RESEARCH
COUNCILS UK

HECToR

CSC / NAG Autumn School on

Core Algorithms in High-Performance Scientific Computing

Maths I

Dwight Barkley

Basic Concepts

Maths I: Basic Concepts

At the innermost core of this subject are vectors and matrices. In this first lecture we shall review these basic concepts and introduce notation and definitions used throughout the remainder of the lectures.

1.1 Notation

Vectors: We typically denote vectors by lower case letters. These are understood to be column vectors. For example, let x be a vector of real numbers of length m and let y be a vector of complex numbers of length n. Then $x \in \mathbb{R}^m$ and $y \in \mathbb{C}^n$.

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_j \\ \vdots \\ x_m \end{pmatrix} = (x_1, x_2, \cdots, x_j, \cdots, x_m)^T \qquad y = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_j \\ \vdots \\ y_{n-1} \\ y_n \end{pmatrix} = (y_1, y_2, y_3, \cdots, y_j, \cdots, y_{n-1}, y_n)^T$$

where $x_1, x_2, \cdots, x_j, \cdots, x_m$ are the real components of x, and $y_1, y_2, \cdots, y_j, \cdots, y_n$ are the complex components of y.

Throughout these notes a superscript T will always denote transpose. In this case the transpose of a row vector is a column vector.

Matrices: We typically denote matrices by upper case letters. For example, let A be a real m by n matrix (m rows and n columns). Then $A \in \mathbb{R}^{m\times n}$. We write

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \quad \text{or sometimes} \quad \mathrm{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & & \vdots \\ A_{m1} & A_{m2} & \cdots & A_{mn} \end{pmatrix}$$

where the matrix elements are denoted a_{ij} or A_{ij}.

As we shall see sometimes, for theoretic and practical reasons, it is useful to think of A in terms of its columns and we write:

$$A = \left(\begin{array}{c|c|c|c} c_1 & c_2 & \cdots & c_n \end{array} \right)$$

where the c_j are m-vectors, the columns of A, e.g.

$$c_1 = \begin{pmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{pmatrix}$$

We can also consider the rows of A

$$A = \begin{pmatrix} r_1 \\ \hline r_2 \\ \hline \vdots \\ \hline r_m \end{pmatrix}$$

Scalars. These are simply real or complex numbers. Often they are denoted by Greek letters, α, β, etc.

Note that an n-dimensional column vector is a $n \times 1$ matrix. A scalar is a 1×1 matrix.

Bar, star, T and H

We shall use the following notation regarding complex conjugation and transposition.

Complex conjugation of a scalar α is denoted $\bar{\alpha}$. If α is real then $\bar{\alpha} = \alpha$.

The **Transpose** of an $m \times n$ matrix A is an interchange of the rows and columns of A: $a_{ij} \to a_{ji}$. This can be thought of as flipping A across its diagonal. The transpose is denoted A^T, so

$$A^T = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}^T = \begin{pmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{pmatrix}$$

Examples:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix}^T = \begin{pmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \end{pmatrix} \qquad x^T = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}^T = (x_1, x_2, \cdots, x_m) \qquad (A^T)^T = A$$

A **symmetric matrix** is a matrix A such that $A^T = A$. Symmetric matrices are necessarily square. These are frequently denoted by S to emphasize that they are symmetric. Note that a symmetric $n \times n$ matrix is specified by only $(n^2 + n)/2$ distinct values, not n^2.

The **adjoint** or **Hermitian conjugate** of an $m \times n$ matrix A is the conjugate transpose of A. This is commonly denoted by A^* or by A^H, so

$$A^* = A^H = \begin{pmatrix} \bar{a}_{11} & \bar{a}_{21} & \cdots & \bar{a}_{m1} \\ \bar{a}_{12} & \bar{a}_{22} & \cdots & \bar{a}_{m2} \\ \vdots & \vdots & & \vdots \\ \bar{a}_{1n} & \bar{a}_{2n} & \cdots & \bar{a}_{mn} \end{pmatrix}$$

Examples:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix}^H = \begin{pmatrix} \bar{a}_{11} & \bar{a}_{21} & \bar{a}_{31} \\ \bar{a}_{12} & \bar{a}_{22} & \bar{a}_{32} \end{pmatrix} \qquad x^* = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}^* = (\bar{x}_1, \bar{x}_2, \cdots, \bar{x}_m) \qquad \left(A^H\right)^H = A$$

A **Hermitian** or **self-adjoint matrix** is a (necessarily square) matrix A such that $A^H = A$.

