component in the direction of the eigenvector associated with such an eigenvalue, the solution may still be obtained to high accuracy (so that the *problem* is well-conditioned) despite the ill-conditioned nature of the matrix.

Computing the condition number of a matrix is of comparable difficulty (and computational complexity) to inverting the matrix in the first place. However there are good algorithms for estimating the condition number which may be used to estimate the number of correct digits in the solution to a linear system. These are discussed in [7].

Error estimation. The conventional way of measuring the error in the solution to a linear system is by what we might term a "relative norm error". Recently there has been extensive work on reanalysing methods in terms of a componentwise relative error which overcomes some of the overestimation of errors which may arise out of a well-conditioned problem which has an ill-conditioned matrix. (For more detail of this approach see [4].)

Suppose that the exact solution of $A\mathbf{x} = \mathbf{b}$ is $\mathbf{x}^*$ while the computed solution is $\widehat{\mathbf{x}}$. We denote by $\mathbf{r}$ the *residual* vector

$$\mathbf{r} = \mathbf{b} - A\widehat{\mathbf{x}}$$

The relative error in the solution can now be bounded by

$$\frac{1}{\kappa(A)}\frac{||\mathbf{r}||}{||\mathbf{b}||} \le \frac{||\widehat{\mathbf{x}} - \mathbf{x}^*||}{||\mathbf{x}^*||} \le \kappa(A)\frac{||\mathbf{r}||}{||\mathbf{b}||} \tag{9}$$

(This residual vector is also useful for carrying out iterative refinement of the computed solution by solving the system $A\mathbf{y} = \mathbf{r}$ using the LU factors and then adding $\mathbf{y}$ to the original solution $\widehat{\mathbf{x}}$.)

The other error bounds which are useful here give estimates of the effect of errors in the original matrix and/ or right-hand side vector. The effect of an error in $\mathbf{b}$ is estimated in a similar manner to the use of the residual above. If $\delta\mathbf{b}$ is the error vector (meaning that we have solved $A\widehat{\mathbf{x}} = (\mathbf{b} + \delta\mathbf{b})$ instead of $A\mathbf{x}^* = \mathbf{b}$) then we obtain

$$\frac{1}{\kappa(A)}\frac{||\delta\mathbf{b}||}{||\mathbf{b}||} \le \frac{||\widehat{\mathbf{x}} - \mathbf{x}^*||}{||\mathbf{x}^*||} \le \kappa(A)\frac{||\delta\mathbf{b}||}{||\mathbf{b}||}$$

which we see has just the same form as (9).

The corresponding bound for the situation where the matrix A has errors is

$$\frac{\|\hat{\mathbf{x}} - \mathbf{x}^*\|}{\|\mathbf{x}^*\|} \leq \kappa(A)\frac{\|\delta A\|}{\|A\|}$$

These various bounds can be combined to estimate the accuracy of a final solution in the presence of roundoff errors and, perhaps, noise. Again see either [3] or [7] for more details.

Effective rank. In the presence of noise (or even just roundoff error) most singular matrices would appear to be of full rank after the LU factorization is completed. There is considerable importance put on the detection of the "true" rank of the underlying matrix. This problem is of considerable importance in a variety of naval applications. The work of [12] represents the state-of-the-art in a signal processing environment. It is based on a statistical analysis of the SVD (singular value decomposition) algorithm to obtain the appropriate tolerance levels in order to get reliably accurate rank information. More recently Gleeson [6] has undertaken a study of the viability of using a (precomputed) statistical study of coefficients of the characteristic polynomial of the matrix in order to identify the "signature" of a matrix of given rank. The idea behind these studies is to try to obtain the rank without the need for a full eigenvalue or singular value analysis of the matrix.

Gauss elimination offers a cheap alternative to these approaches which is reliable most of the time. (This statement is contrary to that made in [12] which was based on a wasteful and numerically unstable implementation of GE. However in the case of an ill-conditioned matrix the GE approach still fails. Briefly the use of GE for effective rank determination is based on the observation that the noisy matrix is almost surely nonsingular as a result of the combination of roundoff and noise. Therefore all entries on the diagonal of the upper triangular factor will be nonzero and the algorithm will be successfully completed. By counting the number of these diagonal entries which are greater than some well-chosen threshold value, the "correct" rank is obtained.

The "correct" tolerance for this algorithm appears to be approximately nS where n is the dimension of the matrix and S is a bound for the elements of the noise matrix. With this tolerance most matrix rank problems are correctly resolved - but most is not necessarily sufficient.

The package MATLAB has a built-in rank function which can take a tolerance parameter. This algorithm is based on the singular value decomposition of the matrix and is therefore much more expensive to compute. There remains a question of what tolerance to use although again nS appears to be a good choice. This algorithm is *very* reliable provided that the tolerance is not comparable with the entries (and singular values) of A. An alternative algorithm [24] based on the use of least squares approximations of rows of A by linear combinations of other rows is currently under investigation in another NAWC project and appears to have some promise as being both reliable and of comparable cost to GE.

3. GAUSS ELIMINATION OVER THE INTEGERS

For computation over the integers, it is necessary to make changes to the basic GE algorithms described in Section 2. The most obvious cause is the fact that the integers are not closed under division. This has the side effect that the magnitudes of integers generated during the computation can grow rapidly. The range of integer values required or available is known as the *dynamic range*. Overflowing the integer range in binary integer arithmetic often results in the phenomenon known as "integer wraparound" (see [3], Chapter 1, for example) which is the effect of the "clock" arithmetic modulo 2^N where N is the integer wordlength in bits. Further complications that arise out of integer arithmetic (however it is performed) include choosing a good pivoting strategy.

In a Computer Algebra System (CAS) such as Maple [2] or Mathematica [27], some of these problems are avoided at the expense of computational speed since exact arithmetic with very long integers can be performed in software. However such computation becomes very slow when the arithmetic wordlength gets long — and the rate of growth of the dynamic range can be very rapid.

One way of overcoming some of the difficulties is the use of alternative representation and arithmetic formats for the integers. The residue number systems RNS are well-suited to some of these tasks. In such a system, an integer is represented by its residues modulo a number of different prime numbers. The advantage here is that the growth of the dynamic range is achieved by extending the set of basis primes being used. RNS arithmetic has a natural short wordlength parallelism which avoids the slowdown caused by very long wordlength arithmetic. However it brings with it other difficulties which are less easily resolved. Even if one integer divides another and both are within the dynamic range of the system, there is no simple division algorithm which returns this result; range checking is (at best) very difficult; comparison is not an RNS operation.

In this section, we discuss the implementation and use of GE using integer arithmetic. Our primary focus will be on binary integer arithmetic. The use of RNS arithmetic within the specific context of adaptive beamforming was discussed in a series of reports and papers [9], [10], [11], [21], [22], [23]. Specific reference to using RNS arithmetic will be included only where it is helpful to understanding in the present context.

3.1. Division-free Gauss elimination. The simplest way of modifying GE to integer arithmetic is to eliminate the divisions by performing "cross-multiplications" between the rows of the matrix. This is the form which generates the greatest rate of growth in the dynamic range. This corresponds to the transformation of 2×2 matrices $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ to $\begin{bmatrix} a & b \\ 0 & ad-bc \end{bmatrix}$ instead of $\begin{bmatrix} a & b \\ 0 & d-b(c/a) \end{bmatrix}$. This is achieved by multiplying the second row by a, and subtracting from it c times the first one. The overall divisionless GE algorithm consists of repeating this for all the appropriate submatrices as in the basic forms of the algorithm in Section 2.

As with the real arithmetic forms of GE there are many ways of arranging the operations. We concentrate here on just the simplest — the ijk-form corresponding to Algorithm 1.

Algorithm 11 Division-free GE ijk form

```
Input    $n \times n$ integer matrix $A$ (and right-hand side $\mathbf{b}$ if solving a system)
Compute
    for $i = 1$ to $n-1$
        for $j = i+1$ to $n$
            $b_j := a_{ii}b_j - a_{ji}b_i$        (if solving a system)
            for $k = i+1$ to $n$
                $a_{jk} := a_{ii}a_{jk} - a_{ji}a_{ik}$
```

$a_{ji} := 0$

Output (modified) matrix A (and $\mathbf{b}$ if solving a system)

Depending on the individual task being performed, this elimination algorithm would then be followed with the appropriate final stages — a modified back substitution for solving a system or other modifications of the algorithms of Section 2 for rank determination or determinant calculation. These are discussed later in Section 3.4 after the effect of this divisionless elimination on the resulting matrix elements has been analysed.

3.2. Growth in dynamic range. The question of the range growth in divisionless GE was addressed in some detail in [10], [11], [23] in order to analyse the possibilities for RNS arithmetic within the context of adaptive beamforming. In this section we begin by summarizing these results for general integer arithmetic. In order to develop satisfactory algorithms for the various underlying problems, it is also useful to examine the relation between the matrix entries arising from the divisionless algorithm and the corresponding algorithm using division. This question is addressed later in this subsection.

To get a feel for the potential range growth in the divisionless GE algorithm, consider first just a 2×2 matrix with integer entries in the range $[-M, M]$. Usual binary integer representations have a range of the form $[-M, M-1]$; but for the present purpose, we remark that the initial range does not necessarily match the available dynamic range. The symmetric range simplifies the analysis of the growth.

The transformation of the 2×2 matrix $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ to $\begin{bmatrix} a & b \\ 0 & ad-bc \end{bmatrix}$ results in an element $ad - bc \in [-2M^2, M^2]$. If $M = 2^K - 1$, this implies that the dynamic range required has increased from a required minimum wordlength of $K+1$ bits to $2K+2$ bits. This same exponential rate of growth is possible at every stage of the outer loop of Algorithm 11.

Thus the initial required wordlength is (approximately) doubled $N-1$ times during the divisionless GE elimination for an $N \times N$ matrix. The final wordlength needed is therefore around $2^{N-1}(K+1)$. It is easy to see that this would very quickly exhaust any normally available dynamic range.

For example, if K were just 3 so that the initial integer range is restricted to just $[-7, 7]$ with $N = 6$, the final dynamic range would need a wordlength of at least $2^5 \times 4 = 128$ bits which is already double the length of any commonly found built-in integer format. Clearly for a more realistic range for the initial data and larger matrix dimensions this growth would very quickly exceed all plausible ranges even for software implementation.

In the case of complex integer arithmetic as in [10] the growth is even slightly more rapid than the above analysis suggests and, perhaps more importantly, the worst case growth is achieved in realistic examples. Clearly it becomes necessary to find ways of restricting this growth. We discuss this aspect shortly but first need to get a clear picture of the comparative magnitudes of matrix elements resulting from the divisionless algorithm and that using division.

Comparison between matrix entries with and without divisions. In this subsection, we look in some detail at the relations between the matrix elements generated by the divisionless form of GE and those that would be obtained with divisions. This is important for the completion of the various applications especially computing the determinant which was especially straightforward for the basic algorithms of Section 2 using division. In order to make this comparison, we assume that the divisions can be performed exactly in some rational arithmetic system. Since these divisions only occur within an entirely theoretical version of the algorithm, the comparisons remain valid and useful.

In [10] it is stated that the the growth of the matrix elements is such that the final value of a_{NN} at the conclusion of the elimination phase is $\det A$. This turns out to be overoptimistic in general. The potential growth is much greater than this although it becomes apparent from the analysis that the algorithm can be modified to have this result. First we need some notation to distinguish between elements resulting from the different forms of GE under consideration. For simplicity here we shall completely ignore any pivoting and shall assume that the algorithm does not break down due to a zero pivot.

Following convention, we denote by a_{ij} the elements of the original matrix. All reference will be to the simplest ijk-form of GE described by Algorithm 1. By the i-th *stage* of the algorithm we mean the outermost loop for the value i which performs elimination below the diagonal in the i-th column. The entries resulting from the i-th GE with divisions are denoted by $a_{jk}^{(i)}$. The corresponding entries for the divisionless form Algorithm 11 will be denoted by $b_{jk}^{(i)} \quad (j,k > i)$. The "final" values in the two algorithms are therefore given by $a_{jk}^{(j-1)}$, $b_{jk}^{(j-1)} \quad (j = 0, 1, \ldots, N-1;\ k \geq j)$ where $a_{1k}^{(0)} = b_{1k}^{(0)} = a_{1k}$.

