How to Eliminate Pivoting from Gaussian Elimination — by Randomizing Instead

D. Stott Parker* Dinh Lê
stott@cs.ucla.edu dinh@cs.ucla.edu
Computer Science Department
University of California
Los Angeles, CA 90024-1596

July 23, 1995

Abstract

Gaussian elimination is probably the best known and most widely used method for solving linear systems, computing determinants, and finding matrix decompositions. While the basic elimination procedure is simple to state and implement, it becomes more complicated with the addition of a *pivoting* procedure, which handles degenerate matrices having zero elements on the diagonal. Pivoting can significantly complicate the algorithm, increase data movement, and reduce speed, particularly on high-performance computers.

In this paper we propose an alternative scheme for performing Gaussian elimination that first preconditions the input matrix by multiplying it with random matrices, whose inverses can be applied subsequently. At the expense of these multiplications, and making the linear system dense if it was not already, this approach makes the system 'nondegenerate' — subsystems have full rank — with probability 1. This preconditioning has the effect of (almost certainly) eliminating the need for pivoting.

The randomization approach also suggests many interesting developments beyond the elimination of pivoting. Generally, randomization is a new technique for developing matrix computations that are suitable for high-performance computers.

1 Motivation

Today high-performance computers are used heavily for solving large linear systems, and they run variants of Gaussian elimination and LU decomposition constantly. These computers favor algorithms with little data movement, tight loops that run in cache, simple computations that can be performed with independent parallelism over a large set of values, and things like sums that can be formed in pipeline or with fan-in.

Unfortunately naked Gaussian elimination is unstable. This instability is avoided by *pivoting*, which dynamically permutes rows and columns of the input matrix so that large (nonzero) matrix elements are moved to the diagonal for use in elimination. Pivoting is a necessity in practice. Although partial pivoting is unstable in theory, fifty years' worth of experience has suggested that it is usually stable in practice [7] (though very recently this has come into question again [10, 23, 30]). However, even partial pivoting interrupts the algorithm's data flow and introduces data movement that can slow high-performance computers.

*This research partially supported by NSF grant IRI-8917907.

Pivoting introduces complexity into the otherwise straightforward nested-loop Gaussian elimination computation. The problem of managing this complexity without loss of performance has resulted in hundreds of different implementations for high-performance machines. Dongarra et al. [3] survey a variety of techniques for squeezing additional performance out of Gaussian elimination on vector and parallel architectures. Robert [21] reviews many variations of the Gaussian elimination algorithm that have evolved to exploit features of particular parallel machine architectures.

The communication overhead of pivoting can be significant on a parallel machine. A single pivot can take $O(n)$ steps on machines with limited interconnection schemes; in such a machine, the $O(n^2)$ data movements potentially required by pivoting and the synchronization they require altogether can actually take longer than the $O(n^3)$ arithmetic operations required by Gaussian elimination. Significant overhead is also generated on both MIMD and SIMD machines. A number of authors have considered scheduling algorithms for Gaussian elimination on MIMD machines [21]. For example, Bampis et al. [1] point out that MIMD overhead (communication and idle time) can run to $O(n^3)$, even when pivoting is ignored; Veldhorst [26] has considered the effect of partial pivoting. In [20], implementations of Gaussian elimination with pivoting on SIMD hypercubes are detailed, and experiments show how the communications overhead grows as the number of processors increases.

Another reason for wanting to dispense with the need pivoting is that it appears to obstruct general systolic architectures for Gaussian elimination. This is discussed by Robert [21].

There is a final very significant reason why pivoting is undesirable. Pivoting strongly encourages an iterative, column-at-a-time style; it obstructs a modern recursive, block-at-a-time style and real parallel computation. Real improvements in parallel software development will result if we can avoid bottlenecks like pivoting.

This paper shows that *there is an alternative to pivoting.* The primary role of pivoting is to avoid degeneracies of the input matrix. Pivoting can therefore be replaced by any transformation of the matrix that avoids nondegeneracy.

2 Gaussian Elimination and Pivoting

For completeness we quickly review Gaussian elimination here. It is one of the oldest and best-known numerical algorithms, having been used not only by Gauss, but also by Chinese mathematicians dating to the second century B.C. [22, p.37ff]. It is an extremely popular process for solving linear systems, finding LU and similar factorizations, and computing determinants. It has found a key position in the kernel of many numerical methods (although some have questioned this [23]). An excellent detailed presentation of Wilkinson's work on Gaussian elimination [27, 28], a cornerstone of modern numerical analysis, appears in Section 21 of [6]. A more complete overview, with comprehensive references up to the late-1980's, can be found in Chapter 3 of [7].

2.1 Naked Gaussian Elimination

In its basic form, Gaussian elimination is a sequence of transformations to an $n \times n$ square matrix $A = (a_{ij})$, reducing it to upper-diagonal form in n steps. It can be defined equationally, with the initial assignment $a_{ij}^{(1)} = a_{ij}$ and the recursive definition of $a_{ij}^{(k+1)}$ for $1 \leq k \leq n-1$:

$$a_{ij}^{(k+1)} = \begin{cases} 0 & i \geq k+1,\ j = k \\ a_{ij}^{(k)} - a_{ik}^{(k)} / a_{kk}^{(k)} \cdot a_{kj}^{(k)} & i \geq k+1,\ j \geq k+1 \\ a_{ij}^{(k)} & \text{otherwise.} \end{cases}$$

We will find an exact solution to these equations shortly.