For the majority of the theory, we do not need to distinguish between real and complex cases. Whenever possible we will keep things simple and refer to n-vectors or $m \times n$ matrices. It is common, particularly in numerical software, to distinguish the real and complex cases. At times we will want to make this distinction. For example, even though A^* and A^H mean the same thing, we will use A^H when we want to emphasize that the entries in A are known to be complex.

Symbol	Meaning	Comment on usage in these lectures
$\bar{}$	Complex Conjugate	Used only for scalars
T	Transpose	No complex conjugation, even for complex entries
$*$	Adjoint	Complex transpose, for both real or complex matrices
H	Hermitian conjugate	Same as $*$; favoured when entries are known to be complex

Note: in some texts the matrix adjoint, typically written adj(A), has a different meaning from that used in these lectures and in most numerical linear algebra books.

Theory vs Practice

Already we begin to encounter possible differences between "theory", that is the equations and algorithms we shall state, and what might be used when implemented in practice when calling numerical libraries. We will alert the reader to these potential differences in Theory vs Practice boxes.

- In the theoretical treatment here and in most books, vectors and matrices will be 1-based, e.g. the n-vector x has components x_1 to x_n. In practice what is used depends on several things, most notably the programming language.
- While often the distinction between real and complex cases is best ignored in theoretical discussion, in practice one often needs to distinguish carefully between real and complex cases.
- In the theoretical treatment we do not concern ourselves with how values are stored in memory, but in practice this is very important.

1.2 Elementary Operations

Inner (dot) product

The standard inner product, also known as dot product or scalar product, is

$$\langle x, y\rangle = \sum_{i=1}^{n} \bar{x}_i y_i$$

Using the $*$ notation, this is conveniently written

$$\langle x, y\rangle = x^* y$$

For real vectors

$$\langle x, y\rangle = x^T y = \sum_{i=1}^{n} x_i y_i$$

The dot notation, $\mathbf{x} \cdot \mathbf{y}$, will not be used in these lectures.

Two vectors are **orthogonal** if their inner product is zero.

Some further details:
The inner product is **bilinear** so that:

$$\begin{aligned}(x_1 + x_2)^* y &= x_1^* y + x_2^* y\\ x^*(y_1 + y_2) &= x^* y_1 + x^* y_2\\ (\alpha x)^*(\beta y) &= \bar{\alpha}\beta x^* y\end{aligned}$$

The inner product is said to be linear in the second component y, and anti-linear in the first component, x.

More generally one can define an inner product by

$$\langle x, y\rangle_W = x^* W y$$

where W is a Hermitian positive-definite matrix.

Vector norms

The standard inner product gives rise to the **standard vector norm** via

$$||x|| = \sqrt{\langle x, x\rangle} = (\sum_{i=1}^{n} |x_i|^2)^{1/2}$$

It is possible to define other vector norms and these play an important role in numerical linear algebra, since in particular they are used to quantify accuracy. The 3 most important vector norms are:

- **1-norm:** $||x||_1 = \sum_{i=1}^{n} |x_i|$.
- **2-norm or Euclidean norm:** $||x||_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2} = \sqrt{x^* x}$.
- **∞-norm:** $||x||_\infty = \max_{1\leq i\leq n} |x_i|$.

The standard norm, 2-norm, and Euclidean norm all mean the same thing. If we write $||x||$ with no subscript, you should assume the standard norm or that the particular norm is not important.