To see the relations between the entries $b_{jk}^{(i)}$ and $a_{jk}^{(i)}$, we look first at the transformation of $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$ to $\begin{bmatrix} a & b \\ 0 & ad-bc \end{bmatrix}$ as opposed to $\begin{bmatrix} a & b \\ 0 & d-b(c/a) \end{bmatrix}$. It is easy to see that, in the current notation,

$$b_{22}^{(1)} = ad - bc = a\,[d - b(c/a)] = a_{11}a_{22}^{(1)}$$

and that the corresponding relation would hold for all other matrix entries resulting from the first stage. That is

$$b_{jk}^{(1)} = a_{11}a_{jk}^{(1)} \qquad (2 \leq j,k \leq N) \tag{10}$$

In particular, we see that $b_{22}^{(1)}$ is the determinant of the top-left 2×2 submatrix. It will be convenient to denote this by d_2. In general, we shall denote by d_i the determinant of the top-left $i \times i$ submatrix of A.

The next stage of the elimination, is essentially the same as this first stage operating on the bottom-right $(N-1) \times (N-1)$ square submatrix — the "active matrix". It follows that there is a further scaling of all affected entries by the pivot element $b_{22}^{(1)}$. Thus, we deduce that

$$b_{jk}^{(2)} = a_{11}b_{22}^{(1)}a_{jk}^{(2)} \qquad (3 \leq j,k \leq N) \tag{11}$$

and continuing in this way we obtain the general relation:

$$b_{jk}^{(i)} = a_{11}b_{22}^{(1)} \cdots b_{ii}^{(i-1)}a_{jk}^{(i)} \qquad (i < j,k \leq N) \tag{12}$$

Each "final" divisionless entry is its corresponding value from Algorithm 1 scaled by the product of all the final diagonal entries *of the divisionless algorithm* above it.

Remark 15. *We note immediately that this analysis suggests that entries in the active matrix have common factors which have been included by the algorithm itself. These represent one obvious way of reducing the range growth and altering the algorithm. This modification, which is presented below, will result in each diagonal element being the determinant d_i of the principal minor of the appropriate dimension.*

3.3. Reducing the range growth. Before considering how to take full advantage of the enforced common factors which result from the divisionless algorithm, we look at the simplest range reduction technique resulting from the removal of unnecessary factors.

Greatest common divisors. The simplest approach is just to divide out any common factors in the "cross-multiplication" operations which are at the heart of the divisionless GE algorithm. Again, this idea is easily understood by considering the 2×2 elimination beginning with $A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$. Suppose that a, c have a common factor p so that there exist integers α, γ such that $a = p\alpha$, $c = p\gamma$. Then the usual divisionless transformation to $\begin{bmatrix} a & b \\ 0 & ad - bc \end{bmatrix}$ can be replaced by subtracting γ times the first row from α times the second to yield $\begin{bmatrix} a & b \\ 0 & \alpha d - b\gamma \end{bmatrix}$ since $\alpha c - a\gamma = \alpha p\gamma - p\alpha\gamma = 0$.

This suggests that it is desirable to find the *greatest common divisor*, $\gcd(a, c)$, of a, c before proceeding with the elimination. This same principle can be applied at each step of the process and it can be applied in more general settings than just integer arithmetic. For example, this idea is incorporated into the polynomial fraction-free Gauss elimination functions for Maple.

Further simplification may be achievable using the gcd in the (perhaps unlikely) event that a complete row of the matrix at some stage has a nontrivial common factor. This too can be divided out, reducing the range growth in subsequent stages of the elimination. Of course, if the goal is to obtain $\det A$ then a record of (the product of) these factors must be kept to multiply the final result. Finding the gcd of a complete row of a matrix of anything more than very small dimension may be at a price which does not justify the savings.

The simplest technique for finding $\gcd(a, c)$ is the Euclidean algorithm which is easily implemented as follows:

Algorithm 12 Euclidean algorithm for integer gcd

```
Input      Positive integers $c < a$
Initialize     $q := c$, $tmpc := a$
Repeat
    $tmpa := tmpc$; $tmpc := q$
    $q := tmpa \bmod tmpc$
until $q := 0$
Output $\gcd(a, c)$ is $tmpc$
```

This can be used within the following modified divisionless GE algorithm.

Fraction-free algorithm. A fraction-free version of GE can be defined using the gcd algorithm above to reduce the range growth effect. The resulting algorithm still needs no divisions.

Algorithm 13 Fraction-free GE using greatest common divisors *ijk* form

```
Input      $n \times n$ integer matrix $A$ (and right-hand side b if solving a system)
Compute
        for $i = 1$ to $n - 1$
            for $j = i + 1$ to $n$
        $p := \gcd(a_{ii}, a_{ji})$
        $\alpha := a_{ii}/p;\ \gamma := a_{ji}/p$
        $b_j := \alpha b_j - \gamma b_i$        (if solving a system)
        for $k = i+1$ to $n$
            $a_{jk} := \alpha a_{jk} - \gamma a_{ik}$
        $a_{ji} := 0$
Output (modified) matrix $A$ (and $\mathbf{b}$ if solving a system)
```

Remark 16. *Since we have no advance knowledge of the existence of nontrivial factors (that is $p > 1$) this algorithm does nothing to moderate the worst case range growth of the analysis in Section 3.2. However, if the wordlengths grow according to the needs of the particular application, it is likely that some savings will materialize.*

Using the gcd of complete rows can be incorporated into the algorithm at any stage. Potentially such factors could exist in any row of the active matrix and such factors could be sought in every row at every stage of the elimination. The algorithmic changes are straightforward but the cost is likely to be too high and so we do not elucidate further here — except for one very important special situation which does arise as a result of the elimination process itself.

From (11) it follows, in particular, that

$$b_{33}^{(2)} = a_{11} b_{22}^{(1)} a_{33}^{(2)} = a_{11} \left(a_{11} a_{22}^{(1)} \right) a_{33}^{(2)} = a_{11} d_3$$

where, we recall, $d_3 = a_{11} a_{22}^{(1)} a_{33}^{(2)}$ is the determinant of the principal 3×3 minor. We remark that d_3 is an integer. It follows that $b_{33}^{(2)}$ has the factor a_{11}. Using this same reasoning, we see that $b_{jk}^{(2)}$ has a factor a_{11} for every $j, k \geq 3$ since each such element is a_{11} times the (integer) determinant of a 3×3 minor. This known factor can easily be removed prior to the next stage of the elimination.

Similar factors are introduced into the active matrix at each subsequent stage. These too can be divided out. The factors introduced at the subsequent stages of the elimination are the (modified) diagonal entries. Since these are known factors, there is no need for any calls to a gcd function.

The removal of these factors has an obvious effect on the range growth in subsequent stages of the elimination even though the worst case analysis is unchanged. The growth in the required wordlength would need to be computed "on-the-fly" in order to take advantage of this saving.

The division by these factors is incorporated into Algorithm 14 below. We observe that this algorithm, like Algorithm 13, is fraction-free but not division-free. All divisions that are performed are *integer operations with exact integer results.*

Algorithm 14 Fraction-free GE ijk form

```
Input    $n \times n$ integer matrix $A$ (and right-hand side $\mathbf{b}$ if solving a system)
Compute
    for $i = 1$ to $n-1$
        for $j = i+1$ to $n$
            $b_j := a_{ii} b_j - a_{ji} b_i$        (if solving a system)
            for $k = i+1$ to $n$
        $a_{jk} := a_{ii}a_{jk} - a_{ji}a_{ik}$
    $a_{ji} := 0$
  if $i \geq 2$ then            (removal of common factors)
    for $j = i + 1$ to $n$
    $b_j := b_j / a_{i-1,i-1}$        (if solving a system)
      for $k = i + 1$ to $n$
        $a_{jk} := a_{jk} / a_{i-1,i-1}$
Output (modified) matrix $A$ (and $\mathbf{b}$ if solving a system)
```

To gain some insight into the saving that results from this algorithm, consider just the final value of a_{NN}. In the division-free Algorithm 11, the analysis of Section 3.2 yields, using (12) for $i = N - 1$:

$$b_{NN}^{(N-1)} = a_{11}b_{22}^{(1)} \cdots b_{N-1,N-1}^{(N-2)}a_{NN}^{(N-1)}$$

which in turn yields

$$\begin{aligned} b_{NN}^{(N-1)} &= a_{11}a_{22}^{(1)} \cdots a_{N-1,N-1}^{(N-2)}a_{NN}^{(N-1)} \left\{ a_{11}^{N-2} \left[b_{22}^{(1)} \right]^{N-3} \cdots b_{N-2,N-2}^{(N-3)} \right\} \\ &= d_N \left\{ a_{11}^{N-2} \left[b_{22}^{(1)} \right]^{N-3} \cdots b_{N-2,N-2}^{(N-3)} \right\} \end{aligned} \tag{13}$$

The corresponding final value for Algorithm 14 is just

$$a_{NN} = \det A = d_N \tag{14}$$

which is typically very much smaller since these additional factors have been removed during the modified elimination. Indeed it turns out for this Algorithm 14 that at the conclusion of the elimination we have

$$a_{ii} = d_i. \tag{15}$$

Example 3. *Comparison of Algorithms 11 and 14 for a 4×4 matrix. The results of exact arithmetic in Algorithm 1 are also included for comparison and illustration.*

Let

$$A = \begin{bmatrix} 8 & 7 & 4 & 1 \\ 4 & 6 & 7 & 3 \\ 6 & 3 & 4 & 6 \\ 4 & 5 & 8 & 2 \end{bmatrix}$$

Algorithm 1 yields the following sequence of modified matrices:

$$\begin{bmatrix} 8 & 7 & 4 & 1 \\ 0 & 5/2 & 5 & 5/2 \\ 0 & -9/4 & 1 & 21/4 \\ 0 & 3/2 & 6 & 3/2 \end{bmatrix} \quad \begin{bmatrix} 8 & 7 & 4 & 1 \\ 0 & 5/2 & 5 & 5/2 \\ 0 & 0 & 11/2 & 15/2 \\ 0 & 0 & 3 & 0 \end{bmatrix} \quad \begin{bmatrix} 8 & 7 & 4 & 1 \\ 0 & 5/2 & 5 & 5/2 \\ 0 & 0 & 11/2 & 15/2 \\ 0 & 0 & 0 & -45/11 \end{bmatrix}$$

from which we may deduce that $\det A = (8)(5/2)(11/2)(-45/11) = -450$.

The division-free Algorithm 11 gives:

$$\begin{bmatrix} 8 & 7 & 4 & 1 \\ 0 & 20 & 40 & 20 \\ 0 & -18 & 8 & 42 \\ 0 & 12 & 48 & 12 \end{bmatrix} \quad \begin{bmatrix} 8 & 7 & 4 & 1 \\ 0 & 20 & 40 & 20 \\ 0 & 0 & 880 & 1200 \\ 0 & 0 & 480 & 0 \end{bmatrix} \quad \begin{bmatrix} 8 & 7 & 4 & 1 \\ 0 & 20 & 40 & 20 \\ 0 & 0 & 880 & 1200 \\ 0 & 0 & 0 & -576000 \end{bmatrix}$$

which shows the very rapid growth which is possible with this algorithm.

The fraction-free Algorithm 14 also gives

$$\begin{bmatrix} 8 & 7 & 4 & 1 \\ 0 & 20 & 40 & 20 \\ 0 & -18 & 8 & 42 \\ 0 & 12 & 48 & 12 \end{bmatrix} \quad \begin{bmatrix} 8 & 7 & 4 & 1 \\ 0 & 20 & 40 & 20 \\ 0 & 0 & 880 & 1200 \\ 0 & 0 & 480 & 0 \end{bmatrix}$$

but the active part of this matrix is then divided by the common factor 8 to yield

$$\begin{bmatrix} 8 & 7 & 4 & 1 \\ 0 & 20 & 40 & 20 \\ 0 & 0 & 110 & 150 \\ 0 & 0 & 60 & 0 \end{bmatrix}$$

which in turn gives

$$\begin{bmatrix} 8 & 7 & 4 & 1 \\ 0 & 20 & 40 & 20 \\ 0 & 0 & 110 & 150 \\ 0 & 0 & 0 & -9000 \end{bmatrix}$$

The active matrix is now just the bottom-right element which needs to be divided by the "common factor" 20 to yield:

$$\begin{bmatrix} 8 & 7 & 4 & 1 \\ 0 & 20 & 40 & 20 \\ 0 & 0 & 110 & 150 \\ 0 & 0 & 0 & -450 \end{bmatrix}$$

The final values of the diagonal entries can easily be seen (by comparison with the partial products of the diagonal of the final upper triangle generated by Algorithm 1) to be the determinants of the appropriate principal minors, as predicted by (14) and (15).

