Numerical Analysis

Lloyd N. Trefethen
Oxford University
March 2006

Acknowledgments

This article will appear in the forthcoming *Princeton Companion to Mathematics*, edited by Timothy Gowers with June Barrow-Green, to be published by Princeton University Press.

In preparing this essay I have benefitted from the advice of many colleagues, who corrected a number of errors of fact and emphasis. I have not always followed their advice, however, preferring, as one friend put it, to "put my head above the parapet". So I take full responsibility for errors and omissions here.

With thanks to: Aurelio Arranz, Carl de Boor, Folkmar Bornemann, Richard Brent, Sam Clark, Iain Duff, Don Estep, Janice Giudice, Gene Golub, Nick Gould, Tim Gowers, Anne Greenbaum, Leslie Greengard, Martin Gutknecht, Raphael Hauser, Des Higham, Nick Higham, Arieh Iserles, David Knezevic, Randy LeVeque, Bill Morton, Michael Overton, Beresford Parlett, Bill Phillips, Mike Powell, Alex Prideaux, Thomas Schmelzer, Hans Stetter, Gil Strang, Endre Süli, Daniel Szyld, and Andy Wathen.

1 The Need for Numerical Computation

Everyone knows that when scientists and engineers need numerical answers to mathematical problems, they turn to computers. Nevertheless, there is a widespread misconception about this process.

The power of numbers has been extraordinary. It is often noted that the scientific revolution was set in motion when Galileo and others made it a principle that everything must be measured. Numerical measurements led to physical laws expressed mathematically, and, in the remarkable cycle whose fruits are all around us, finer measurements led to refined laws, which in turn led to better technology and still finer measurements. The day has long since passed when an advance in the physical sciences could be achieved, or a significant engineering product developed, without numerical mathematics.

Computers certainly play a part in this story, yet there is a misunderstanding about what their role is. Many people imagine that scientists and mathematicians generate formulas, and then, by inserting numbers into these formulas, computers grind out the necessary results. The reality is nothing like this. What really goes on is a far more interesting process of execution of *algorithms*. In most cases the job could not be done even in principle by formulas, for most mathematical problems cannot be solved by a finite sequence of elementary operations. What happens instead is that fast algorithms quickly converge to "approximate" answers that are accurate to three or ten digits of precision, or a hundred. For a scientific or engineering application, such an answer may be as good as exact.

We can illustrate the complexities of exact versus approximate solutions by an elementary example. Suppose we have one polynomial of degree 4,

$$p(z) = c_0 + c_1 z + c_2 z^2 + c_3 z^3 + c_4 z^4,$$

and another of degree 5,

$$q(z) = d_0 + d_1 z + d_2 z^2 + d_3 z^3 + d_4 z^4 + d_5 z^5.$$

It is well known that there is an explicit formula expressing the roots of p in terms of radicals (discovered by Ferrari around 1540), but no such formula for the roots of q (as shown by Ruffini and ABEL more than 250 years later; see THE INSOLUBILITY OF THE QUINTIC for more details). Thus, in a certain philosophical sense the root-finding problems for p and q are utterly different. Yet in practice they hardly differ at all. If a scientist or a mathematician wants to know the roots of one of these polynomials, he or she will turn to a computer and get an answer to 16 digits of precision in less than a millisecond. Did the computer use an explicit formula? In the case of q, the answer is certainly no, but what about p? Maybe, maybe not. Most of the time, the user neither knows nor cares, and probably not one mathematician in a hundred could write down formulas for the roots of p from memory.

Here are three more examples of problems that can be solved in principle by a finite sequence of elementary operations, like rootfinding for p.

(1) Linear equations: solve a system of n linear equations in n unknowns.

(2) Linear programming: minimize a linear function of n variables subject to m linear constraints.

(3) Traveling salesman problem: find the shortest tour between n cities.

And here are five that, like rootfinding for q, cannot generally be solved in this manner.

(4) Find an eigenvalue of an $n \times n$ matrix.

(5) Minimize a function of several variables.