Often this definition is viewed as summarizing a program – an implementation of Gaussian elimination. However: *in this paper, the definition is viewed as a set of equations, without 'roundoff error'*; we will discuss roundoff error later. Gaussian elimination is typically implemented with **for**–loops in a program requiring $n(n^2-1)/3$ assignments altogether:

$$
\begin{array}{ll}
\textbf{for } k = 1 \textbf{ to } n-1 \textbf{ do} & \\
\quad \textbf{for } i = k+1 \textbf{ to } n \textbf{ do} & \\
\quad\quad \textbf{begin} & \\
\quad\quad\quad \ell_{ik} \ := \ a_{ik} \ / \ a_{kk}; & \{ \ a_{ik}^{(k)} \ / \ a_{kk}^{(k)} \ \} \\
\quad\quad\quad \textbf{for } j = k+1 \textbf{ to } n \textbf{ do} & \\
\quad\quad\quad\quad a_{ij} \ := \ a_{ij} \ - \ \ell_{ik} \ \cdot \ a_{kj} & \{ \ a_{ij}^{(k)} \ - \ a_{ik}^{(k)} \,/\, a_{kk}^{(k)} \ \cdot \ a_{kj}^{(k)} \ \} \\
\quad\quad \textbf{end} &
\end{array}
$$

Starting with $A^{(1)} = A$, each iteration of the outer loop computes $A^{(k+1)} = (a_{ij}^{(k+1)})$ for $1 \leq k \leq n-1$, except that as written above the program does not zero the elements of A below the diagonal in the first k columns. (These elements are in precisely the same positions as the multipliers ℓ_{ik}, so many implementations store ℓ_{ik} in a_{ik}.) With the program we can form the triangular matrices

$$
L \ = \ \begin{pmatrix} 1 & & & & \\ \ell_{21} & 1 & & & \\ \ell_{31} & \ell_{32} & 1 & & \\ \vdots & \vdots & \vdots & \ddots & \\ \ell_{n1} & \ell_{n2} & \ell_{n3} & \cdots & 1 \end{pmatrix}, \qquad U \ = \ \begin{pmatrix} a_{11}^{(n)} & a_{12}^{(n)} & a_{13}^{(n)} & \cdots & a_{1n}^{(n)} \\ & a_{22}^{(n)} & a_{23}^{(n)} & \cdots & a_{2n}^{(n)} \\ & & a_{33}^{(n)} & \cdots & a_{3n}^{(n)} \\ & & & \ddots & \vdots \\ & & & & a_{nn}^{(n)} \end{pmatrix}
$$

such that $A^{(n)} = U$ and $A = LU$ (again, ignoring roundoff for now). The computation reduces a general linear problem of the form $A\mathbf{x} = \mathbf{b}$ to a pair of back-substitution problems $L\mathbf{y} = \mathbf{b}$, $U\mathbf{x} = \mathbf{y}$. For a particularly good introduction see [6].

Two problems can arise in implementing this computation on modern computers:

1. One or more of the diagonal elements $a_{kk}^{(k)}$ can be zero. When the matrix A is of rank less than n, this is inevitable: the matrix is singular. However, A can be of full rank yet degenerate, in the sense that zeroes appear on the diagonal. For example, the program above fails for the nonsingular matrix $A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$.

2. The computed values $\widehat{L}$ and $\widehat{U}$ of L and U can be inaccurate because of roundoff errors. By careful tallying of these errors, we can show that $A = \widehat{L}\widehat{U} + E$, where E is a matrix of error values. More precisely $A = \widehat{L}\widehat{U} + \sum_{k=1}^{n-1} E^{(k)}$, where $E^{(k)} = (\epsilon_{ij}^{(k)})$ such that

$$
\epsilon_{ij}^{(k)} \ = \ \begin{cases} \widehat{a}_{ij}^{(k)}\, \delta_{ij} & i \geq k+1, \ j = k \\ -\widehat{\ell}_{ik}\, \widehat{a}_{kj}^{(k)}\, \delta_{ij} \ - \ \widehat{a}_{ij}^{(k+1)}\, \delta'_{ij} & i \geq k+1, \ j \geq k+1 \\ 0 & \text{otherwise} \end{cases}
$$

   the δ values being the roundoff errors introduced by floating point operations [6, p.101]. These errors involve the computed values $\widehat{\ell}_{ik}$ and $\widehat{a}_{ij}^{(k)}$ of ℓ_{ik} and $a_{ij}^{(k)}$, which can be difficult to bound, and when the matrix A is ill-conditioned, or nearly singular, they can get quite large relative to the values in A.

Both of these problems are mitigated by pivoting.

2.2 Gaussian Elimination with Pivoting

Complete pivoting extends Gaussian elimination by permuting rows and columns in order that nonzero elements of A appear on the diagonal. With **complete pivoting**, the largest element in the submatrix is permuted to the current diagonal position. The program above can be modified to implement this as follows:

```
for k := 1 to n − 1 do
    begin
    { find max value in the square subarray a_ij, i, j ≥ k }
        maxvalue := 0 ; r := 0 ; c := 0 ;
        for i := k to n do
            for j := k to n do
                if |a_ij| > maxvalue then
                  begin maxvalue := |a_ij| ; r := i ; c := j end ;
        if maxvalue = 0 then HALT ;
    { swap the k-th and r-th rows }
        for j := k to n do
            begin t := a_kj ; a_kj := a_rj ; a_rj := t end ;
    { swap the k-th and c-th columns }
        for i := 1 to n do
            begin t := a_ik ; a_ik := a_ic ; a_ic := t end ;
    { finally perform Gaussian elimination for the k-th column }
        for i := k + 1 to n do
            begin
                ℓ_ik := a_ik / a_kk ;
                for j := k + 1 to n do
                    a_ij := a_ij − ℓ_ik · a_kj
            end
    end
```

The beauty of complete pivoting is that it both avoids the problem of zero diagonal elements and also permits bounds to be derived. For example, since the maximum element is always swapped into a_{kk}, it is always the case that $|\ell_{ik}| \leq 1$. It leads to the following theorem of Wilkinson [27, 28]:

Theorem 1 *Suppose A is an $n \times n$ nonsingular matrix. and t-digit, base-β floating point arithmetic is used, where $\beta^{1-t} \leq 1/n$. Then the matrices $\widehat{L}$ and $\widehat{U}$ computed by Gaussian elimination with pivoting in floating point with unit roundoff $\mathbf{u}$ satisfy $\widehat{L}\widehat{U} = A + E$, where E is a matrix of roundoff errors such that*[1]

$$\| E \|_\infty \quad \leq \quad n^2 \max_{i,j,k} |\widehat{a}_{ij}^{(k)}| \quad \mathbf{u}.$$

If Gaussian elimination is used to solve $A\mathbf{x} = \mathbf{b}$, the computed solution $\widehat{\mathbf{x}}$ will satisfy $(A + \delta A)\widehat{\mathbf{x}} = b$ where δA is an $n \times n$ matrix of error values such that

$$\| \delta A \|_\infty \quad \leq \quad p(n) \max_{i,j,k} |\widehat{a}_{ij}^{(k)}| \quad \mathbf{u}$$

where $p(n) = O(n^3)$ is a polynomial. Furthermore, when complete pivoting is used,

$$|\widehat{a}_{ij}^{(k)}| \quad \leq \quad k^{1/2} \left(2^1\, 3^{1/2}\, 4^{1/3} \cdots k^{1/(k-1)}\right)^{1/2} \max_{i,j} |a_{ij}| \quad = \quad O(\, k^{1/2}\, k^{(1/4)\ln k}\,) \max_{i,j} |a_{ij}|.$$

[1]Here the matrix infinity-norm is defined by $\| A \|_\infty = \max_{i=1}^n \sum_{j=1}^n |a_{ij}|$. The unit roundoff $\mathbf{u}$ on a machine with t-digit, base-β floating-point arithmetic is $\frac{1}{2}\beta^{1-t}$ when floating point operations are rounded, and β^{1-t} when they are truncated. The bound implicitly assumes $\beta^{1-t} \leq 1/n$.

Often numerical analysts argue that complete pivoting is pessimistic, and it also takes a considerable amount of time ($O(n^3)$) in searching for *maxvalue*. An approximation of this algorithm called **partial pivoting** is therefore justifiable. Partial pivoting is simply the restriction of the program above that searches only the current column for a *maxvalue*:

```
for k := 1 to n − 1 do
    begin
    { find max value in the column subarray a_ik, i ≥ k }
        maxvalue := 0 ; r := 0 ;
        for i := k to n do
            if |a_ik| > maxvalue then
              begin maxvalue := |a_ik| ; r := i end ;
        if maxvalue = 0 then HALT ;
    { swap the k-th and r-th rows }
        for j := k to n do
            begin t := a_kj ; a_kj := a_rj ; a_rj := t end ;
    { finally perform Gaussian elimination for the k-th column }
        for i := k + 1 to n do
            begin
                ℓ_ik := a_ik / a_kk ;
                for j := k + 1 to n do
                    a_ij := a_ij − ℓ_ik · a_kj
            end
    end
```

This is essentially identical to the implementation of the workhorse Gaussian elimination routine `dgefa` in LINPACK, and the implementation in standard toolkits like [19]. Although the overwhelming opinion is that partial pivoting is adequate in practice, the penalty paid by using partial pivoting is that the error bounds are considerably weaker. Wilkinson points out that the bound $|\widehat{a}_{ij}^{(k)}| \leq 2^{k-1} \max_{i,j} |a_{ij}|$ holds, and is actually attained by certain matrices. This bound is a very poor guarantee of accuracy for moderate or large n. Recently several naturally-arising problems have been identified for which partial pivoting does give poor numerical results [10, 30].

2.3 Block Gaussian Elimination

Many matrix operations can be expressed in block form, in terms of submatrices [5, 7]. For example, the following block-Gaussian elimination step can be used recursively on square submatrices of a nonsingular square matrix, assuming that the first block A is nonsingular:

$$\begin{pmatrix} I & 0 \\ -CA^{-1} & I \end{pmatrix} \begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} A & B \\ 0 & D - CA^{-1}B \end{pmatrix}.$$

With this step one can derive the block LDU-decomposition

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} I & 0 \\ CA^{-1} & I \end{pmatrix} \begin{pmatrix} A & 0 \\ 0 & D - CA^{-1}B \end{pmatrix} \begin{pmatrix} I & A^{-1}B \\ 0 & I \end{pmatrix}.$$

The determinant can also be computed recursively as

$$det \begin{pmatrix} A & B \\ C & D \end{pmatrix} = det\ A \cdot det\ (D - CA^{-1}B).$$

Wise [29] emphasizes the case where A, B, C, D are **quadrants** of the matrix (blocks of equal size). This yields elegant recursive algorithms that can be run effectively on some parallel computers.

While recursive specification of matrix operations using this block style is elegant, it is usually viewed as impractical. The block-Gaussian elimination step above, for example, works only when the submatrix A is nonsingular, the upper left blocks of both A and $D - CA^{-1}B$ are nonsingular, the upper left blocks of both of these upper left blocks are nonsingular, etc. Although this recursive condition is known to hold for some important classes of input matrices, such as symmetric positive definite matrices, it certainly will not hold for all. This is a key reason why recursive decomposition is not used more widely. For a detailed discussion see [18].