Although there has been some growth in the magnitudes of the matrix elements it has been kept in much better check. The whole of this computation could have been achieved using standard 16-bit integer arithmetic - even including the temporary values. The division-free algorithm in this case requires at least 20 bits even though the original matrix has integer elements bounded by 8.

Remark 17. *It is necessary to observe that the various fraction-free versions of GE are well-suited to conventional binary integer arithmetic - but cannot easily be applied in other integer arithmetic implementations such as RNS since the divisions can not be performed within the RNS system itself. This is true even in the ideal situation of Algorithm 14 where all divisions are known to have exact integer results.*

3.4. Effect on computation. In this section, we describe the effect of the modified fraction-free GE Algorithm 14 on the solution of the various underlying problems. From the observations in the last section, it is plain that the evaluation of $\det A$ is particularly simple, provided only that no zero pivots arise during the computation.

Linear systems. For the solution of a (nonsingular) linear system using Algorithm 14, there is of course no guarantee that the solution vector has only integer components. If the system is known (because of the context, perhaps) to have an integer solution then this can be computed by applying the conventional back substitution (Algorithm 7 or Algorithm 8, for example) in which case the divisions will again have integer results.

Otherwise the solution can be expressed as fractions by performing rational arithmetic. In its column form equivalent to Algorithm 8, the rational arithmetic back substitution can be described as follows. The algorithm below makes use of a function reduce $[p, q]$ which simply reduces a rational number p/q to its normal form by removing any common factors.

Algorithm 15 Back substitution using rational arithmetic — column form

```
Input      $n \times n$ upper triangular matrix $U$, $n$-vector $\mathbf{b}$
Initialize rational solution   $\mathbf{m} := 0$; $\mathbf{n} := 1$     (solution will be $x_i = m_i/n_i$)
Rationalize right-hand side   $\mathbf{p} := \mathbf{b}$; $\mathbf{q} := 1$     (right-hand side $b_i = p_i/q_i$)
Compute
    for $j = n$ downto 1
        $m_j := p_j$; $n_j := q_j * u_{jj}$
        reduce $[m_j, n_j]$
        for $i = 1$ to $j - 1$
            $p_i := p_i * n_j - u_{ij} * m_j * q_i$
            $q_i := q_i * n_j$
            reduce $[p_i, q_i]$
Output solution $[\mathbf{m}, \mathbf{n}]$
```

Remark 18. *Note that the inner loop call to* reduce $[p_i, q_i]$ *could be eliminated at little expense. It's removal would have the benefit that the same denominator would be used for each b_i for $i < j$. The cost in terms of dynamic range would likely be minimal since the required range would only be decreased if every call to* reduce $[p_i, q_i]$ *for some value of j results in a smaller value of q_i.*

Example 4. *Consider the coefficient matrix of Example 3 with the original right-hand side vector* $[25, 20, 25, 20]^T$.

Algorithm 14 would generate the augmented matrix

$$\left[\begin{array}{cccc|c} 8 & 7 & 4 & 1 & 25 \\ 0 & 20 & 40 & 20 & 60 \\ 0 & 0 & 110 & 150 & 260 \\ 0 & 0 & 0 & -450 & -450 \end{array}\right]$$

Applying Algorithm 15, we first get, for $j = 4$, $m_4 = -450$, $n_4 = -450$ and the result of reduce $[m_4, n_4]$ is then $m_4 = n_4 = 1$. The inner loop, with the further call to reduce, then yields $p_1 = 24, p_2 = 40, p_3 = 110$ and $q_1 = q_2 = q_3 = 1$. Next, with $j = 3$, we get $m_3 = n_3 = 1$ and then $p_1 = 20, p_2 = 0$ with, still, $q_1 = q_2 = 1$. The next iteration of the j-loop gives $m_2 = 0, n_2 = 1, p_1 = 20$ and $q_1 = 1$ from which, finally, we get $m_1 = 20, n_1 = 8$ which reduces to $m_1 = 5, n_1 = 2$ so that the complete rational solution is $[5/2, 0/1, 1/1, 1/1]^T$. In this particular case the reduction in the inner loop, had no effect since the outer loop reduction resulted in $q_2 = q_3 = q_4 = 1$.

Determinant. We have already seen that Algorithm 14 delivers det A automatically as the final value of a_{NN}.

Rank determination. In the absence of a zero pivot, again there is nothing further to be done. In the event of such a zero, then interchanging the pivot row with any lower row with a nonzero entry in the pivot column will allow the fraction-free algorithm to proceed. Simply counting the number of nonzero entries on the diagonal gives the rank, as before. This simple interchange is not necessarily the optimal pivoting strategy for integer computation.

In the event of a rank-deficient matrix, obtaining the solution space for an underdetermined system can be accomplished by modifying Algorithm 15 in just the same way as was proposed in Section 2.3 for real-number computation.

3.5. Pivoting. Clearly in the event of a zero pivot some pivoting is necessary in any good implementation of GE. The question is what is the right strategy for integer computing? Choosing the largest element in the floating-point algorithm has the virtue of keeping all multipliers small and therefore restricting the growth of the matrix elements. However that restricted growth is only realized because of the divisions that are performed in Algorithms 1 through 6.

At least intuitively, choosing the largest element in the pivot column is not necessarily good for integer computation. Indeed a large prime pivot is probably near-worst since that appears to almost guarantee rapid growth in the dynamic range. In (11) and then (12) we see that each $b_{jk}^{(2)}$ has the factor a_{11} and each further stage has this factor and then has it repeated in the factors $b_{ii}^{(i-1)}$. The earlier a particular pivot is used the greater the impact it has on subsequent range growth. This is made plain in equation (13).

In the fraction-free version Algorithm 14, however, the range growth is essentially independent of the order of the use of the rows. In the case of a full rank matrix A, the final entry $a_{NN} = d_N$ as in (14) is necessarily the largest of the principal minor determinants and this value is invariant (up to sign changes) under row interchanges. It follows that the simplest pivoting strategy is probably the best for this algorithm. That is, at stage i, we should use the first row for which the pivot column entry is nonzero: choose $pivot = \min\{p \geq i : a_{pi} \neq 0\}$.

It is conceivable that a different ordering may represent some minor improvement on this. For example, looking for pivots which are either powers of the binary (or other computational) base may make subsequent divisions particularly simple. However, searching for the appropriate pivot element in this regard would (almost surely) be more wasteful than beneficial.

3.6. Complexity revisited. In this section, we begin by obtaining the basic integer operation counts for the fraction-free GE Algorithm 14. There are two essential differences between these counts and those for the floating-point algorithms presented in Section 2.4: the fundamental elimination loops contain no divisions but have twice as many multiplications, and there is the additional complexity of the removal of the common factors after the first two stages.

The additional multiplications in the elimination are easily counted. The common factor removal entails only divisions. The total number of these can be assessed from the loop control limits. For each $i \geq 2$, there is one division in the j-loop itself (assuming we are solving a system) which runs from $j = i+1$ to n. Also in this loop is the k-loop which has the same limits and contains another division. The total division count is therefore

$$\sum_{i=2}^{n-1}(n-i)(n-i+1) = \sum_{l=1}^{n-2} l(l+1) = \frac{1}{3}n(n-1)(n-2)$$

The total operation counts are summarized in Table 4.

TABLE 4 Integer arithmetic operation counts for the fraction-free GE Algorithm 14 for solution of an $n \times n$ linear system $Ax = \mathbf{b}$.

Operation	**Matrix Factorization**	**Right-hand side**	**TOTAL**
+ or −	$\frac{1}{3}n\,(n^2-1)$	$\frac{1}{2}n\,(n-1)$	$\frac{1}{6}n\,(n-1)\,(2n+5)$
×	$\frac{2}{3}n\,(n^2-1)$	$n\,(n-1)$	$\frac{1}{3}n\,(n-1)\,(2n+5)$
/	$\frac{1}{6}(n-2)\,(n-1)\,(2n-3)$	$\frac{1}{2}(n-1)(n-2)$	$\frac{1}{3}n\,(n-1)\,(n-2)$

Note that this operation count does *not* include the back substitution Algorithm 15. What we see most importantly is that the number of multiplications has doubled *and*, even worse, the number of divisions has increased by a complete order from $O(n^2)$ to $O(n^3)$.

Unfortunately this is not the end of the story. The complexity of this algorithm is further increased by virtue of the fact that these integer divisions are typically more difficult than their floating-point counterparts — especially with the range growth which we have already seen can be substantial. This is likely to necessitate the use of *long* integer arithmetic for which special algorithms must be used.

3.7. Arithmetic with *long* integers. In this section we discuss briefly some aspects of the long integer arithmetic which is required for integer Gauss elimination algorithms. Long integer arithmetic can be simulated using multiple words of some basic wordlength.

For example, if the underlying integer wordlength is 8 bits (or 1 byte) then such an integer can be regarded as a radix $2^8 = 256$ digit. Conventionally the range of values of the base 256 digits would be $-128, -127, \ldots, 127$ so that signed integers can be represented. For simplicity in the current description, we restrict our attention to nonnegative integers with a digit range of $0, 1, \ldots, 255$. Very large integers could then be stored using a vector of such integers using conventional place value.

In general, suppose the base wordlength is L bits so that the effective radix is $R = 2^L$. The vector $(d_0, d_1, \ldots, d_{K-1})$ would then be used to represent the integer $N \in [0. R^K - 1]$ given by

$$N = d_{K-1}R^{K-1} + d_{K-2}R^{K-2} + \cdots + d_1 R + d_0 \tag{16}$$

where each digit satisfies $0 \leq d_i < R$. For efficient arithmetic using such a representation we require an integer accumulator with $2L$ bits. Among other consequences of this are that addition can be computed in "digit-parallel" with subsequent attention to any carries which may be propagated. A large radix carry lookahead or conditional sum adder could be constructed to improve the efficiency of addition. These addition algorithms are simple generalizations of the usual binary addition algorithms which are described, for example, in [19].

Alternative approaches to long wordlength integer arithmetic are provided by using Residue Number Systems, RNS. This is a naturally parallel integer arithmetic which does not readily admit division. It is therefore not very suitable for use with the fraction-free Gauss elimination, Algorithm 14. It has been considered in detail for naval applications in [9], [10] and [11].

Convolution form of product. Multiplication of long integers can be achieved with reasonable efficiency using the convolution form of the product working with the component words of the large integers. Again for simplicity, we shall only consider multiplication of positive integers. The multiplication of two integers in the form (16) will result in an integer whose representation requires at most $2K$ L-bit words.

Suppose that we require the product of the long integers

$$m = \sum_{i=0}^{K-1} \alpha_i R^i, \qquad n = \sum_{i=0}^{K-1} \beta_i R^i$$

This product is given by

$$m * n = \left(\sum_{i=0}^{K-1} \alpha_i R^i \right) \left(\sum_{j=0}^{K-1} \beta_j R^j \right) = \sum_{k=0}^{2K-2} \left(\sum_{i=0}^{K-1} \alpha_i \beta_{k-i} \right) R^i \tag{17}$$

where we use the convention $\beta_j = 0$ if $j < 0$. This is essentially a convolution product of the coefficient vectors.