(6) Evaluate an integral.

(7) Solve an ordinary differential equation (ODE).

(8) Solve a partial differential equation (PDE).

Can we conclude that (1)–(3) will be easier than (4)–(8) in practice? Absolutely not. Problem (3) is usually very hard indeed if n is, say, in the hundreds or thousands. Problems (6) and (7) are usually rather easy, at least if the integral is in one dimension. Problems (1) and (4) are of almost exactly the same difficulty: easy when n is small, like 100, and often very hard when n is large, like 1 000 000. In fact, in these matters philosophy is such a poor guide to practice that, for each of the three problems (1)–(3), when n and m are large one often ignores the exact solution and uses approximate (but fast!) methods instead.

Numerical analysis is the study of algorithms for solving the problems of continuous mathematics, by which we mean problems involving real or complex variables. (This definition includes problems like linear programming and the traveling salesman problem posed over the real numbers, but not their discrete analogues.) In the remainder of this article we shall review some of its main branches, past accomplishments, and possible future trends.

2 A Brief History

Throughout the centuries, leading mathematicians have been involved with scientific applications, and in many cases this has led to the discovery of numerical algorithms still in use today. Gauss, as usual, is an outstanding example. Among many other contributions, he made crucial advances in least-squares data fitting (1795), systems of linear equations (1809), and numerical quadrature (1814), as well as inventing the fast Fourier transform (1805), though the last did not become widely known until its rediscovery by Cooley and Tukey in 1965.

Around 1900, the numerical side of mathematics started to become less conspicuous in the activities of research mathematicians. This was a consequence of the growth of mathematics generally and of great advances in fields in which, for technical reasons, mathematical rigor had to be the heart of the matter. For example, many advances of the early twentieth century sprang from mathematicians' new ability to reason rigorously about infinity, a subject relatively far from numerical calculation.

A generation passed, and in the 1940s computers were invented. From this moment numerical mathematics began to explode, but now mainly in the hands of specialists. New journals were founded such as *Mathematics of Computation* (1943) and *Numerische Mathematik* (1959). The revolution was sparked by hardware, but it included mathematical and algorithmic developments that had nothing to do with hardware. In the half-century from the 1950s, machines sped up by a factor of around 10^9, but so did the best algorithms known for some problems, generating a combined increase in speed of almost incomprehensible scale.

Half a century on, numerical analysis has grown into one of the largest branches of mathematics, the specialty of thousands of researchers who publish in dozens of mathematical journals as well as applications journals across the sciences and engineering. Thanks to the efforts of these people going back many decades, and thanks to ever more powerful computers, we have reached a point where most of the classical mathematical problems of the physical sciences can be solved numerically to high accuracy. Most of the algorithms that make this possible were invented since 1950.

Numerical analysis is built on a strong foundation: the mathematical subject of *approximation theory.* This field encompasses classical questions of interpolation, series expansions, and HARMONIC ANALYSIS associated with NEWTON, FOURIER, Gauss, and others; semiclassical problems of polynomial and rational minimax approximation associated with names such as CHEBYSHEV and Bernstein; and major newer topics including splines, radial basis functions, and WAVELETS. We shall not have space to address these subjects, but in almost every area of numerical analysis it is a fact that, sooner or later, the discussion comes down to approximation theory.

3 Machine Arithmetic and Rounding Errors

It is well known that computers cannot represent real or complex numbers exactly. A quotient like 1/7 evaluated on a computer, for example, will normally yield an inexact result. (It would be different if we designed machines to work in base 7!) Computers approximate real numbers by a system of *floating-point arithmetic*, in which each number is represented in a digital equivalent of scientific notation, so that the scale does not matter unless the number is so huge or tiny as to cause overflow or underflow. Floating-point arithmetic was invented by Konrad Zuse in Berlin in the 1930s, and by the end of the 1950s it was standard across the computer industry.