Matrix-vector product

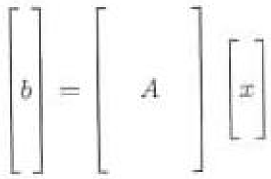
Let A be an $m \times n$ matrix and let x be an n-vector and b be an m-vector. Then

$$b = Ax$$

is given in component form by:

$$b_i = \sum_{j=1}^{n} a_{ij}\, x_j$$

It is often nice (although difficult to typeset) to represent such operations pictorially by square brackets, as this emphasizes the relative dimensions of the vectors and matrices, e.g. for $m > n$ we might show:



One can view the matrix-vector product as:

- Components of b are given by dot products of rows of A with x.

  $$b_i = (r_i, x)$$

  The vectors r_i's are the rows of A. The computation of b is given by m dot products of length n.

- b as a weighted sum of the columns of A

  $$b = x_1 c_1 + \cdots + x_j c_j + \cdots + x_n c_n = \sum_{j=1}^{n} x_j\, c_j$$

  The vectors c_j's are the columns of A. The computation of b is given by n saxpy operations (you will learn what these are) of length m. This second view may be less familiar but is quite important.

Matrix norms

Given a vector norm, the induced matrix norm is defined as

$$||A|| = \sup_{||x||=1} ||Ax||$$

The norm on the left is the matrix norm while the one on the right is a vector norm. The definition says that the norm (size) of the matrix A is given by the largest length of Ax for x on the unit ball. As with vector norms, the most common matrix norms are the 1-norm, 2-norm and ∞-norms induced from the corresponding vector norms.

The definition of matrix norm implies that

$$||Ax|| \leq ||A||\, ||x||$$

where the norms are understood to be any vector norm and corresponding induced matrix norm. This is a key relationship between vector and matrix norms that is used throughout the analysis of this subject.

Matrix-matrix product

This is where the fun begins in actual hardware implementation. For now we simply note that given an $m \times n$ matrix A, an $n \times \ell$ matrix B and an $m \times \ell$ matrix C, we have

$$C = A\,B \quad \text{or in component form} \quad c_{ij} = \sum_{k=1}^{n} a_{ik}\, b_{kj}$$

We shall refer to this sum as the **standard method**.

Pictorially, for $n < m < \ell$,

$$\begin{bmatrix} & & \\ & C & \\ & & \end{bmatrix} = \begin{bmatrix} \\ A \\ \\ \end{bmatrix} \begin{bmatrix} & & & B & & & \end{bmatrix}$$

Once again it is possible to view the computations as element-by-element dot products (the standard method which probably is the way you learned in school) or as linear combinations of columns of A (something that will be helpful later). However, ...

Theory vs Practice

In practice, matrix-matrix multiplication would almost never be performed in either of these two ways because these ways are inefficient. Performance libraries order operations to be maximally efficient on actual computer hardware.

1.3 Simple Examples

Matrix-matrix product reduces to other cases

For example:

- **Multiplication of scalars:** $m = n = \ell = 1$

$$[C] = [A]\ [B]$$

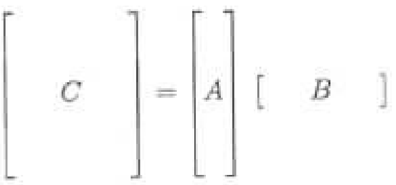
- **Dot Product:** $m = \ell = 1, n > 1$

$$[C] = \begin{bmatrix} & A & \end{bmatrix} \begin{bmatrix} \\ B \\ \\ \end{bmatrix}$$

- **Outer Product:** The outer product of two (real) vectors x and y is

$$xy^T = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_j \\ \vdots \\ x_m \end{pmatrix} (y_1, y_2, \cdots, y_j, \cdots, y_n) = \begin{pmatrix} x_1y_1 & x_1y_2 & \cdots & x_1y_n \\ x_2y_1 & x_2y_2 & \cdots & x_2y_n \\ \vdots & \vdots & & \vdots \\ x_my_1 & x_my_2 & \cdots & x_my_n \end{pmatrix}$$

corresponding to $n = 1, m > 1, \ell > 1$



1.4 Matrix Factorizations and Special Matrices

If you look at any library of numerical linear algebra routines, you will immediately see many references to things like QR and LU etc, where Q, R, L and U are matrices with particular structure. One may take the view that much of numerical linear algebra consists of finding and understanding clever ways of factorizing matrices into products of matrices with special structure, e.g. $A = QR$.