Now each coefficient product $\alpha_i \beta_{k-i}$ can be viewed as a base-R^2 digit which we can write in the form $\alpha_i \beta_{k-i} = \gamma_{k,i} R + \delta_{k,i}$ where each $\gamma_{k,i}$ and $\delta_{k,i}$ is a base-R digit. Thus $\gamma_{k,i}$ is the most significant and $\delta_{k,i}$ the least significant R-digit of $\alpha_i \beta_{k-i}$. The product (17) can therefore be written as

$$m * n = \sum_{k=0}^{2K-2} \left(\sum_{i=0}^{K-1} \gamma_{k,i} R + \delta_{k,i} \right) R^i = \sum_{k=0}^{2K-1} \left(\sum_{i=0}^{K-1} \gamma_{k-1,i} + \delta_{k,i} \right) R^i \tag{18}$$

where $\gamma_{k,i} = \delta_{k,i} = 0$ for $i > k$.

Remark 19. *Carries beyond the R^{2K-1} position is not possible since $m*n \leq (R^K - 1)^2 < R^{2K}$.*

Remark 20. *The results of the inner sums in (18) will usually create further carries which must be accounted for. However, we note that for almost any minimal degree of parallelism in the processor, each of these inner sums can be performed simultaneously since they consist of at most $2K$ terms each less than R and we would expect $2K \ll R$. Therefore the sum will be less than R^2 so that it will be representable in a double length accumulator.*

For the 8-bit basic wordlength the restriction is only that our integers do not exceed $(2^8)^{256} = 2^{2048} > 10^{616}$ which is well beyond any typical integer computing range for linear algebra problems.

The length of the inner sums in (18) will also restrict the size of any carry and therefore may be useful in bounding the range of the propagation of such carries. This could be used to improve the efficiency of such a convolution product. We do not consider such details further in this report.

What effect does such a long multiplication have on the arithmetic complexity of an integer algorithm? The product formula (17) entails K^2 basic multiplications. these components must then be broken into their two component digits and then approximately K^2 additions together with the carries these generate. In Table 4, this means that each of the (approximately) $\frac{2}{3}n^3$ multiplications entails something like $2K^2$ regular integer arithmetic operations. For even quite moderate range growth with $K = 4$ this has the effect of increasing this part of the complexity by a factor of 32.

Division. Division of long integers can also be achieved by generalizing some of the standard algorithms which are used in binary integer hardware such as the SRT division algorithm which is based on the idea of nonrestoring division. This algorithm is well-suited to high-radix division and so can be modified to the long integer framework.

The SRT algorithm relies on repeated addition and subtraction using signed digits. Since we have only dealt with arithmetic of nonnegative long integers here, we do not discuss the detailed implementation further. For details of the basic algorithm and its implementation at least for radix 4, see [19].

3.8. Should we use real arithmetic anyway?. Unless roundoff errors are severe, it may well be prudent to compute the solutions to integer linear algebra problems by applying the appropriate real, floating-point, algorithm and rounding the results. All the difficulties of range growth, avoiding divisions or additional complexity are then removed.

From the error analysis summarized in Section 2.5, we can obtain estimates of the number of correct digits in the final result. First note that since the underlying problem has an all-integer coefficient matrix (and right-hand side) then we may assume that we have exact data. The only contribution to the error in the solution of such a linear system therefore arises from the roundoff errors of floating-point arithmetic. Only for very ill-conditioned matrices is the upper bound on the error given by the right-hand side of the inequality (9) likely to indicate that the rounded solution of an integer system may be incorrect. With the improved elementwise bounds obtained by Demmel and others even greater confidence in the computed results would be obtained. (See [4] for an introduction to the literature on this subject.)

We have also seen that the complexity of integer-arithmetic GE is increased relative to the floating-point algorithms. Compare Tables 2 and 4. By its nature the fraction-free Algorithm 14 is "row-oriented" which may render it advantageous on some parallel architectures. for instance the removal of the common factors would be a simple array-operation on a massively parallel array processor. However whatever advantages could be realized for this algorithm could also be obtained for the appropriate arrangement of the original LU factorization algorithm such as Algorithms 4, 5 or 6.

On most modern computer architectures the speed of floating-point operations is comparable with all but the shortest of integer wordlengths. The range growth problem of integer computation makes such short wordlengths impractical for most problems. If, using floating-point, the arithmetic is essentially no slower, the complexity is reduced because division is available and the accuracy of integer arithmetic can be recovered in the final results, it seems this (real arithmetic) would be better in most circumstances.

4. OVER THE RATIONALS

In this section,we consider the use of Gauss elimination in the setting of rational arithmetic. Of course in some sense, floating-point *is* rational arithmetic but it uses a very special subset of the rationals and does not comply with the axioms of conventional rational arithmetic. We are concerned here with matrices with entries in the field **Q** of rational numbers. The arithmetic operations will be similar in nature to those of Algorithm 15 for back substitution in the integer GE solution of a linear system.

4.1. Rational representations.

There are choices to be made over the way in which rational numbers are to be stored and manipulated within the computer. The conceptually simplest option is simply to store $q \in \mathbf{Q}$ as a pair of integers representing its numerator and (positive) denominator in its maximally reduced form. Alternatives that have been extensively researched include the use of continued fraction representations and, in particular, the lexicographic continued fraction, or LCF, arithmetic of Kornerup and Matula [13], [14], [15], for example.

$[n/m]$ form. The standard representation of rational numbers is as a quotient of two co-prime integers

$$q = \frac{n}{m}$$

where $n, m \in \mathbf{Z}$ have no common factors and $m > 0$. Arithmetic operations are then defined according to the usual rules of rational arithmetic with reduction to this "normalized" form after each arithmetic operation.

It is immediately apparent that many of the range growth problems which plague integer GE will reappear here with comparable severity. Of course the divisions which are inherent in the basic forms of GE and LU factorization can be performed here and restrict the range of values of the rational numbers being represented - *but* these divisions do not necessarily reduce the range of integers needed for the numerators and denominators separately.

The chief virtue that would be derived here is the elimination of any rounding errors and therefore definitive answers to questions such as the rank of the matrix and exact values for the determinant and for the solution vector of a linear system.. This last would be computed using the (obvious and minor) modification of Algorithm 15.

Use of continued fractions. An alternative representation of rational numbers is provided by using continued fractions. This possibility has been extensively explored by Kornerup and Matula in a substantial series of papers including [13], [14] and [15]. In the later papers in this series, techniques are described for performing rational arithmetic in a bit-pipelined manner. This has the effect of allowing many simple rational functions of arguments expressed in continued fraction form to be computed much more quickly than might otherwise be the case. This is based in part on the use of a particular encoding of the numbers which has retains the natural ordering of the representations. This is called the lexicographic continued fraction representation or LCF. Its details are not important here.

The continued fraction representation of a positive rational number is given by

$$q = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + }}} \tag{19}$$

where the integers $a_0 \geq 0$, and $a_i \geq 1$ for $i \geq 1$.

The manifestation of the range growth problem here lies not in the *magnitude* of the various coefficients so much as in the *number* of them; that is, the length of the string $a_0, a_1, a_2, \ldots$ used to represent a particular rational number. For details of LCF arithmetic, its implementation and its use the reader is referred to the original papers. This section is included solely to increase awareness of the existence of alternative forms of rational arithmetic which could be used.

4.2. GE algorithm for rational arithmetic. The algorithm for performing the factorization of a rational matrix can be any of the variants discussed in Section 2 with the real arithmetic operations replaced by rational arithmetic. There is no benefit in detailing all these algorithms explicitly. In order to discuss the dynamic range growth in this context however it is helpful to have at least the basic ijk-form described for this context. We shall denote the original matrix elements by

$$a_{ij} = \frac{p_{ij}}{q_{ij}}$$

and those of the right-hand side vector by

$$b_i = \frac{r_i}{s_i}$$

where all fractions are assumed to be in reduced form.

Algorithm 16 Basic GE for rational arithmetic ijk form

```
Input    $n \times n$ matrix rational $A$ (and right-hand side b if solving a system)
Compute
      for $i = 1$ to $n-1$
          for $j = i+1$ to $n$
              $m_{ji} := p_{ji}q_{ii}$;   $n_{ji} := p_{ii}q_{ji}$
              reduce $[m_{ji}, n_{ji}]$
              $r_j := r_j n_{ji} s_i - s_j m_{ji} r_i$;   $s_j = s_j n_{ji} s_i$        (for system)
              reduce $[r_j, s_j]$
              for $k = i+1$ to $n$
                  $p_{jk} := p_{jk} n_{ji} q_{ik} - q_{jk} m_{ji} p_{ik}$;   $q_{jk} = q_{jk} n_{ji} q_{ik}$
                  reduce $[p_{jk}, q_{jk}]$
              $p_{ji} := 0$;   $q_{ji} := 1$
Output (modified) matrix $A$ (and b if solving a system)
```

Remark 21. *Algorithm 16 is just the basic Algorithm 1 modified for rational arithmetic. It is plain that the arithmetic complexity has been increased considerably as a result of storing each element as a quotient of integers and by the need for reduction of these rational numbers to their "normalized" form.*

No pivoting has been incorporated here and as with the standard integer setting it is not obvious what the optimal pivoting strategy would be. In the above algorithm it is simply assumed that no zero pivots are encountered. If such a pivot element does in fact arise we can simply interchange the pivot row with a row which has a nonzero entry in the pivot column.

From the point of view of range growth there may be some merit in choosing pivots to be as "simple" as possible. Simple here would correspond roughly to having the smallest denominator (or maybe the denominator whose largest prime factor is minimum). Such a search for the simplest pivot may well be more expensive than is merited by any reduction which may be achieved in the dynamic range. The conventional argument for pivoting (apart from avoiding zero pivots) is not relevant to either integer or rational arithmetic since roundoff error, and the associated question of numerical stability, are not concerns.

4.3. Complexity. The most obvious increase in the complexity in Algorithm 16 arises out of the mere fact that it is performing rational arithmetic and so every arithmetic operation entails manipulation of both the numerator and denominator. There is also the further complication arising from the reduction of each ordered pair to represent an irreducible fraction. This requires either that each integer is stored as a product of its prime factors, or more reasonably, that the Euclidean algorithm, Algorithm 12, for finding the gcd of two integers is employed and followed with two integer divisions. Because the Euclidean algorithm is iterative we cannot determine the number of integer mod operations that are required. (Also any bounds which could be derived would be hopelessly pessimistic.)

In Table 5, we list the numbers of conventional integer arithmetic operations together with the number of gcd's that are needed. Typically we might expect that a gcd would be equivalent to several integer divisions and that these divisions are, in turn, equivalent to several multiplications. However it should also be noted that, because of the range growth, the multiplications may involve long wordlength integers. On the other hand we may expect the gcd to be much smaller so that the divisor wordlengths may be less extreme. In the operation counts in Table 5, no attempt is made to account for dynamic range considerations or the relative weights to be given to the different operations.

TABLE 5 Integer arithmetic operation counts for the rational GE Algorithm 16 for solution of an $n \times n$ linear system $A\mathbf{x} = \mathbf{b}$.

Operation	Matrix Factorization	Right-hand side	TOTAL
+ or −	$\frac{1}{3}n\left(n^2-1\right)$	$\frac{1}{2}n\left(n-1\right)$	$\frac{1}{6}n\left(n-1\right)\left(2n+5\right)$
×	$2n^2\left(n-1\right)$	$3n\left(n-1\right)$	$n\left(n-1\right)\left(2n+3\right)$
/	$\frac{2}{3}n\left(n^2-1\right)$	$n(n-1)$	$\frac{1}{3}n\left(n-1\right)\left(2n+5\right)$
gcd	$\frac{1}{3}n\left(n^2-1\right)$	$\frac{1}{2}n(n-1)$	$\frac{1}{6}n\left(n-1\right)\left(2n+5\right)$

Again the biggest single effect is that the number of divisions has increased to $O(n^3)$ in addition to the $O\left(n^3\right)$ gcd operations. This time the number of multiplications has also increased by a substantial factor. The overall operation counts suggest that the cost of rational computation may be too high to justify the increased stability and accuracy — especially when the range growth and resulting additional complexity of these operations is considered too.