Until the 1980s, different computers had widely different arithmetic properties. Then, in 1985, after years of discussion, the IEEE (Institute of Electrical and Electronics Engineers) standard for binary floating-point arithmetic was adopted, or *IEEE arithmetic* for short. This standard has subsequently become nearly universal on processors of many kinds. An IEEE (double precision) real number consists of a 64-bit word divided into 53 bits for a signed fraction in base 2 and 11 bits for a signed exponent. Since $2^{-53} \approx 1.1 \times 10^{-16}$, IEEE numbers represent the numbers of the real line to a relative accuracy of about 16 digits. Since $2^{\pm 2^{10}} \approx 10^{\pm 308}$, this system works for numbers up to about 10^{308} and down to about 10^{-308}.

Computers do not merely represent numbers, of course; they perform operations on them such as addition, subtraction, multiplication, and division, and more complicated results are obtained from sequences of these elementary operations. In floating-point arithmetic, the computed result of each elementary operation is almost exactly correct in the following sense: if "$*$" is one of these four operations in its ideal form and "$\circledast$" is the same operation as realized on the computer, then for any floating-point numbers x and y, assuming that there is no underflow or overflow,

$$x \circledast y = (x * y)(1 + \varepsilon).$$

Here ε is a very small quantity, no greater in absolute value than a number known as *machine epsilon*, denoted by $\varepsilon_{\text{mach}}$, that measures the accuracy of the computer. In the IEEE system, $\varepsilon_{\text{mach}} = 2^{-53} \approx 1.1 \times 10^{-16}$.

Thus, on a computer, the interval $[1, 2]$, for example, is approximated by about 10^{16} numbers. It is interesting to compare the fineness of this discretization with that of the discretizations of physics. In a handful of solid or liquid or a balloonful of gas, the number of atoms or molecules in a line from one point to another is on the order of 10^8 (the cube root of Avogadro's number). Such a system behaves enough like a continuum to justify our definitions of physical quantities such as density, pressure, stress, strain, and temperature. Computer arithmetic, however, is more than a million times finer than this. Another comparison with physics concerns the precision to which fundamental constants are known, such as (roughly) 4 digits for the gravitational constant G, 7 digits for Planck's constant h and the elementary charge e, 8 digits for the speed of light c, and 12 digits for the ratio $\mu_{\text{e}}/\mu_{\text{B}}$ of the magnetic moment of the electron to the Bohr magneton. At present, almost nothing in physics is known to more than 12 digits of accuracy. Thus IEEE numbers are orders of magnitude more precise than any number in science. (Of course, purely mathematical quantities like π are another matter.)

In two senses, then, floating-point arithmetic is far closer to its ideal than is physics. It is a curious phenomenon that, nevertheless, it is floating-point arithmetic rather than the laws of physics that is widely regarded as an ugly and dangerous compromise. Numerical analysts themselves are partly to blame for this perception. In the 1950s and 1960s, the founding fathers of the field discovered that inexact arithmetic can be a source of danger, causing results to be in error that "ought" to be right. The source of such problems is *numerical instability*: that is, the amplification of rounding errors from microscopic to macroscopic scale by certain modes of computation. These men, including von Neumann, Wilkinson, Forsythe, and Henrici, took great pains to publicize the risks of careless reliance on machine arithmetic. These risks are very real, but the message was communicated all too successfully, leading to the current widespread impression that the main business of numerical analysis is coping with rounding errors. In fact, the main business of numerical analysis is designing algorithms that converge quickly; rounding-error analysis, while often a part of the discussion, is rarely the central issue. If rounding errors vanished, 90% of numerical analysis would remain.

4 Numerical Linear Algebra

Linear algebra became a standard topic in undergraduate mathematics curriculums in the 1950s and 1960s, and has remained there ever since. There are several reasons for this, but I think one is at the bottom of it: the importance of linear algebra has exploded since the arrival of computers.