Several approaches have been proposed for addressing the requirement that submatrices be non-degenerate (recursively nonsingular). First, notice that only one diagonal block need be inverted. Second, Bunch and Hopcroft [2] point out that nonsingular input matrices can always be permuted so that the nonsingularity requirement needed for Gaussian elimination is met. Bunch and Hopcroft also mention[2] that any nonsingular matrix A can be made recursively block-nonsingular by premultiplying with its Hermitian adjoint (conjugate transpose) A^*. The resulting matrix A^*A is Hermitian and positive definite. With this approach, for example, we can compute the inverse A^{-1} by computing a generalized inverse of A:

$$A^+ \quad = \quad (A^*A)^{-1} \; A^*.$$

When A is nonsingular, $A^+ = A^{-1}$. Unfortunately this approach is relatively expensive, since it requires multiple matrix multiplications. Furthermore, it is less stable than Gaussian elimination with pivoting, since the condition number of A^*A is always larger than that of A.[3]

3 Randomization as an Alternative to Pivoting

The purpose of this paper is to show how pivoting can be escaped with randomization, and how block matrix algorithms can be made more feasible. Let us sketch the idea of a randomizing linear transformation, and how it can be used.

3.1 The Basic Idea

Let A be a nonsingular $n \times n$ matrix, and let U and V be random $n \times n$ matrices.

Our basic idea is that, given the randomness of U and V, *we can perform Gaussian elimination on the product matrix UAV without any pivoting.* Intuitively UAV is sufficiently 'random' that, with probability 1, pivoting is never needed.

This idea is justified formally in [16]. Let us take it on faith for the moment, and see how it gives something interesting.

Specifically, suppose A is a nonsingular real $n \times n$ matrix, let U and V be nonsingular random $n \times n$ matrices. (By a random matrix, we mean a matrix whose elements are chosen independently from a suitable distribution. We can guarantee such matrices to be nonsingular by choosing them from a nonsingular class of matrices, like orthogonal matrices. More on this shortly.) Then:

- To solve $A\mathbf{x} = \mathbf{b}$, we can instead solve $(UAV)\mathbf{y} = U\mathbf{b}$. Afterwards, it follows that $\mathbf{x} = V\mathbf{y}$.
- To compute A^{-1}, we can instead compute $(UAV)^{-1}$, since U and V are both guaranteed to be nonsingular. Afterwards, it follows that $A^{-1} = V\,(UAV)^{-1}\,U$.

[2] Bunch and Hopcroft attribute the idea to Schönhage, but it is older. For example, it appears in [13, p.1056].

[3] In fact $\| A^*A \|_2 = \| A \|_2^2$, where $\| \cdot \|_2$ is the spectral matrix norm, so with this norm the condition number of A^*A is the square of that for A. See e.g. [12, p.271].

- To compute $det\,A$, we can instead compute $det\,(U A V)/(det\,U\;det\,V)$, assuming the determinants of U and V can be computed easily.

Each of the computations here is easily implemented if Gaussian elimination without pivoting successfully factors the preconditioned matrix UAV.

Putting it another way, Gaussian elimination needs pivoting for **degenerate matrices**: matrices that yield zeroes on the diagonal. In the space of matrices, degenerate matrices have measure zero: almost every matrix is nondegenerate. So, we can transform any problem involving a possibly degenerate matrix A into a problem involving a (randomized, almost certainly) nondegenerate matrix UAV, and then transform back (derandomize) the resulting solution.

3.2 Example: Using Randomization instead of Pivoting

To illustrate the basic idea, consider the 2×2 matrix

$$A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

If we arbitrarily select the random matrices

$$U = \begin{pmatrix} +.6483 & -.7614 \\ +.7614 & +.6483 \end{pmatrix}, \qquad V = \begin{pmatrix} +.7279 & -.6857 \\ +.6857 & +.7279 \end{pmatrix}$$

we get the random transform

$$UAV \;=\; \begin{pmatrix} +.6483 & -.7614 \\ +.7614 & +.6483 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} +.7279 & -.6857 \\ +.6857 & +.7279 \end{pmatrix} \;=\; \begin{pmatrix} .1097 & .9940 \\ .9940 & .1097 \end{pmatrix}.$$

Gaussian elimination can be applied directly to this matrix without any use of pivoting. For example, with $\mathbf{b} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$,

$$U\,\mathbf{b} \;=\; \begin{pmatrix} +.6483 & -.7614 \\ +.7614 & +.6483 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \end{pmatrix} \;=\; \begin{pmatrix} -.9870 \\ 3.468 \end{pmatrix}.$$

Naked Gaussian elimination on $U\,A\,V\,\mathbf{y} = U\,\mathbf{b}$ then gives

$$\mathbf{y} \;=\; \begin{pmatrix} 3.556 \\ -.6005 \end{pmatrix},$$

so the solution to $A\mathbf{x} = \mathbf{b}$ is

$$\mathbf{x} \;=\; V\,\mathbf{y} \;=\; \begin{pmatrix} 3.000 \\ 2.001 \end{pmatrix}$$

— correct to three digits.

3.3 What Randomization Must Accomplish to Replace Pivoting

Gaussian elimination works only when the $a_{kk}^{(k)}$ are nonzero, for $1 \leq k \leq n$. It is well-known that this is equivalent to the following:

Definition 1 *An $n \times n$ matrix A is* **Gauss eliminable** *if* $det\,A[\,1,\ldots,k\,|\,1,\ldots,k\,] \neq 0$, $1 \leq k \leq n$.

As above, assume that the process of randomization transforms a nonsingular input matrix A to UAV, for some random matrices U and V. To be useful in Gaussian elimination, then, this randomizing transform needs to accomplish several things:

1. The result of randomization must be Gauss eliminable.
2. The cost of randomization must be reasonable.
3. Ideally, the randomization process should be easily invertible.
4. The roundoff errors incurred in the randomization, and in the subsequent Gaussian elimination (without pivoting), must be acceptable.