4.4. Growth in dynamic range. In attempting to analyse the potential growth in the necessary dynamic range for implementing Algorithm 16, we cannot assume any useful reduction in the rational quantities other than that which is a necessary consequence of the elimination procedure itself. For simplicity in this setting, we make the assumption that the original matrix A is in fact an *integer* matrix. This allows us to compare the results of the (rational) Algorithm 16 with those that would be obtained using the divisionless integer version Algorithm 11. In much the same way as was described for the fraction-free Algorithm 14, we shall see that this comparison reveals certain common factors which will be removed by the various reduction steps. This has the effect of reducing the potential range growth in a similar manner to that which we observed in Algorithm 14.

As for the analysis in Section 3.3, we shall concentrate on the matrix itself. Similar analysis is valid for the right-hand side of a linear system but adds nothing to the overall picture. Again following the earlier analysis, we shall denote by a_{ij} the integer elements of the original matrix. The results of the i-th stage of the integer Algorithm 11 will again be denoted by $b_{jk}^{(i)}$. The corresponding rational values will be denoted by $c_{jk}^{(i)}$. These are of course just rational representations of the quantities $a_{jk}^{(i)}$ generated by the original Algorithm 1.

During the first stage of the elimination, the multipliers used are (irreducible rational representations) of

$$m_{j1} = \frac{a_{j1}}{a_{11}}$$

Except in the unlikely event that all elements in the first column of A share a common factor, the range for the denominator cannot be reduced. The results of the first stage are

$$c_{jk}^{(1)} = a_{jk} - \frac{a_{j1}}{a_{11}} a_{1k} = \frac{a_{11}a_{jk} - a_{j1}a_{1k}}{a_{11}} = \frac{b_{jk}^{(1)}}{a_{11}} \tag{20}$$

and again unless there is some general cancellation the range growth for the numerator is identical to that for Algorithm 11 at this stage.

Similarly the second stage results in

$$c_{jk}^{(2)} = c_{jk}^{(1)} - \frac{c_{j2}^{(1)}}{c_{22}^{(1)}} c_{2k}^{(1)}$$

and we observe that in light of (20) the multiplier here is the same as would be used with the integer matrix. Indeed, substituting (20) into the last equation we get

$$c_{jk}^{(2)} = \frac{b_{jk}^{(1)}}{a_{11}} - \frac{b_{j2}^{(1)}b_{2k}^{(1)}}{b_{22}^{(1)}a_{11}} = \frac{b_{22}^{(1)}b_{jk}^{(1)} - b_{j2}^{(1)}b_{2k}^{(1)}}{b_{22}^{(1)}a_{11}} = \frac{b_{jk}^{(2)}}{b_{22}^{(1)}a_{11}} \tag{21}$$

However, we recall from Section 3.3 that each $b_{jk}^{(2)}$ has the factor a_{11} which will therefore be removed by the rational reduction step. That is the only guaranteed factor which can be used to help control the dynamic range growth. The range required for teh numerator at this stage is therefore precisely the same as that for teh fraction-free Algorithm 14. The denominator (assuming no further general common factor is present) is $b_{22}^{(1)} = d_2$, the determinant of the 2×2 principal minor.

Because of this common denominator, the next stage is again essentially equivalent to that for Algorithm 14 since $b_{22}^{(1)}$ becomes a common factor in the remaining active matrix. Ultimately then the range growth for Algorithm 16 is identical to that which is encountered in the fraction-free Algorithm 14. The inference to be drawn from this analysis is that if the initial matrix is an integer matrix, using rational arithmetic yields no benefit relative to the fraction-free algorithm.

If the original matrix consists of rational entries, the range growth problem becomes potentially even more critical since, the "cross-multiplications" needed for the elimination do not appear to generate any obvious and general common factors. Unless the original denominators are small and have similar factors it seems unlikely that any purely rational algorithm will be practical and the best solution would almost certainly be to use real (floating-point) arithmetic.

5. General rings

In this section, we consider the applicability (and application) of GE in a more general algebraic setting. There are at least two fundamental questions which arise immediately:

"When do the problems have solutions?" and

"When do the algorithms make sense?"

We are interested here in matrices with entries in a general ring $\mathcal{R}$. In order to answer the fundamental questions, we shall need to impose further conditions on this ring. For details of the definitions and properties of the various algebraic structures see [5] or any standard text on abstract algebra. The key properties which are being used will be summarized as they are introduced here.

In essence a ring $\mathcal{R}$ is a structure which has both "addition" and "multiplication" defined on it. The addition has all the properties associated with the various number systems. The multiplication need not have all the properties of the real number system. In a general ring, all we may assume is that the multiplication is associative and that it is distributive over addition. "Vector spaces" over rings are called *modules*. The concepts of linear independence, basis and scalar multiplication carry over in a natural way to modules.

Special cases of rings include *fields* of which the rational, real and complex numbers $\mathbf{Q}$, $\mathbf{R}$ and $\mathbf{C}$ are the most important examples, and *integral domains* such as the integers $\mathbf{Z}$ and the ring of polynomials over the real numbers $\mathbf{R}[x]$. In a field, multiplication and division (except by zero) are defined and have the expected properties. In an integral domain, the multiplication is commutative, there is a multiplicative identity, 1, and there are no divisors of zero. The integers themselves have further properties which are relevant to the above questions. Specifically, the integers form a *unique factorization domain* which is to say that every positive integer $n \neq 0, 1$ can be written as the product of prime numbers.

Associated with every integral domain is its *field of fractions*. For the integers this field is just $\mathbf{Q}$, the rational numbers. The field of fractions $\mathcal{Q}$ for a general integral domain $\mathcal{Z}$ is a field which results from the analogous construction and consists of ordered pairs (m, n) where $m, n \in \mathcal{Z}$ with its "arithmetic" defined just as for real fractions. As was the case in Section 3.4, it may be the case that a linear system of equations has no solution in $\mathcal{Z}$ but does have a "rational" solution in its field of fractions $\mathcal{Q}$.

5.1. Problem definitions. Before discussing any algorithms or the algebraic structures within which they make sense, it is necessary to reconsider the definitions of the various linear problems under discussion and to determine when the questions themselves are well-defined.

Since the concept of linear dependence carries over to arbitrary modules, we may define the rank of a matrix A with entries in a general ring $\mathcal{R}$ by the maximum number of linearly independent rows of A. The question of determining the rank of A therefore can be addressed for a completely general ring $\mathcal{R}$. We shall also see that using (a minor modification of the divisionless) Algorithm 11 will solve this problem.

Similarly the determinant can be defined for matrices over a general ring using the "permutation-based" definition for the expansion of a determinant. However, the divisionless GE algorithm does not readily yield the det A even for the integers $\mathbf{Z}$ due to the growth of the elements resulting from the elementary row operations performed. More structure is needed in the ring $\mathcal{R}$ if we are to be able to obtain det A using GE. The fraction-free Algorithm 14 can be performed in any unique factorization domain. The basic GE Algorithm 1 can be modified to solve the problem in a (noncommutative) division ring.

Something similar to division is clearly needed in order that a system of linear equations can be solved. Even when such a system is well-defined, it may be necessary to go to the field of fractions in order to obtain the solution. So for the solution of systems of linear equations, the underlying algebraic ring must have most of the properties of the integers and the familiar number systems.

5.2. Algorithms.

Rank. The division-free GE Algorithm 11 is well-defined for any commutative ring. For a completely general ring $\mathcal{R}$ the only modification needed is that the order of the various products must be consistent. The notion of rank is also well-defined. The following algorithm is a simple modification of Algorithm 7.

Algorithm 17 Division-free GE in a general ring $\mathcal{R}$

```
Input    $n \times n$ matrix $A$ with entries from $\mathcal{R}$
Compute
        for $i = 1$ to $n - 1$
            for $j = i + 1$ to $n$
                for $k = i + 1$ to $n$
                    $a_{jk} := a_{ii}a_{jk} - a_{ik}a_{ji}$
                $a_{ji} := 0$
Output (modified) matrix $A$
```

Remark 22. *The order of the products in the elimination step is chosen to be consistent with the elimination of a_{ji}.*

Remark 23. *In order to obtain the (row) rank of A, pivoting must be included if a pivot element is 0. The pivot row can be interchanged with a row of the active matrix with a nonzero entry in the pivot column. If no such exists then the elimination proceeds to the next column. The rank is given by the number of nonzero elements on the diagonal of the output matrix.*

Determinant. The determinant of a matrix with elements in $\mathcal{R}$ can be defined using permutations and using that definition it is at least feasible [5] to define the idea of an inverse matrix when $\mathcal{R}$ is a commutative ring with a multiplicative identity element 1. (The existence of 1 is needed in order even to define an identity matrix. Commutativity is necessary for the identity $\det AB = (\det A)(\det B)$ to hold.)

However these conditions are not sufficient for the computation of the determinant using GE. The division-free algorithm does not readily yield det A. The general ring equivalent of (13) clearly demands division in order to extract the determinant. The fraction-free Algorithm 14 however is well-defined for any unique factorization domain, UFD, since "division" by a common factor is now available.

Algorithm 18 Computation of $\det A$ by fraction-free GE in a unique factorization domain

```
Input    $n \times n$ matrix $A$ with entries in a UFD $\mathcal{R}$
Compute
        for $i = 1$ to $n - 1$
            factor $a_{ii}$
            for $j = i + 1$ to $n$
                for $k = i + 1$ to $n$
                    $a_{jk} := a_{ii}a_{jk} - a_{ji}a_{ik}$
                $a_{ji} := 0$
            if $i \geq 2$ then          (removal of common factors)
                for $j = i + 1$ to $n$
                for $k = i + 1$ to $n$
                    factor $a_{jk}$
                    replace $a_{jk}$ by product of all its factors except those of $a_{i-1,i-1}$
Output $\det A = a_{nn}$
```

Remark 24. *The factorization of a_{ii}, a_{jk} and the replacement of the latter with a reduced product is the equivalent of division in the integer version, Algorithm 14.*

Remark 25. *Such an algorithm as this could be used for example in the polynomial ring $\mathbf{R}[x]$.*

Remark 26. *Pivoting is needed in the event that any zero-pivot is encountered. This can be done in the usual way by interchanging the pivot row with any suitable active row — and negating the final determinant value.*

For a noncommutative division ring — that is a ring with all the properties of a field *except* that multiplication is not commutative — the original Algorithm 1 can be modified to provide *a* determinant.

Algorithm 19 Computation of $\det A$ by GE in a division ring $\mathcal{R}$

```
Input    $n \times n$ matrix $A$ with entries from $\mathcal{R}$
Compute
        for $i = 1$ to $n - 1$
            for $j = i + 1$ to $n$
                for $k = i + 1$ to $n$
                    $a_{jk} := a_{jk} - (a_{ik}/a_{ii})a_{ji}$
                $a_{ji} := 0$
Output $\det A = \prod_{i=1}^{n} a_{ii}$
```

Remark 27. *Again, of course, some pivoting must be included in the event that any pivot is zero.*

5.3. Solvability for systems of equations. The first question to be addressed is "For what types of ring can we solve such systems?" In [5] we see that the notion of an inverse matrix is definable for any commutative ring $\mathcal{R}$ with 1. However, in such a general setting, the definition of a matrix being *nonsingular* can be confused.

The usual (though not computationally *useful*) cofactor definitions can be used to obtain both $\det A$ and a matrix B satisfying

$$AB = BA = (\det A)\, I \tag{22}$$

and if $\det A$ is a unit in $\mathcal{R}$ then $(\det A)^{-1} B$ is the inverse of a A. (Recall that a *unit* u in $\mathcal{R}$ is an element which has a multiplicative inverse u^{-1} such that $uu^{-1} = u^{-1}u = 1$.) In such circumstances, the system $A\mathbf{x} = \mathbf{b}$ has a solution.

We already see that the notion of nonsingularity is open to many interpretations. In a more general ring still, we can define and compute the rank of a matrix using Algorithm R1 for example which allows nonsingularity of A to be defined by its having full rank. This definition applies even when $\det A$ is not well-defined. However, if $\det A$ is well-defined and is nonzero, then A has full rank and so there is no confusion between these two definitions of nonsingular when they both make sense. From (22) it is apparent that nonsingular and invertible need not be equivalent since $\det A$ could be nonzero but not a unit.