We give a list of important special matrices and factorizations at the end of the chapter for reference. We will consider these in more detail as they arise. Here we emphasize two points.

1. In pictures, non-zero entries are indicated by $\cdot$, $\times$ or by lines (when emphasizing columns, rows, and diagonals). Zero entries are blank.

2. We frequently consider rectangular matrices, especially those with $m > n$, and terminology such as diagonal matrix, applies to these as well as to square matrices.

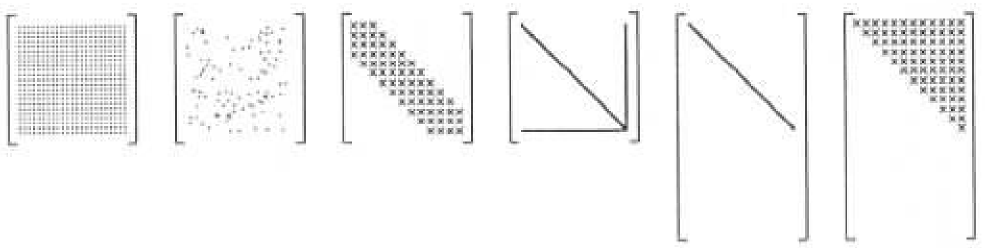


Figure 1.1: From left to right: dense matrix with non-zero entries, located almost everywhere, indicated by points; sparse matrix with far fewer than n^2 non-zero entries scattered throughout the matrix; banded matrix with non-zero entries indicated by $\times$; matrix with a non-zero row, column and diagonal; diagonal rectangular matrix; upper triangular rectangular matrix.

A Real Example

A less trivial example of matrix-vector multiplication, which is also practical and will be relevant to the CFD case study, is the following. Consider the discrete representation of a function $u(x)$ as a vector $\mathbf{U}$ with components $U_j = u(x_j)$. For simplicity suppose the values of x_j are equally spaced on a grid between $x = 0$ and $x = 1$ say with a spacing h. A simple finite-difference approximation to the first derivative of $u(x)$ at a grid point x_j is given by the formula:

$$\frac{\partial u}{\partial x}(x_j) \simeq \frac{-U_{j-1} + U_{j+1}}{2h}$$

Suppose we want to differentiate the function at all grid points. Let U' be a vector representing the first derivative on the grid $U'_j \simeq u'(x_j)$. Then

$$\begin{pmatrix} \vdots \\ \\ U'_j \\ \\ \vdots \end{pmatrix} = \begin{pmatrix} -\frac{1}{h} & \frac{1}{h} & & & & & & & \\ -\frac{1}{2h} & 0 & \frac{1}{2h} & & & & & & \\ & & \ddots & & & & & & \\ & & -\frac{1}{2h} & 0 & \frac{1}{2h} & & & & \\ & & & -\frac{1}{2h} & 0 & \frac{1}{2h} & & & \\ & & & & -\frac{1}{2h} & 0 & \frac{1}{2h} & & \\ & & & & & & \ddots & & \\ & & & & & & -\frac{1}{2h} & 0 & \frac{1}{2h} \\ & & & & & & & -\frac{1}{h} & \frac{1}{h} \end{pmatrix} \begin{pmatrix} \vdots \\ U_{j-1} \\ U_j \\ U_{j+1} \\ \vdots \end{pmatrix}$$

where necessarily a different approximation is used at the two end points. This can be written compactly as

$$\mathrm{U}' = \mathrm{D}_1\,\mathrm{U}$$

D_1 is called a differentiation matrix. In this case it is tridiagonal. Other, more accurate approximations to the first derivative give matrices with more non-zero diagonals.