The starting point of this subject is *Gaussian elimination*, a procedure that can solve n linear equations in n unknowns using on the order of n^3 arithmetic operations. Equivalently, it solves equations of the form $Ax = b$, where A is an $n \times n$ matrix and x and b are column vectors of size n. Gaussian elimination is invoked on computers around the world almost every time a system of linear equations is solved. Even if n is as large as 1000, the time required is less than one second on a typical 2006 desktop machine. The idea of elimination was first discovered by Chinese scholars about 2000 years ago, and more recent contributors include Lagrange, Gauss, and Jacobi. The modern way of describing such algorithms, however, was apparently introduced as late as 1944 by Dwyer. Suppose that, say, α times the first row of A is subtracted from the second row. This operation can be interpreted as the multiplication of A on the left by the lower-triangular matrix M_1 consisting of the identity with the additional nonzero entry $m_{21} = -\alpha$. Further analogous row operations correspond to further multiplications on the left by lower-triangular matrices M_j. If k steps convert A to an upper-triangular matrix U, then we have $MA = U$ with $M = M_k \cdots M_2 M_1$, or, upon setting $L = M^{-1}$,

$$A = LU.$$

Here L is unit lower-triangular, that is, lower-triangular with all its diagonal entries equal to 1. Since U represents the target structure and L encodes the operations carried out to get there, we can say that Gaussian elimination is a process of *lower-triangular upper-triangularization.*

Many other algorithms of numerical linear algebra are also based on writing a matrix as a product of matrices that have special properties. To borrow a phrase from biology, we may say that this field has a central dogma:

$$\text{algorithms} \longleftrightarrow \text{matrixfactorizations.}$$

In this framework we can quickly describe the next algorithm that needs to be considered. Not every matrix has an LU factorization; a 2×2 counterexample is the matrix

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

Soon after computers came into use it was observed that even for matrices that do have LU factorizations, the pure form of Gaussian elimination is unstable, amplifying rounding errors by potentially large amounts. Stability can be achieved by interchanging rows during the elimination in order to bring maximal entries to the diagonal—a process known as *pivoting.* Since pivoting acts on rows, it again corresponds to a multiplication of A by other matrices on the left. The matrix factorization corresponding to Gaussian elimination with pivoting is

$$PA = LU,$$

where U is upper-triangular, L is unit lower-triangular, and P is a permutation matrix, i.e., the identity matrix with permuted rows. If the permutations are chosen to bring the largest entry below the diagonal in column k to the (k, k) position before the kth elimination step, then L has the additional property $|\ell_{ij}| \leq 1$ for all i and j.

The discovery of pivoting came quickly, but its theoretical analysis has proved astonishingly hard. In practice, pivoting makes Gaussian elimination almost perfectly stable, and it is routinely done by almost all computer programs that need to solve linear systems of equations. Yet it was realized in around 1960 by Wilkinson and others that for certain exceptional matrices, Gaussian elimination is still unstable, even with pivoting. The lack of an explanation of this discrepancy represents an embarrassing gap at the heart of numerical analysis. Experiments suggest that the fraction of matrices for which Gaussian elimination amplifies rounding errors by a factor greater than $\rho n^{1/2}$ is in a certain sense exponentially small as a function of ρ as $\rho \to \infty$, where n is the dimension (for example, among random matrices with independent normally distributed entries), but a theorem to this effect has never been proved.

Meanwhile, beginning in the late 1950s, the field of numerical linear algebra expanded in another direction: the use of algorithms based on ORTHOGONAL or UNITARY matrices, that is, real matrices with $Q^{-1} = Q^{\mathrm{T}}$ or complex ones with $Q^{-1} = Q^{*}$, where Q^{*} denotes the conjugate transpose. The starting point of such developments is the idea of *QR factorization.* If A is an $m \times n$ matrix with $m \geq n$, a QR factorization of A is a product

$$A = QR,$$

where Q has orthonormal columns and R is upper-triangular. One can interpret this formula as a matrix expression of the familiar idea of *Gram–Schmidt orthogonalization*, in which the columns $q_1, q_2, \ldots$ of Q are determined one after another. These column operations correspond to multiplication of A on the right by elementary upper-triangular matrices. One could say that the Gram–Schmidt algorithm aims for Q and gets R as a by-product, and is thus a process of *triangular orthogonalization.* A big event was when Householder showed in 1958 that a dual strategy of *orthogonal triangularization* is more effective for many purposes. In this approach, by applying a succession of elementary matrix operations each of which reflects $\mathbb{R}^m$ across a hyperplane, one reduces A to upper-triangular form via orthogonal operations: one aims at R and gets Q as a by-product. The Householder method turns out to be more stable numerically, because orthogonal operations preserve norms and thus do not amplify the rounding errors introduced at each step.