We have already seen that for $\det A$ to be computed by GE, we require that $\mathcal{R}$ is a UFD. In such circumstances (and assuming A is nonsingular) it follows that a system $A\mathbf{x} = \mathbf{b}$ has a solution in the field of fractions $\mathcal{Q}$. This solution would be computed by Algorithm 15 modified to the "arithmetic" of $\mathcal{R}$ and $\mathcal{Q}$ with the operation reduce being interpreted as removal of any common factors using the unique factorization property rather than the gcd. If $\mathcal{R}$ is a Euclidean domain, then (a suitably modified form of) the Euclidean algorithm for obtaining greatest common divisors can be used.

Unique factorization domains. In order to obtain the solution of a system in the field of fractions $\mathcal{Q}$ of a UFD $\mathcal{R}$ we can use the following approach.

Algorithm 20 Solution of a system by GE in the field of fractions $\mathcal{Q}$ of a UFD $\mathcal{R}$

```
Input      $n \times n$ matrix $A$ and an $n$-vector $\mathbf{b}$ with entries in a UFD $\mathcal{R}$
  (Fraction-free forward elimination as in Algorithm UFDdet
    modified for right-hand side)
Compute
        for $i = 1$ to $n-1$
            factor $a_{ii}$
            for $j = i+1$ to $n$
                $b_j := a_{ii}b_j - a_{ji}b_i$
                for $k = i+1$ to $n$
                    $a_{jk} := a_{ii}a_{jk} - a_{ji}a_{ik}$
                $a_{ji} := 0$
            if $i \geq 2$ then          (removal of common factors)
                for $j = i+1$ to $n$
                factor $b_j$
                replace $b_j$ by product of all its factors *except* those of $a_{i-1,i-1}$
                for $k = i+1$ to $n$
                factor $a_{jk}$
                replace $a_{jk}$ by product of all its factors *except* those of $a_{i-1,i-1}$
  (Back substitution as in Algorithm 15 modified for $\mathcal{R}$, $\mathcal{Q}$)
*Initialize rational solution*  $\mathbf{m} := 0$; $\mathbf{n} := 1$    (solution will be $x_i = m_i/n_i$)
*Rationalize right-hand side*  $\mathbf{p} := \mathbf{b}$; $\mathbf{q} := 1$    (right-hand side $b_i = p_i/q_i$)
*Compute*
        for $j = n$ downto 1
            $m_j := p_j$; $n_j := q_j * u_{jj}$
            reduce $[m_j, n_j]$    (factor both arguments, remove all common factors)
            for $i = 1$ to $j - 1$
                $p_i := p_i * n_j - u_{ij} * m_j * q_i$
                $q_i := q_i * n_j$
                reduce $[p_i, q_i]$
*Output* solution $[\mathbf{m}, \mathbf{n}]$
```

Integral domains. Using the division-free Algorithm 11 for the forward elimination would allow the UFD solution to be extended to the case where $\mathcal{R}$ is a general integral domain. Again a solution will exist in the field of fractions $\mathcal{Q}$ provided the matrix is nonsingular. In this situation, just as with divisionless integer arithmetic, the problems of range growth become potentially severe.

Algorithm 21 Division-free solution of a system by GE in a the field of fractions $\mathcal{Q}$ of an integral domain $\mathcal{R}$

```
*Input*     $n \times n$ integer matrix $A$ and right-hand side $\mathbf{b}$
*Compute*   (Division-free forward elimination)
        for $i = 1$ to $n - 1$
            for $j = i + 1$ to $n$
                $b_j := a_{ii}b_j - a_{ji}b_i$
                for $k = i + 1$ to $n$
                    $a_{jk} := a_{ii}a_{jk} - a_{ji}a_{ik}$
                $a_{ji} := 0$
*Initialize rational solution*  $\mathbf{m} := 0$; $\mathbf{n} := 1$    (solution will be $x_i = m_i/n_i$)
*Rationalize right-hand side*  $\mathbf{p} := \mathbf{b}$; $\mathbf{q} := 1$    (right-hand side $b_i = p_i/q_i$)
*Compute*   ("Rational" back substitution)
        for $j = n$ downto 1
            $m_j := p_j$; $n_j := q_j * u_{jj}$
            for $i = 1$ to $j - 1$
                $p_i := p_i * n_j - u_{ij} * m_j * q_i$
                $q_i := q_i * n_j$
*Output* solution $[\mathbf{m}, \mathbf{n}]$
```

Remark 28. *The potential for range growth is shown very clearly by the absence here of any reduction of the fractions generated in the back substitution phase.*

Solution in $\mathcal{R}$. Just as with the integers themselves, it may be that a particular system has a solution in the original ring $\mathcal{R}$ without the need to go to the field of fractions and rational "arithmetic". In the case where $\mathcal{R}$ is a UFD the "integer" solution, if it exists, would be delivered since the various reductions and factorizations would result in all denominators being 1. However, if we know in advance that a particular system has a solution in $\mathcal{R}$, then at least the back substitution phase can be simplified since there is then no need to introduce the fractions at all.

The simplifications to the above algorithms are then obvious and we do not detail them here.

6. THE IMPACT OF PARALLELISM

The recent expansion in parallel computing has had profound effects on many areas of computation - but perhaps nowhere has this impact been so marked as in numerical linear algebra. There are many different parallel architectures including many special purpose processors for tasks such as signal processing. In this section, we restrict attention to two basic types of parallelism: vector processing and array processors.

The former are characterized by the highly optimized vector pipeline processors of the Cray XMP series. These machines also typically featured some "true" parallelism in that several such processors are available to be used simultaneously working on different data. The latter class of (massively parallel) array processors is characterized by machines such as the Connection Machines and the MasPar MP-1 and MP-2 series.

Each individual architecture has its own particular properties but there are some general properties which can usefully be discussed here. For much more detail on the use of vector machines for solution of linear systems see [18]. For further details on parallel algorithms and array processors, [17] provides an excellent introduction with special reference to linear algebra problems. The discussion of basic linear algebra operations in [7] does not pay particular attention to any one architecture but does address architecture-related issues through discussions of saxpy and other special operations and their relations to storage schemes. An introduction to some of these ideas and to performing basic linear algebra operations on an array processor can also be found in [3].

6.1. Vector and pipeline architectures.

By a "vector" architecture, we mean a computer which is designed to perform operations on vector quantities at high speed. This is most frequently achieved through efficient use of a pipeline processor. In such a machine the operation of adding two vectors for example is greatly enhanced by allowing certain subtasks (such as reading form memory, adding, rounding and normalizing, and writing to memory) to be performed simultaneously. From the point of view of programming a vector machine, it is usually valid to think of the operation

```
for $i = 1$ to $n$
  $c_i := a_i + b_i$
```

as a single vector operation

$$\mathbf{c} := \mathbf{a} + \mathbf{b}$$

where each component of $\mathbf{c}$ is computed *at the same time.*

There are some occasions when it is necessary to pay more careful attention to the pipelined nature of such a vector operation but for the most part thinking of it as a vector operation is appropriate. In fact most vector machines are designed to perform a typical saxpy operation such as $\mathbf{v} := a\mathbf{x} + \mathbf{y}$ ("scalar a x plus y") very efficiently.

Note that the innermost loop of the basic ijk form of Gauss elimination, Algorithm 1, is performing a row-saxpy. this implementation of GE would therefore be well-suited to a vector computer in which matrices are stored by rows. Similarly the ikj form, Algorithm 5, also has a saxpy at its heart - but this time a column-saxpy making this version of GE better suited to a vector processor with matrices stored by columns.

To see the real effect of the vector processing here, we consider the column-oriented version. For notational convenience we denote that part of the i-th column of A below the main diagonal by $\mathbf{c}^{(i)}$ so that the k-th component of this column vector is $c_k^{(i)} = a_{i+k,i}$. The part of the right-hand side vector below the i-th position is denoted $\mathbf{b}^{(i)}$. With this notation the vector form of Algorithm 5 is

Algorithm 22 Basic GE: Column vector form

```
Input    $n \times n$ matrix $A$ (and right-hand side $\mathbf{b}$ if solving a system)
Compute
    for $i = 1$ to $n - 1$
        $\mathbf{m}^{(i)} := \mathbf{c}^{(i)}/a_{ii}$
        $\mathbf{a}^{(i)} := \mathbf{0}$
        $\mathbf{b}^{(i)} := \mathbf{b}^{(i)} - b_i\mathbf{m}^{(i)}$        (if solving a system)
        for $k = i + 1$ to $n$
            $\mathbf{c}^{(k)} := \mathbf{c}^{(k)} - a_{ik}\mathbf{m}^{(i)}$
Output (modified) matrix $A$ (and $\mathbf{b}$ if solving a system)
```

The major impact on the complexity of the algorithm is immediately apparent: the replacement of the j-loop by a single vector operation reduces the overall complexity to $O(n^2)$. Computation of the multiplier *vector* is consists of division of a vector by a constant which can be achieved with a single (scalar) reciprocation followed by a saxpy in which one of the vectors is $\mathbf{0}$. Similarly the update of the right-hand side vector is a column-saxpy.

The vector operation counts for this algorithm are summarized in Table 6 along with those for the back substitution phase which is a similarly vectorized version of Algorithm 8.

TABLE 6 Vector floating-point operation counts for column GE solution of an $n \times n$ linear system $A\mathbf{x} = \mathbf{b}$.

Operation	Forward elimination	Back substitution	TOTAL
Scalar /	$n - 1$	n	$(2n - 1)$
saxpy	$\frac{1}{2}(n-1)(n+2)$	$n - 1$	$\frac{1}{2}(n-1)(n+4)$

In the case of a pipeline computer as opposed to a truly parallel vector machine, some of the vector lengths would become too short to justify using the pipeline. Nonetheless we see that the potential saving achieved by even the most elementary parallelism is great. In the case of a "true" vector processor, we have made an implicit assumption that the limit on the vector length is at least n.

We see that adding a "vector dimension" to the processor power has reduced the algorithmic complexity by an order of magnitude. In the case of an array processor with at least n^2 processors we might therefore hope to reduce the Gauss elimination algorithm to $O(n)$ parallel floating-point operations, or *paraflops*.

6.2. Array architectures. In this section, we consider the use of massively parallel array processors for Gauss elimination. Such a computer architecture consists of a (typically square or rectangular) array of processing elements operating in the SIMD paradigm. That is to say that, at any given time, each processor is either performing the *same* instruction *on its own data* or is idle. (SIMD is an acronym for Single Instruction Multiple Data stream.)

The individual processors in such an array are usually *much* less powerful than would be encountered on a serial machine. The power of the parallelism comes from its scale. Even if individual operations are much slower, adding two matrices using a single paraflop with the instruction

$$C := A + B$$

is likely to be significantly faster than the conventional double loop

```
for $i = 1$ to $n$
  for $j = 1$ to $n$
    $c_{ij} := a_{ij} + b_{ij}$
```

with its n^2 flops — especially when n gets large.

Typical parallel arrays have at least $1024 = 32 \times 32$ processors. Arrays of at least 4096 are now more common. On such an array the comparison for matrix addition is therefore between one (slow) paraflop and 4096 (fast) serial floating-point operations. Even if the individual arithmetic operations are slower by a factor of 32, the overall performance would be speeded up by a factor of 128 which is certainly worth achieving. Many linear algebra operations can be readily modified to be performed on an array processor with close to the maximal speed-up. Matrix multiplication, for example can be reduced to just $O(n)$ paraflops on an $n \times n$ array.

For simplicity, we shall suppose throughout this section that the matrices under consideration fit the array — so that both are $n \times n$ arrays. For larger matrices, the sort of algorithms discussed here can be modified to deal with the matrices in pieces. For smaller matrices, the algorithms described can be used but will not necessarily take full advantage of the processor array.

The biggest new cost associated with implementing algorithms such as Gauss elimination or LU factorization on an array processor comes form the inter-processor communications which are needed. This cost must be weighed alongside all the arithmetic and other costs in deciding what architecture should be used for a particular problem. In this section we are concerned only with implementing GE on such a processor.