Differentiation of a function is a real example of a matrix-vector product in which one could potentially be interested in quite long vectors (many grid points) and hence large matrices. Differentiation matrices have structure which can come into play in practice.

1.5 Complexity of Algorithms

The complexity of an algorithm $C(n)$ is the number of operations (time complexity) or storage locations (space complexity) as a function of the problem size n. The complexity may depend on more than one parameter, e.g. $C(m, n)$.

When considering operations, we will count the number of additions, subtractions, multiplications and divisions, and possibly square roots. Each such operation is one **flop (floating point operation).** We will use the terms complexity, cost and work interchangeably when referring to operation counts.

One typically is not very concerned with the exact operation count of an algorithm. Rather one is interested in the dominant operation count and especially how this behaves for large n.

For example, consider an algorithm for which the complexity is

$$C(n) = \frac{2}{3}n^3 + \frac{1}{2}n^2 - \frac{7}{6}n.$$

(LU factorization has this complexity.) The most important part of this expression is that the work grows as n^3 for large n, rather than some other power of n. The next most important feature is the coefficient $\frac{2}{3}$ in front of n^3. This might be significant, for example, in comparing two algorithms whose work grows as n^3.

The way to quantify the important parts of $C(n)$ is via **big O notation,** where for this example we write

$$C(n) = O(n^3)$$

and **asymptotic order** where for this example we write

$$C(n) \sim \frac{2}{3}n^3$$

We shall favour the second but use both.

Terminology for common cases

Order	Name
$O(n)$	**Linear**
$O(n \log n)$	**Log-linear** **(Linearithmic, or quasilinear)**
$O(n^2)$	**Quadratic**
$O(n^3)$	**Cubic**
$O(n^a), a > 0$	**polynomial or algebraic**

Complexity of Basic Operations

n vectors. $n \times n$ matrices. matrix-matrix multiply by standard method.

BLAS	Operation	CPU (flops)	Memory
Level 1	dot	$\sim 2n = O(n)$	$\sim 2n = O(n)$
Level 2	matrix-vector	$\sim 2n^2 = O(n^2)$	$\sim n^2 = O(n^2)$
Level 3	matrix-matrix	$\sim 2n^3 = O(n^3)$	$\sim 3n^2 = O(n^2)$

Some further details:

The formal definitions of **big O** and $\sim$ are

$$f(n) = O(g(n)) \text{ as } n \to \infty$$

if and only if there exists a positive real number M and an integer N such that $|f(n)| \leq M|g(n)|$ for $n > N$. (Or equivalently $\limsup_{n\to\infty} \left|\frac{f(n)}{g(n)}\right| < \infty$.)

$$f \sim g \quad (\text{as } n \to \infty)$$

if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$$

Divide and conquer

The divide-and-conquer approach is extremely powerful when it can be applied. The basic idea is to recursively break down a problem into two (or possibly more) sub-problems of the same type. The recursion continues until the sub-problems become sufficiently small and are solved directly. The solutions to the sub-problems are then re-combined to give a solution to the original problem.

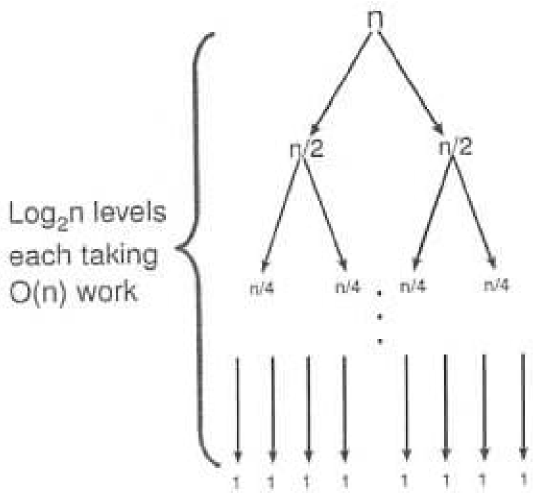


Figure 1.2: Illustration of Divide and Conquer

One of the most famous examples of this approach is the fast Fourier transform (FFT) which will be covered in other parts of the course. For the present we use it to illustrate recursion and the divide-and-conquer approach. The Fourier transform of a vector x of length n can be written:

$$y = \mathrm{FT}\, x$$

where FT is an $n \times n$ matrix and y is the result of the transform. Thus, naively, a Fourier transform takes $O(n^2)$ operations, and $O(n^2)$ memory locations to hold the matrix FT.