Simply multiplying by arbitrary random matrices will achieve the first goal. However, this multiplication will be expensive, requiring more time than Gaussian Elimination itself. Also, it is nontrivial to invert this transformation. Furthermore, multiplying by arbitrary random matrices can lose numerical accuracy. We should therefore consider classes of random matrices that, at least on some machines, justify their expense in time and accuracy.

In the next section we describe a specific randomizing transform that is both efficient and has the property that, for any nonsingular matrix A, the transform of A is Gauss eliminable with probability 1. Thus, with extremely high probability, Gaussian elimination will succeed.

The magnitude of roundoff errors incurred without pivoting can be very significant when the matrix A is poorly conditioned. In [15] we study roundoff errors in Gaussian elimination. We give explicit formulas for $a_{ij}^{(k)}$ and show $|\,a_{ij}^{(k)}\,|$ is bounded by the condition number of the leading $k \times k$ principle submatrix of A. This gives a deeper appreciation for the backwards error bounds mentioned in Theorem 1. Still, the point is that a randomized version of Gaussian elimination will, very likely, be less stable than one with pivoting.

Ultimately we feel a statistical error analysis like that in [24] is important for testing any real variant of Gaussian elimination. Backwards error analyses, useful though they are, give only partial certification of any numerical algorithm.

4 Random Butterfly Transforms

In the paper [16] a randomizing transformation is presented that accomplishes the goals listed above. It is called the Random Butterfly Transform (RBT), and uses random matrices that can be expressed as a product of (sparse) butterfly matrices.

4.1 Recursive Butterfly Matrices

We define a class of matrices that make computationally efficient transformations.

Definition 2 *A* **butterfly matrix** *will be defined here as any $n \times n$ matrix*

$$B^{\langle n \rangle} \quad = \quad \frac{1}{\sqrt{2}} \begin{pmatrix} R_0 & R_1 \\ R_0 & -R_1 \end{pmatrix}$$

where $n \geq 2$ and R_0 and R_1 are diagonal and nonsingular $(n/2) \times (n/2)$ matrices.

When R_0 and R_1 are both unitary (i.e., both have the form $exp(\mathrm{i}\Theta)$ for real diagonal Θ) then $B^{\langle n \rangle}$ is unitary. An important special case is where $\Theta = 0$, in which case $B^{\langle n \rangle}$ is

$$O_n \quad = \quad \frac{1}{\sqrt{2}} \begin{pmatrix} I_{n/2} & I_{n/2} \\ I_{n/2} & -I_{n/2} \end{pmatrix}.$$

In particular, $B^{\langle n\rangle} = O_n R$, *where* $R = (R_0 \oplus R_1)$ *is a diagonal matrix.*

A **Recursive Butterfly** *is a product of butterfly matrices. Specifically, a* 1×1 *matrix* $U^{\langle 1\rangle}$ *is a recursive butterfly, as is any matrix* $U^{\langle n\rangle}$ *with* $n = 2^\nu$ *that is the product of the direct sum of two smaller recursive butterflies* $U_0^{\langle n/2\rangle}$, $U_1^{\langle n/2\rangle}$, *with an* $n \times n$ *butterfly* $B^{\langle n\rangle}$*:*

$$U^{\langle n\rangle} \quad = \quad (U_0^{\langle n/2\rangle} \;\oplus\; U_1^{\langle n/2\rangle}) \;\; B^{\langle n\rangle} \quad = \quad \begin{pmatrix} U_0^{\langle n/2\rangle} & 0 \\ 0 & U_1^{\langle n/2\rangle} \end{pmatrix} \;\; B^{\langle n\rangle}.$$

When $n = 4$, the recursive butterfly definition can be manipulated as follows:

$$\begin{aligned} U^{\langle 4\rangle} &= (U_0^{\langle 2\rangle} \;\oplus\; U_1^{\langle 2\rangle}) \quad B^{\langle 4\rangle} \\ &= (((U_{00}^{\langle 1\rangle} \;\oplus\; U_{01}^{\langle 1\rangle}) \;\; B_0^{\langle 2\rangle}) \;\oplus\; ((U_{10}^{\langle 1\rangle} \;\oplus\; U_{11}^{\langle 1\rangle}) \;\; B_1^{\langle 2\rangle})) \quad B^{\langle 4\rangle} \\ &= (U_{00}^{\langle 1\rangle} \;\oplus\; U_{01}^{\langle 1\rangle} \;\oplus\; U_{10}^{\langle 1\rangle} \;\oplus\; U_{11}^{\langle 1\rangle}) \quad (B_0^{\langle 2\rangle} \;\oplus\; B_1^{\langle 2\rangle}) \quad B^{\langle 4\rangle}. \end{aligned}$$

The definition for $U^{\langle 4\rangle}$ can be appreciated visually:

$$\begin{aligned} U^{\langle 4\rangle} &= \frac{1}{2} \;\; (U_{00}^{\langle 1\rangle} \oplus U_{01}^{\langle 1\rangle} \oplus U_{10}^{\langle 1\rangle} \oplus U_{11}^{\langle 1\rangle}) \qquad (B_0^{\langle 2\rangle} \;\oplus\; B_1^{\langle 2\rangle}) \qquad\qquad B^{\langle 4\rangle} \\ &= \frac{1}{2} \begin{pmatrix} r_1^{\langle 1\rangle} & & & \\ & r_2^{\langle 1\rangle} & & \\ & & r_3^{\langle 1\rangle} & \\ & & & r_4^{\langle 1\rangle} \end{pmatrix} \begin{pmatrix} r_1^{\langle 2\rangle} & r_2^{\langle 2\rangle} & & \\ r_1^{\langle 2\rangle} & -r_2^{\langle 2\rangle} & & \\ & & r_3^{\langle 2\rangle} & r_4^{\langle 2\rangle} \\ & & r_3^{\langle 2\rangle} & -r_4^{\langle 2\rangle} \end{pmatrix} \begin{pmatrix} r_1^{\langle 4\rangle} & & r_3^{\langle 4\rangle} & \\ & r_2^{\langle 4\rangle} & & r_4^{\langle 4\rangle} \\ r_1^{\langle 4\rangle} & & -r_3^{\langle 4\rangle} & \\ & r_2^{\langle 4\rangle} & & -r_4^{\langle 4\rangle} \end{pmatrix} \\ &= \frac{1}{2} \begin{pmatrix} r_1^{\langle 1\rangle}\, r_1^{\langle 2\rangle}\, r_1^{\langle 4\rangle} & r_1^{\langle 1\rangle}\, r_2^{\langle 2\rangle}\, r_2^{\langle 4\rangle} & r_1^{\langle 1\rangle}\, r_1^{\langle 2\rangle}\, r_3^{\langle 4\rangle} & r_1^{\langle 1\rangle}\, r_2^{\langle 2\rangle}\, r_4^{\langle 4\rangle} \\ r_2^{\langle 1\rangle}\, r_1^{\langle 2\rangle}\, r_1^{\langle 4\rangle} & -\, r_2^{\langle 1\rangle}\, r_2^{\langle 2\rangle}\, r_2^{\langle 4\rangle} & r_2^{\langle 1\rangle}\, r_1^{\langle 2\rangle}\, r_3^{\langle 4\rangle} & -\, r_2^{\langle 1\rangle}\, r_2^{\langle 2\rangle}\, r_4^{\langle 4\rangle} \\ r_3^{\langle 1\rangle}\, r_3^{\langle 2\rangle}\, r_1^{\langle 4\rangle} & r_3^{\langle 1\rangle}\, r_4^{\langle 2\rangle}\, r_2^{\langle 4\rangle} & -\, r_3^{\langle 1\rangle}\, r_3^{\langle 2\rangle}\, r_3^{\langle 4\rangle} & -\, r_3^{\langle 1\rangle}\, r_4^{\langle 2\rangle}\, r_4^{\langle 4\rangle} \\ r_4^{\langle 1\rangle}\, r_3^{\langle 2\rangle}\, r_1^{\langle 4\rangle} & -\, r_4^{\langle 1\rangle}\, r_4^{\langle 2\rangle}\, r_2^{\langle 4\rangle} & -\, r_4^{\langle 1\rangle}\, r_3^{\langle 2\rangle}\, r_3^{\langle 4\rangle} & r_4^{\langle 1\rangle}\, r_4^{\langle 2\rangle}\, r_4^{\langle 4\rangle} \end{pmatrix}. \end{aligned}$$

When the values r_i are selected appropriately (e.g., when all the r_i values are roots of unity), recursively butterflies U turn out to be *unitary*, i.e., the adjoint (conjugate transpose) U^* is U^{-1}. Thus unitary recursive butterflies are easily invertible.

4.2 Recursive Butterflies are Related to the FFT

There is a strong connection between recursive butterfly matrices and the FFT. The **Discrete Fourier Transform (DFT)** is just the unitary $n \times n$ matrix

$$F_n \quad = \quad \frac{1}{\sqrt{n}} \, (\, \omega_n^{(i-1)(j-1)} \,)$$

where typically $\omega_n = exp(-\frac{2\pi}{n}\,\mathrm{i})$. The **Fast Fourier Transform (FFT)** results from factoring F_n into recursive products of butterfly transformations and permutations. Specifically:

$$F_{2n} \quad = \quad B_{2n} \;\; (F_n \oplus F_n) \;\; {\sigma_{2n}}^{-1} \quad = \quad B_{2n} \;\; (I_2 \otimes F_n) \;\; {\sigma_{2n}}^{-1}$$

where I_n is the $n \times n$ identity and B_{2n} is the butterfly

$$B_{2n} \quad = \quad \frac{1}{\sqrt{2}} \begin{pmatrix} I_n & D_n \\ I_n & -D_n \end{pmatrix}, \qquad \text{where} \qquad D_n \quad = \quad diag(\, 1,\; \omega_{2n},\; {\omega_{2n}}^2,\; \ldots,\; {\omega_{2n}}^{n-1} \,)$$

and ${\sigma_{2n}}^{-1}$ is the 'perfect unshuffle' permutation [14, 25]. Thus the adjoint, or inverse DFT, ${F_{2n}}^*$ is a recursive butterfly preceded on the left by a product of unshuffle permutations (a bit-reversal permutation). Van Loan [25] reviews how F_n can be implemented efficiently on high-performance computers. Recursive butterflies can be also.

4.3 The RBT yields Gauss Eliminable Matrices

In [16] we define a **Random Butterfly Transformation (RBT)** of a square matrix A to be a product

$$\widetilde{A} \quad = \quad U^* \, A \, V$$

where U and V are random recursive butterflies. The use of the adjoint (transpose conjugate) operator U^* is actually significant, as it permits proof of the following theorem:

Theorem 2 *Let A be an $n \times n$ nonsingular matrix, where n is a power of 2. If U and V are random recursive butterflies, then, with probability 1, the RBT $\widetilde{A} = U^* \, A \, V$ is Gaussian eliminable.*

The proof shows that, with probability 1,

$$det\, \widetilde{A}[\, 1, \ldots, k \,|\, 1, \ldots, k \,] \quad \neq \quad 0$$

for $1 \leq k \leq n$.[4] The proof relies on the Binet-Cauchy theorem, which gives an explicit formula for the determinant of a product of matrices in terms of the determinants of various submatrices. We show that the entire determinant expands to be a polynomial in the random variables r_i appearing in the random butterflies. It is not difficult to show that this polynomial, which has a finite number of zeroes, is nonzero with probability 1.