In keeping with the notation of the previous section, we shall denote by $\mathbf{r}^{(i)}$ the i-th row of the current matrix A. We shall also define two *logical* arrays rows and columns which are used to control which processors perform particular operations. For the i-th stage of GE, these sets will are $rows = \{j : j > i\}$ and $columns = \{k : k \geq i\}$. The "active" part of the matrix would then be identified with the *mask*

```
if (rows) and (columns) ...
```

which has the effect that only those processors for which both $j > i$ and $k \geq i$ would perform the instructions covered by this test.

In Algorithm 23, below, for the case of linear systems we consider the situation of solving a matrix system

$$AX = B$$

where B can have as many as n columns. There is no additional cost associated with this. We shall denote the i-th row of the current matrix B by $\mathbf{b}^{(i)}$. (Note this is different from the notation for Algorithm 22 for a vector processor which was column-oriented.) The notation $\mathbf{m}^{(i)}, \mathbf{c}^{(i)}$ used in the last section is retained.

The other point to be emphasized before detailing the algorithm is that arithmetic operations referred to here are *array* operations *not* matrix operations. Thus the product MA refers to the elementwise product of these two arrays.

Algorithm 23 Basic GE: Array processor form

```
Input    $n \times n$ matrix $A$ (and right-hand side $B$ if solving a system)
Compute
    for $i = 1$ to $n - 1$
        $rows := \{j : j > i\}$
        $columns := \{k : k \geq i\}$
        copy $\mathbf{a}^{(i)}$ to all rows to generate the array $rowa$
        copy $\mathbf{b}^{(i)}$ to all rows to generate the array $rowb$
        $\mathbf{m}^{(i)} := \mathbf{c}^{(i)}/a_{ii}$
        copy $\mathbf{m}^{(i)}$ to all columns to generate the array $M$
        if ($rows$)
            $B := B - M * rowb$    (if solving a system)
            if ($columns$) $A := A - M * rowa$
Output (modified) matrix $A$ (and $B$ if solving a system)
```

Remark 29. *The arithmetic complexity of this algorithm is easy to compute: There are just 5 paraflops used at each stage so that the total count is* $5(n-1)$ *paraflops consisting of* $(n-1)$ *parallel divisions,* $2(n-1)$ *parallel multiplications and* $2(n-1)$ *parallel additions.*

It is also easy to see that Algorithm 23 is wasteful. At stage i, all processors in the first i rows are inactive. However, *at no additional cost*, the elimination could be performed *above* the diagonal simultaneous with the triangular factorization. That is we could perform the Gauss-Jordan algorithm for the same cost as GE. (Recall that for a serial machine, Gauss-Jordan is twice as expensive as GE.) The effect of this is that the back substitution is replaced by solving a diagonal system - a very simple task - especially on a parallel computer. The only change needed is to modify the definition of the row mask to $rows := \{j : j \neq i\}$.

The solution of the resulting diagonal system would be achieved by simply copying the diagonal entries across their respective rows and then using the single parallel division:

$X := B/diag$

This even raises the possibility that there may be circumstances in which computing a matrix inverse may be a practical and desirable computation. Of course, it should be recalled that the Gauss-Jordan algorithm is numerically less stable than Gauss elimination. For ill-conditioned matrices, therefore it may be better not to use this variation - however for such matrices we would probably want to use a more stable algorithm than GE, too!

Pivoting. As in all other cases of GE, it is possible that zero or small pivots will be encountered and that pivoting is needed. In most parallel programming languages there is a built-in reduction algorithm for locating the maximum element in an array. Therefore finding the conventional pivot element for GE is a simple task. The choice between row interchanges and storing a pivot vector is much the same as for serial implementations. The pivot vector could easily be replaced in this context with another mask *used* so that the search for the pivot element would only consider rows which had not yet been used as a pivot row.

6.3. Parallelism for long integer arithmetic. As a final comment on the impact of parallelism on performing Gauss elimination, we return to the question of integer computation and the need for *long* integer arithmetic to accommodate the growth in the dynamic range. In Section 3.7 we considered the requirements of such long integer arithmetic and, in particular, the convolution form of the product of such integers.

The problem was described as finding the product of

$$m = \sum_{i=0}^{K-1} \alpha_i R^i, \qquad n = \sum_{i=0}^{K-1} \beta_i R^i$$

where R is the effective radix resulting form breaking a long wordlength integer into shorter integer pieces. Recall that in (17), the product is given by

$$m * n = \left(\sum_{i=0}^{K-1} \alpha_i R^i\right)\left(\sum_{j=0}^{K-1} \beta_j R^j\right) = \sum_{k=0}^{2K-2}\left(\sum_{i=0}^{K-1} \alpha_i \beta_{k-i}\right) R^i$$

which is a convolution product of the coefficient vectors. Each coefficient product $\alpha_i\beta_{k-i}$ is a base-R^2 digit which we write as $\alpha_i\beta_{k-i} = \gamma_{k,i}R + \delta_{k,i}$ where each $\gamma_{k,i}$ and $\delta_{k,i}$ is a base-R digit. The product (17) can therefore be written as

$$m * n = \sum_{k=0}^{2K-2}\left(\sum_{i=0}^{K-1} \gamma_{k,i}R + \delta_{k,i}\right) R^i = \sum_{k=0}^{2K-1}\left(\sum_{i=0}^{K-1} \gamma_{k-1,i} + \delta_{k,i}\right) R^i$$

where $\gamma_{k,i} = \delta_{k,i} = 0$ for $i > k$.

An array processor could be used to accelerate this algorithm greatly. Firstly, all the coefficient products in the inner sum in (17) can be computed simultaneously. The realignment of the upper and lower halves of these products needed for the inner summation in the last equation is then a simple shift of data to a neighboring processor and these sums could then also be performed simultaneously. The final carries would be the only part that requires serial processing.

These arithmetic-algorithmic considerations are analogous to the design decisions which are used with $R = 2$ in the design of hardware arithmetic units where bit-parallelism is exploited wherever possible.

7. NOISY MATRICES

In this section we address the problem of computing with "noisy" matrices. That is we shall assume that the underlying matrix A is contaminated with a noise matrix E so that the matrix with which we must compute is

$$\widehat{A} = A + E \tag{23}$$

We shall also assume here that the level of the noise can be estimated so that we have bounds on the elements of E such as

$$|e_{ij}| < S \tag{24}$$

or that we know the elements of E are normally distributed random variables with mean 0 and standard deviation σ. In the latter case, it follows that for each i, j

$$|e_{ij}| < 2\sigma \tag{25}$$

with probability about 95%. (See any standard text on Statistics such as [8].) From the similarity in (24) and (25) it appears that the Gaussian noise situation can be well-modeled by the uniform noise model at least in terms of trying to estimate error bounds or tolerances. For the remainder of this section, we use the uniform noise model (24).

Provided that S is not too large, the effect of using $\widehat{A}$ rather than A is covered by the standard error analyses for all the basic linear algebra problems with matrices of full rank. Essentially S becomes a bound on the absolute errors of the original data or E plays the role of δA in the error analysis described in Section 2.5. (See [3] or, for an extensive treatment, [7] for more details of this error analysis.) One situation in which the noisy matrix (23) can be expected to yield substantially different results from A is when A is rank deficient. The determination of the "effective" rank of rank deficient matrices is an important aspect of several naval applications.

7.1. Determination of effective rank. By determination of the effective rank, we mean finding rank(A) even though we are given only $\widehat{A}$ and the bound (24) on the noise matrix. We assume here that all matrices have dimension $n \times n$ and that A has rank less than n. Typically, we would expect $\widehat{A}$ to be of full rank since the noise matrix is unlikely to reflect the same linear dependencies among its rows (or columns) as those of A.

In the no-noise situation, Algorithm 10 delivers the rank of a matrix simply by counting the number of nonzero elements on the diagonal of the upper triangular factor U resulting from LU factorization with partial pivoting. Of course, even roundoff errors mean that testing these diagonal elements against zero is inadequate. For very ill-conditioned matrices, this effect could be severe. But for other matrices, we may expect that these zeros will be replaced with small values and that the true nonzero diagonal entries are substantially larger.

In [12], it is stated that GE (or LU factorization) cannot satisfactorily distinguish between the "true" nonzeros and the "should-be-zeros". However this conclusion appears to be based on an example in which a divisionless version of Gauss elimination is applied to a small example. If Algorithm 10 (even without pivoting) is applied to this example the results are very clear. In their application the diagonal entries are given as $2.998, -12.922, 0.577$ which should be compared with singular values 10.331, 2.644 and 0.0064. However, with or without pivoting, conventional GE yields a diagonal with entries $2.998, 4.315, -0.014$ which shows a much clearer break between the "large" values and the small one. In this particular example, $S = 0.01$ so that the "zero" has been preserved to the same order of magnitude as S.

The question is how well can GE do as a rank determination algorithm in the presence of noise? Clearly one 3×3 example does not justify its general applicability. Some experiments were conducted using MATLAB with randomly generated noisy matrices where the bound S was small relative to the range of values for the elements of A.

Generation of random noisy matrices. In order to generate random noisy matrices with known effective rank r, first a set of r linearly independent vectors with random entries in a specified interval was generated. Next random linear combinations of these row vectors were used to generate the rows of the matrix A. The coefficients of these linear combinations were also from a specified range. Finally a complete $n \times n$ noise matrix with random elements in the interval $[-S, S]$ was added to yield $\hat{A}$.

Of course there is a remote possibility that the underlying matrix A could have rank less than r. This never happened. An example of the MATLAB code used for this is:

```
function A = testmat(n,r);
 a=20*rand(n)-10;
 A=zeros(n);
 for i=1:n
   for j=1:r
     A(i,:)=A(i,:)+(6*rand-3)*a(j,:);
   end
end

function E=noisemat(n,s);
E=s*(2*rand(n)-1);

Ahat=testmat(n,r)+noisemat(n,s);
```

The tolerance level. With partial pivoting, all multipliers used in the LU factorization are necessarily less than 1. Assuming that the noise is significantly greater than the roundoff error effect but significantly smaller than the range of the matrix entries , it follows that its effect on the LU factors is bounded by nS and so this could be used as a likely tolerance for determining the effective rank of the matrix.

The MATLAB experiments that were performed suggest that this is indeed a satisfactory tolerance to use *most of the time.* Some experiments were carried out using the MATLAB LU factorization routine. The initial tests were based on perturbations of a 7×7 matrix which was used as the basis of the analysis in [12]. This matrix is

$$A = \begin{bmatrix} 3 & 2 & 1 & 7 & 4 & 5 & 3 \\ 1 & 4 & 2 & 6 & 5 & 10 & 3 \\ 8 & 1 & 5 & 13 & 5 & 7 & 0 \\ 4 & 2 & 7 & 15 & 11 & 11 & 4 \\ 1 & 2 & 1 & 3 & 2 & 5 & 1 \\ 2 & 1 & 3 & 5 & 3 & 5 & 0 \\ 3 & 10 & 1 & 5 & 2 & 21 & 1 \end{bmatrix} \qquad (26)$$

which has true rank 4.

The MATLAB function randn was used to generate noise matrices whose elements were normally distributed and with specified standard deviation. The first phase of the experiment was to obtain the effective rank of many such perturbations by counting the number of diagonal entries in U which exceeded some threshold absolute value. Two hundred perturbations were used in each test. The value of the standard deviation and the tolerance used were varied between runs.

It quickly became apparent that choosing a tolerance close to $n\sigma$ provided the best results. The following table of results for $\sigma = 0.3$ illustrate this point.

TABLE 7 Rank tests using LU factorization. 7×7 matrix (26). $\sigma = 0.3$

Shows number of times each rank was obtained with 200 perturbations for each tolerance

Tolerance	Rank=3	Rank=4	Rank=5	Rank=6	Rank=7
σ	0	0	10	72	118
1	0	60	97	39	4
5σ	0	155	43	2	0
10σ	50	150	0	0	0

When the test was repeated for $\sigma = 0.1$, 1000 perturbations using tolerance 10σ yielded rank 4 996 times and rank 5 just 4 times. With a smaller value of σ the results were even more successful. For both these small noise levels, using $tol = 5\sigma$ gave approximately 75% success. These experiments suggest that GE with the appropriate tolerance could be the basis of a successful rank determination algorithm when the noise level is known. It is also already apparent that finding the right tolerance to use is of critical importance and the results may be highly sensitive to this choice.