In practice this is not the case since the FFT works through a divide-and-conquer approach. Assume $n = 2^p$, although this is not necessary in the general case. It turns out that the problem can be written as two sub-problems each of size $n/2$. Each of these can be written as two further sub-problems each of size $n/4$ etc. There will be $p = \log_2 n$ such levels and when everything is taken into account the resulting work for the FFT is $O(n \log_2 n)$ rather than $O(n^2)$. Hence effectively it is possible to perform the matrix-vector multiplication with less than $O(n^2)$ work for this, and certain other, particular matrices.

It is largely through such recursion and divide-and-conquer that computational costs involving $\log n$ arise. A reduction from $O(n^2)$ to $O(n \log n)$, as in the FFT, is pretty much the ideal case. It is very significant in practice.

Beware that recursion does not necessarily lead to improvements in practice. The well-known **Strassen algorithm** for matrix-matrix multiplication uses recursion to reduce the cost of $O(n^3)$ for the standard method to $O(n^{\log_2 7})$. In practice though the method is essentially useless. (Google it if you are interested).

1.6 Convergence

The term **convergence** has different meanings in different contexts in numerical analysis. The basic context, however, is that the answer we want is obtained by a sequence of approximations. A method converges if the approximations reach our answer as the number of steps, or iterations, increases.

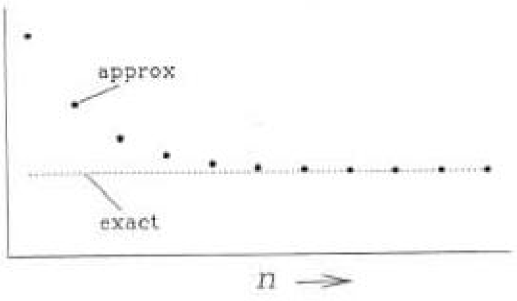


Figure 1.3: Convergence of approximations to an exact result

What is potentially confusing here is the way convergence rates are quantified in different setting. Without all the details:

- Consider iterative methods for root finding, eigenvalues, optimisation, etc discussed later. Let the error between the numerical approximation and the exact answer at step n be denoted e_n. Then one considers the **rate of convergence** of the approximation. For example, a method **converges quadratically** if

$$|e_{n+1}| < C|e_n|^2, \quad 0 < C.$$

- In contrast, consider solving an ordinary differential equation on the interval $0 \leq t \leq 1$ using N time steps of size $\triangle t = 1/N$. Let e_N represent the error at time $t = 1$. In this context a method is called **second order** if:

$$e_N = O(\triangle t^2) = O(N^{-2})$$

While the terms *quadratic* and *second order* seem similar, they refer to very different behaviors.

1.7 Stability and Conditioning

The term **stability** also has several distinct meanings in numerical analysis. The basic idea, however, is that an algorithm is stable if it does not diverge nor produce bad values from good input. This latter case is the main issue for the algorithms we consider in these lectures.

We seek algorithms that keep the roundoff error arising from finite-precision arithmetic from being amplified in some bad way. This is not to say that roundoff errors do not accumulate for stable algorithms - they do. The point is that they do not grow any worse than expected for the difficulty of the problem at hand.

Leaving precise definitions aside, an algorithm is stable if the error it produces satisfies:

$$\text{Relative Error} \leq (\text{Condition Number})\, O(\epsilon_{machine})$$

- Relative Error is the difference between the exact and computed values, relative to the size of the exact value.
- Condition Number is a measure of sensitivity of the problem. It is independent of floating point arithmetic and the algorithm.
- $\epsilon_{machine}$ gives an upper bound on the relative error due to rounding of floating point numbers.