In [17] a variation of the RBT approach is presented that is well-suited to block matrix computations. It turns out that recursive block algorithms can be developed using individual butterflies for randomization. (In other words, we can interleave the recursions of the block algorithm and the randomization.)

4.4 The RBT Yields Acceptable Errors on Well-Conditioned Matrices

To test the concepts presented above, we implemented the RBT in FORTRAN 77, and compared a randomized version of Gaussian Elimination using it with the standard LINPACK `dgefa/dgesl` routines on a Sun Sparcstation 20 at UCLA. Results are tabulated in Figure 1 for five random classes of matrices in [24].

The current implementation is straightforward. It is a sequential program, and does not exploit the parallelism of the algorithm. Also, it is not optimized. For example, Brad Pierce at UCLA has noticed a clever way to reduce the time required by the RBT by a factor of 2, and we will report on this soon. The overhead of random number generation was significant, and dominated the execution times, so we suspect it will be a factor in any future implementation. The current implementation generates an initial set of RBT random values for each value of n, and reuses these in all runs for n, as a simple-minded way of avoiding the overhead. The RBT uses random diagonal matrices with diagonal values $exp(\ r/10\ )$ where the uniform random variables r range over $[-1/2, 1/2]$. The determinant of these butterflies thus has expected value 1. The width of this interval was made a parameter that could be varied, but moderate variations in the width have not affected the roundoff error greatly.

[4] In fact the statement of this theorem in [16] is somewhat more general, holding not only for $1, \ldots, k$, but for any consecutive sequence of indices.

Matrix class	Problem: $A\mathbf{x} = \mathbf{b}$
`normal`	normal random a_{ij} and b_i ($\mu = 0,\ \sigma = 1$)
`[-1,1]`	uniform random a_{ij} and b_i, drawn from $[-1, 1]$
`[0,1]`	uniform random a_{ij} and b_i, drawn from $[0, 1]$
`{-1,1}`	discrete random a_{ij} and b_i ($pr(-1) = pr(1) = 0.5$)
`{0,1}`	discrete random a_{ij} and b_i ($pr(0) = pr(1) = 0.5$)

Matrix Class	Matrix Rank	Runs	RBT avg sec	LINPACK avg sec	avg Error RBT-LINPACK	max Error RBT-LINPACK
`normal`	32	256	0.0754	0.0621	1.66D-12	6.97D-10
`[-1,1]`	32	256	0.0708	0.0617	1.64D-12	2.71D-10
`[0,1]`	32	256	0.0709	0.0613	3.51D-12	7.46D-10
`{-1,1}`	32	256	0.0708	0.0538	1.73D-12	1.93D-10
`{0,1}`	32	256	0.0707	0.0532	2.30D-12	2.67D-10
`normal`	64	128	0.5040	0.2862	3.01D-12	8.83D-10
`[-1,1]`	64	128	0.4971	0.2867	3.89D-12	4.79D-10
`[0,1]`	64	128	0.4965	0.2864	3.23D-12	3.44D-10
`{-1,1}`	64	128	0.4966	0.2489	1.77D-11	5.65D-09
`{0,1}`	64	128	0.4970	0.2516	2.63D-12	2.78D-10
`normal`	128	64	2.9080	1.8112	4.74D-11	1.09D-08
`[-1,1]`	128	64	2.8932	1.8122	5.54D-11	1.08D-08
`[0,1]`	128	64	2.8891	1.8098	7.54D-12	8.26D-10
`{-1,1}`	128	64	2.8935	1.6784	4.25D-12	1.38D-10
`{0,1}`	128	64	2.8845	1.6769	7.84D-12	3.85D-10
`normal`	256	32	26.1787	15.8590	2.81D-11	1.07D-09
`[-1,1]`	256	32	26.1590	15.8459	3.97D-10	1.97D-08
`[0,1]`	256	32	26.1903	15.8391	9.78D-11	7.80D-09
`{-1,1}`	256	32	26.1119	15.1486	4.17D-11	1.38D-09
`{0,1}`	256	32	26.2068	15.2045	3.33D-11	1.96D-09
`normal`	512	20	248.0461	173.0853	4.75D-10	1.67D-08
`[-1,1]`	512	20	248.1719	173.0854	2.14D-10	7.56D-09
`[0,1]`	512	20	248.1133	173.0481	7.41D-10	2.96D-08
`{-1,1}`	512	20	248.1001	169.9761	3.29D-10	7.82D-09
`{0,1}`	512	20	248.0032	169.6792	7.32D-10	4.15D-08

Figure 1: Results of simple implementation: RBT vs. LINPACK's `dgefa/dgesl`

Rounding errors do grow rapidly with the problem size n. The final columns in the table give averages and maxima of 'relative errors'

$$\frac{|\, b_i^{RBT} \; - \; b_i \,|}{\| \, \mathbf{b}^{RBT} \, \|_\infty}$$

where the b_i values are computed solutions from LINPACK. The random matrices considered here are well-conditioned. Edelman [4] has shown that normally distributed random matrices have expected spectral norm essentially $\sqrt{2n}$. However, For poorly-conditioned problems the RBT has large roundoff errors, which are somewhat worse than those of LINPACK. Although the errors were not as bad as we feared, they do suggest the RBT approach as implemented here should be limited to well-behaved problems.

In spite of the fact that the code is not optimized or refined, the results show that the approach is already usable, and for the test matrices used often gives numerical results comparable in quality to that of LINPACK's `dgefa/dgesl`.