One way of reducing this effect is to use the LU factorization as a basis for a probabilistic approach to the problem. To this end a second set of tests was run in which the original matrix A is perturbed. The LU rank test used in the previous experiments is then applied to a number of perturbations of this matrix. The effective rank would then be determined by the most frequent "rank" for these and some estimate of the confidence we should have in the result is provided by the complete array of results. Thus a single test would produce results similar to a row of Table 7.

In the experiments 200 perturbations were again used. Clearly this would be too expensive since for even a poorly conditioned example we would expect to be able to complete the singular value decomposition of $\widehat{A}$ with *much* less effort than 200 LU factorizations. The reason for using so many test runs here is to try to eliminate falsely positive results based on small samples.

The tests were performed using three different noise levels $\sigma = 10^{-6}, 0.1, 0.3$ each with two tolerance levels 7σ and 10σ. The combined results of four runs of each case are summarized in Table 8. Each run uses a different underlying perturbation of the basic 7×7 matrix.

TABLE 8 Rank tests using LU factorization. Perturbations of the 7×7 matrix (26).

Table shows percentage for each rank was obtained in each case and the resulting estimated rank.

Std Deviation	Tolerance	Rank=3	Rank=4	Rank=5	Rank=6	Est. Rank
10^{-6}	7σ	0	71.0	28.0	1.0	4
0.1	7σ	0	76.25	23.38	0.38	4
0.3	7σ	2.88	75.25	20.5	1.38	4
10^{-6}	10σ	0	91.5	8.5	0	4
0.1	10σ	0	92.63	7.38	0	4
0.3	10σ	5.5	88.13	6.38	0	4

This evidence suggests that for a matrix of this dimension a tolerance of 10σ is likely to prove very reliable in predicting the effective rank of a matrix from the LU factorization of even a small number of perturbations of the original matrix. As would be expected the performance appears to be beginning to deteriorate as the noise level rises.

The biggest drawback to using GE in this context is that setting the appropriate tolerance is difficult and especially as the dimension of the matrices increases the cost of performing several LU factorizations of perturbed copies may be too expensive. Other approaches to this problem may be more successful for higher dimensions.

Limited further experiments were conducted with matrices of higher dimension and without the probabilistic element. Specifically within the context of low rank matrices of larger dimension, the instability of LU factorization becomes apparent. On experiments using randomly generated 20×20 matrices with rank 4, there were examples in which the effective rank was correctly predicted by the LU factorization. These typically had moderate condition numbers.

One particular example generated using the functions described above with $n = 20, r = 4, s = 0.5$ had condition number reported by MATLAB as 2.4×10^6. The LU factorization using a tolerance of ns yielded an effective rank of just 2. (The SVD gave the correct result as did a least squares based algorithm.) Studying the diagonal entries of U it was apparent that there was no readily recognizable alternative tolerance was available. The two "large" entries were approximately -39 and 31, the next four were 7.0, 6.8, 5.7 and 4.5 followed by three between 2 and 2.5 and all except one were greater than 0.5.

Trying repeated perturbations of the matrix as in the second phase of the above experiments did not provide any clear conclusion either. After 28 such perturbations had been tested, the results indicated rank=4 17 times, rank=3 10 times and rank=2 once. With a mean rank estimate of 3.57 this is clearly not conclusive. Note too that 28 LU factorizations of different 20×20 matrices required approximately 140,000 flops which does not compare favorably with the 31,000 required for the SVD for this same matrix.

Gauss elimination may be the answer to many linear algebra problems but effective rank determination for low rank matrices of even moderately large dimension in the presence of noise is probably best approached by alternative techniques.

7.2. Other approaches. The most reliable and commonly used technique for this problem is based on the SVD. This algorithm too is sensitive to the tolerance level and that is the main point of [12]. However in general the singular value decomposition is usually expensive to compute.

The work of [6] is based on an alternative which makes use of the coefficients of the characteristic polynomial of the matrix. This is a statistical approach which relies on a finite algebraic algorithm - but one which is also likely to be expensive computationally since it is most naturally suited to working from full rank downwards until the effective rank is revealed. The problem here is that the full rank determination requires the computation of the determinant and so is already as expensive as the LU factorization. There may be alternative arrangements of the algorithm which would overcome this difficulty.

A further alternative currently being investigated uses least squares approximation [24]. The idea here is to have an algorithm which works upwards from rank 1 to determine the rank of a matrix which is known to be of low rank more quickly. The basic idea is to approximate each row of the matrix, in a least squares sense, with a multiple of the largest row. If the residual matrix is small enough then the effective rank is 1. Otherwise the process is repeated with the remaining rows. The early results with this approach are encouraging and the investigation is continuing. For the specific example quoted above the correct rank was determined using just 8000 flops representing a saving of nearly 75% compared with the SVD-based algorithm.

REFERENCES

[1] H.Anton, *Linear Algebra* 4th Ed, Wiley, New York, 1984

[2] N.R.Blachman and M.J.Mossinghoff, *Maple V Quick Reference*, Brooks/ Cole, Pacific Grove, CA, 1994

[3] J.L.Buchanan and P.R.Turner, *Numerical Methods and Analysis*, McGraw-Hill, New York, 1992

[4] J.W.Demmel, *Trading off parallelism and numerical stability*, Linear Algebra for Large-scale and Real-time Applications (G.Golub, M.Moonen & B. de Moor, eds) Kluwer, 1993, pp 49-68

[5] D.S.Dummit and R.M.Foote, *Abstract Algebra*, Prentice-Hall, 1991

[6] R.Gleeson, *Using the coefficients of the characteristic polynomial for effective rank determination*, NAWC-AD Tech Rep. 1996

[7] G,H.Golub and C.F.van Loan, *Matrix Computations* 2nd Ed, Johns Hopkins Press, Baltimore, 1989

[8] R.V.Hogg and A.T.Craig, *Introduction to Mathematical Statistics*, 5th Ed, Prentice-Hall, 1995

[9] B.J.Kirsch, *High throughput signal processor for C3 applications*, NAWCADWAR-95021-4.5, 1995

[10] B.J.Kirsch and P.R.Turner, *Modified Gaussian elimination for adaptive beamforming using complex RNS arithmetic*, NAWCADWAR 94112-50, 1994

[11] B.J.Kirsch and P.R.Turner, *Adaptive beamforming using RNS arithmetic*, Proc ARITH11, IEEE Computer Society, Washington, DC, 1993, pp36-43

[12] K.Konstantinides and Kung Yao, *Statistical analysis of effective singular values in matrix rank determination*, IEEE Trans ASSP 36 (1988) 757-763

[13] P.Kornerup and D.W.Matula, *Finite precision lexicographic continued fraction number systems*, Proc ARITH7, IEEE Computer Society, Washington DC, 1985, pp 207-214

[14] P.Kornerup and D.W.Matula, *An on-line arithmetic for bit-pipelined rational arithmetic*, J Parallel and Dist Comp 5 (1989) 310-330

[15] P.Kornerup and D.W.Matula, *Exploiting redundancy in bit-pipelined rational arithmetic*, Proc ARITH9, IEEE Computer Society, Washington DC, 1989, pp 119-126

[16] The MathWorks, Inc., *The Student Edition of MATLAB Version 4, User's guide*, Prentice-Hall, Englewood Cliffs, NJ. 1995

[17] J.J.Modi, *Parallel Algorithms and Matrix Computations*, Oxford University Press, Oxford, 1988

[18] J.M.Ortega, *Introduction to Parallel and Vector Solution of Linear Systems*, Plenum, New York, 1988

[19] N.R.Scott, *Computer Number Systems and Arithmetic*, Prentice-Hall, 1985

[20] P.H.Sterbenz, *Floating-point computation*, Prentice-Hall, 1974

[21] P.R.Turner, *An Improved RNS division Algorithm*, Report NAWCADWAR-95002-4.5,1995

[22] P.R.Turner and B.J.Kirsch, *An analysis of Gauss elimination for adaptive beamforming*, Report NAWCADWAR - 95003-4.5, 1995

[23] P.R.Turner and B.J.Kirsch, *Operation complexity for integer or RNS Gaussian elimination*, Report NAWCADWAR - 95004-4.5, 1995

[24] P.R.Turner, *Low rank determination using least squares*, NAWC-AD Tech Rep 1996

[25] J.H.Wilkinson, *Rounding Errors in Algebraic Processes*, HMSO, London, 1963

[26] J.H.Wilkinson, *The Algebraic Eigenvalue Problem*, Oxford University Press, Oxford, 1965

[27] S.Wolfram, *Mathematica* 2nd Ed. Addison-Wesley, New York, 1991

Distribution List

Report No. NAWCADPAX- -96 - 194 - TR

	No. of Copies
Office of Naval Research	2
800 N. Quincy St.	
Arlington, VA 22217	
Marine Corps Research Center	
2040 Broadway Street	
Quantico, VA 22134-5107	
Marine Corps University Libraries	2
Naval Air Systems Command	
Air-5002	
Washington, DC 20641-5004	
Technical Information & Reference Center	2
Naval Air Warfare Center, Aircraft Division	
Building 407	
Patuxent River, MD 20670-5407	
Naval Air Station Central Library	2
Naval Air Systems Command (NAVAIR)	
Jefferson Plaza Bldg 1., 1421 Jefferson Davis Hwy	
Arlington, VA 2243-5120 .	
Director Science & Technology (4.0T)	2
Naval Sea Systems Command	
2531 Jefferson Davis Hwy	
Arlington, VA 22242-5100	
Technical Library, (SEA04TD2L)	2
Defense Technical Information Center	
Cameron Station BG5	
Alexandria, VA 22304-6145	
DTIC-FDAB	2
U.S. Naval Academy	
Annapolis, MD 21402-5029	
Peter R. Turner (Mathematics Department)	10
Dr. Richard Werking (Nimitz Library)	2
Naval Air Warfare Center	
Weapons Division	
China Lake, CA 93555-6001	
Head Research & Tech. Div. (NAWCWPNS-474000D)	2
Computational Sciences (NAWCWPNS-474400D)	2
Mary-Deirdre Coraggio (Library Division, C643)	2

Distribution List, cont.

Report No. NAWCADPAX- -96- 194 - TR

	No. of Copies
Naval Postgraduate School	
Monterey, CA 93943-5002	
Dudley Knox Library	2
Naval Research Laboratory(NRL)	
4555 Overlook Ave, SW	
Washington, DC 20375-5000	
Center for Computational Science (NRL-5590)	2
Superint., Lab. for Comput. Phy & Fluid Dynamics (NRL-6400)	2
Ruth H. Hooker Research Library (5220)	2
Naval Command, Control & Ocean Surveillance Center	
200 Catalina Blvd	
San Diego, CA 92147-5042	
Technical Library (NRAD-0274)	2
Signals Warfare Div (NRAD-77)	2
Analysis & Simulation Div. (NRAD-78)	2
Director of Navigation & Air C3 Dept. (NCCOSC-30)	2
Naval Air Warfare Center	
Aircraft Division Warminster	
Warminster, PA 18974-0591	
Warfare Planning Systems (4.5.2.1.00R07)	2
Tactical Inf. Systems (4.5.2.2.00R07)	2
Mission Comp. Processors (4.5.5.1.00R07)	2
Dr. Robert M. Williams (4.5.5.1.00R07)	20
Acoustic Sensors (4.5.5.4.00R07)	2
RF Sensors (4.5.5.5.00R07)	2
EO Sensors (4.5.5.6.00R07)	2
Inductive Analysis Branch (4.10.2.00R86)	2
TACAIR Analysis Division (4.10.1.00R86)	2
Operations Research Analysis Branch (4.10.1.00R86)	2
Advanced Concepts Branch (4.10.3.00R86)	2
Nav. Aval. Sys. Dev. Division (3.1.0.9)	2
Anthony Passamante (4.5.5.3.4.00R07)	2
Elect. Systems BR (4.8.2.2.00R08)	2
Dr. Richard Llorens (4.3.2.1.00R08)	2
Advanced Processors (4.5.5.1.00R07)	2
Mission & Sensors Integrations (4.5.5.3.000R07)	2
Applied Signal Process BR (4.5.5.3.4.00R07)	2