For example, for a problem with a condition number of $\sim 10^6$ (not unreasonably large), a stable algorithm implemented in double precision arithmetic, for which $\epsilon_{machine} \simeq 10^{-16}$, will produce a result with relative error no worse than about 10^{-10}.

No algorithm will, in general, be capable of producing a result with error significantly smaller than this. An unstable algorithm would likely produce a much larger relative error.

We stress: Roundoff errors accumulate for all the algorithms and subroutines considered in these lectures, even though the algorithms are stable. It is fundamentally impossible to eliminate this accumulation since it depends on the sensitivity of the problem (the condition number). For stable algorithms, roundoff errors do not accumulate any worse than expected for the difficulty of the problem at hand. It is often possible to calculate, or at least estimate, the condition number of a problem and hence it is possible to bound the relative error of the answer.

Special Matrices

Type	Commonly Denoted	Properties
	Square Matrices	
Identity:	I	$AI = IA = A$ for all A
Symmetric:	S	$S^T = S$ or $s_{ij} = s_{ji}$ for all i, j
Hermitian:		$A^H = A$ or $a_{ij} = \bar{a}_{ji}$ for all i, j
Symmetric positive definite:[1]		$A^T = A$ and $\langle x, Ax\rangle > 0$, for all $x \neq 0$.
Hermitian positive definite:[2]		$A^H = A$ and $\langle x, Ax\rangle > 0$, for all $x \neq 0$.
Orthogonal:	Q	$Q^T Q = Q Q^T = I$ or $Q^T = Q^{-1}$
Unitary:	U, Q	$U^* U = U U^* = I$ or $U^* = U^{-1}$
	General Matrices	
Diagonal:		$a_{ij} = 0$ for $i \neq j$
Tridiagonal:		$a_{ij} = 0$ for $i > j+1$ and $i < j-1$
Upper Triangular:[3]	R, T, or U	$a_{ij} = 0$ for $i > j$
Lower Triangular:[4]	L	$a_{ij} = 0$ for $i < j$
Upper Hessenberg:	H	$a_{ij} = 0$ for $i > j+1$.
Banded:		$a_{ij} = 0$ for $i > j + kl$ and $i < j - ku$.

Block forms of the general matrices can also be defined.

[1] x is any non-zero real vector. Symmetric positive semi-definite if $\langle x, Ax\rangle \geq 0$, for all real $x \neq 0$. Likewise for symmetric negative definite and semi-definite with the equality reversed.
[2] x is any non-zero complex vector. Same terminology for semi-definite etc. as symmetric matrices.
[3] If the matrix is upper triangular and in addition has $a_{ii} = 0$, then the matrix is strictly upper triangular. If the matrix is upper triangular and in addition has $a_{ii} = 1$, then the matrix is unit upper triangular.
[4] If the matrix is lower triangular and in addition has $a_{ii} = 0$, then the matrix is strictly lower triangular. If the matrix is lower triangular and in addition has $a_{ii} = 1$, then the matrix is unit lower triangular.

Important Factorizations

Name	Property	Comments
LU:	$A = PLU$	A is square and non-singular
Cholesky:	$A = R^* R$	A is Hermitian positive definite
QR:	$A = QR$	A is $m \times n$ with $m \geq n$
Eigenvalue:	$A = V \Lambda V^{-1}$	A is square, non-defective
Schur:	$A = QTQ^*$	A is square
SVD:	$A = U \Sigma V^*$	A is $m \times n$

For A real: $* \to T$, Hermitian $\to$ symmetric, unitary $\to$ orthogonal.
In LU, P is a permutation matrix, L is unit lower triangular and U is upper triangular.
R is everywhere upper triangular.
Q is everywhere unitary (orthogonal if real).
In eigenvalue, Λ is a diagonal matrix of eigenvalues, V in an invertible matrix.
In Schur, T is upper triangular (quasi-upper triangular if A is real).
In SVD, U and V are unitary and Σ is a diagonal matrix of singular values .