5 Conclusion

In this paper we have introduced a new method for implementing Gaussian elimination and related numeric computations, based on the idea of a randomizing linear transform. We have sketched how the use of such transforms can make it possible to implement Gaussian elimination on arbitrary matrices without use of pivoting.

There is no real limit to the number of randomization schemes possible. All that is required is that the matrix A undergoing Gaussian elimination becomes Gauss eliminable when randomized.

The randomization approach also suggests many interesting developments beyond the elimination of pivoting. Randomization is a new tool for developing matrix computations that are suitable for high-performance computers. High-performance machines tend to favor algorithmic simplicity. We have traded naive time complexity (the number of 'steps' performed by an algorithm) for algorithmic simplicity. The result may be better performance, since naive time complexity often does not correspond to real time complexity.

References

[1] E. Bampis, J.C. Konig, D. Trystram, "Impact of communications on the complexity of the Parallel Gaussian Elimination", *Parallel Computing* **17**:1, 55–61, 1991.

[2] J.R. Bunch, J. Hopcroft, "Triangular Factorization and Inversion by Fast Matrix Multiplication", *Math. of Computation* **28**:125, 231–236, 1974.

[3] J. Dongarra, I.S. Duff, D.C. Sorensen, H.A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, Philadelphia, PA: SIAM Press, 1991.

[4] A. Edelman, "Eigenvalues and Condition Numbers of Random Matrices", *SIAM J. Matrix Anal. Appl.* **9**:4, 543–560, 1988.

[5] D.K. Fadeev, V.N. Fadeeva, *Computational Methods of Linear Algebra*, San Francisco, W.H. Freeman, 1963.

[6] G.E. Forsythe, C.B. Moler, *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, 1967.

[7] G.H. Golub, C.F. Van Loan, *Matrix Computations: Second Edition*, Baltimore: Johns Hopkins University Press, 1989.

[8] R.A. Horn, C.R. Johnson, *Matrix Analysis*, NY: Cambridge University Press, 1985; reprinted 1992.

[9] R.A. Horn, C.R. Johnson, *Topics in Matrix Analysis*, NY: Cambridge University Press, 1991.

[10] N.J. Higham, D.J. Higham, "Large Growth Factors in Gaussian Elimination with Pivoting", *SIAM J. Matrix Anal. Appl.* **10**:3, 155-164, April 1989.

[11] N.J. Higham, "Algorithm 694: A Collection of Test Matrices in MATLAB", *ACM Trans. on Math. Software* **17**:3, 289–305, 1991.

[12] A.W. Marshall, I. Olkin, *Inequalities: Theory of Majorization and Its Applications*, NY: Academic Press, 1979.

[13] J. von Neumann, H.H. Goldstine, "Numerical inverting of matrices of high order", *Bull. Amer. Math. Soc.* **53**, 1021–1099, 1947.

[14] D.S. Parker, "Notes on Shuffle/Exchange-Type Switching Networks", *IEEE Trans. Comput.* **C-29:3**, 213-222 (March 1980).

[15] D.S. Parker, "Explicit Formulas for the Results of Gaussian Elimination", Technical Report CSD-920025, UCLA Computer Science Department, 1995.

[16] D.S. Parker, "Random Butterfly Transformations with Applications in Computational Linear Algebra", Technical Report CSD-950023, UCLA Computer Science Department, 1995.

[17] D.S. Parker, "A Randomizing Butterfly Transformation Useful in Block Matrix Computations", Technical Report CSD-950024, UCLA Computer Science Department, 1995.

[18] D.S. Parker, D. Lê, "Quadtree Matrix Algorithms Revisited: Basic Issues and their Resolution", Technical Report CSD-950028, UCLA Computer Science Department, 1995.

[19] W.H. Press, B.P. Flanery, S.A. Teukolsky, W.T. Vetterling, *Numerical Recipes in Pascal — The Art of Scientific Computing*, Cambridge University Press, 1989.

[20] F.F. Rivera, R. Doallo, J.D. Bruguera, E.L. Zapata, R. Peskin, "Gaussian Elimination with Pivoting on Hypercubes", *Parallel Computing* **14**:1, 51–60, 1990.

[21] Y. Robert, *The Impact of Vector and Parallel Architectures on the Gaussian Elimination Algorithm*, NY: Halsted Press, 1990.

[22] A. Schrijver, *Theory of Linear and Integer Programming*, NY: J. Wiley & Sons, 1986.

[23] L.N. Trefethen, "Three mysteries of Gaussian elimination", *ACM SIGNUM Newsletter* **20**:4, October 1985.

[24] L.N. Trefethen, R.S. Schreiber, "Average-Case Stability of Gaussian Elimination", *SIAM J. Matrix Anal. Appl.* **11**:3, 335–360, July 1990.

[25] C. Van Loan, *Computational Frameworks for the Fast Fourier Transform*, Philadelphia, PA: SIAM Press, 1992.

[26] M. Veldhorst, "Gaussian Elimination with Partial Pivoting on an MIMD computer", *J. Parallel Distr. Computing* **6**, 62–68, 1989.

[27] J.H. Wilkinson, "Error analysis of direct methods of matrix inversion", *J. ACM* **8**, 281–330, 1961.

[28] J.H. Wilkinson, *Rounding Errors in Algebraic Processes*, NJ: Prentice-Hall, Inc., 1963.

[29] David S. Wise, Matrix Algebra and Applicative Programming, in Gilles Kahn (ed.), *Functional Programming Languages and Computer Architecture* (Proceedings), Portland, Oregon, USA, Springer LNCS 274, 134–153, September 1987.

[30] S.J. Wright, "A Collection of Problems for which Gaussian Elimination with Partial Pivoting is Unstable", *SIAM J. Sci. Comput.* **14**:1, 231–238, January 1993.