From the QR factorization sprang a rich collection of linear algebra algorithms in the 1960s. The QR factorization can be used by itself to solve least-squares problems and construct orthonormal bases. More remarkable is its use as a step in other algorithms. In particular, one of the central problems of numerical linear algebra is the determination of the EIGENVALUES AND EIGENVECTORS of a square matrix A. If A has a complete set of eigenvectors, then by forming a matrix X whose columns are these eigenvectors and a diagonal matrix D whose diagonal entries are the corresponding eigenvalues, we obtain

$$AX = XD,$$

and hence, since X is nonsingular,

$$A = XDX^{-1},$$

the *eigenvalue decomposition.* In the special case in which A is hermitian, a complete set of orthonormal eigenvectors always exists, giving

$$A = QDQ^{*},$$

where Q is unitary. The standard algorithm for computing these factorizations was developed in the early 1960s by Francis, Kublanovskaya, and Wilkinson: the *QR algorithm.* Because polynomials of degree 5 or more cannot be solved by a formula, we know that eigenvalues cannot generally be computed in closed form. The QR algorithm is therefore necessarily an iterative one, involving a sequence of QR factorizations that is in principle infinite. Nevertheless, its convergence is extraordinarily rapid. In the symmetric case, for a typical matrix A, the QR algorithm converges *cubically*, in the sense that at each step the number of correct digits in one of the eigenvalue–eigenvector pairs approximately triples.

The QR algorithm is one of the great triumphs of numerical analysis, and its impact through widely used software products has been enormous. Algorithms and analysis based on it led in the 1960s to computer codes in Algol and Fortran and later to the software library EISPACK ("Eigensystem Package") and its descendant LAPACK. The same methods have also been incorporated in general-purpose numerical libraries such as the NAG, IMSL, and *Numerical Recipes* collections, and in problem-solving environments such as MATLAB, Maple, and Mathematica. These developments have been so successful that the computation of matrix eigenvalues long ago became a "black box" operation for virtually every scientist, with nobody but a few specialists knowing the details of how it is done. A curious related story is that EISPACK's relative LINPACK for solving linear systems of equations took on an unexpected function: it became the original basis for the benchmarks that all computer manufacturers run to test the speed of their computers. If a supercomputer is lucky enough to make the TOP500 list, updated twice a year since 1993, it is because of its prowess in solving certain matrix problems $Ax = b$ of dimensions ranging from 100 into the millions.

The eigenvalue decomposition is familiar to all mathematicians, but the development of numerical linear algebra has also brought its younger cousin onto the scene: the *singular value decomposition* (SVD). The SVD was discovered by Beltrami, JORDAN, and SYLVESTER in the late nineteenth century, and made famous by Golub and other numerical analysts beginning in around 1965. If A is an $m \times n$ matrix with $m \geq n$, an SVD of A is a factorization

$$A = U\Sigma V^*,$$

where U is $m \times n$ with orthonormal columns, V is $n \times n$ and unitary, and Σ is diagonal with diagonal entries $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$. One could compute the SVD by relating it to the eigenvalue problems for AA^* and A^*A, but this proves numerically unstable; a better approach is to use a variant of the QR algorithm that does not square A. Computing the SVD is the standard route to determining the NORM $\|A\| = \sigma_1$ (here $\|\cdot\|$ is the HILBERT-SPACE or "2" norm), the norm of the inverse $\|A^{-1}\| = 1/\sigma_n$ in the case where A is square and nonsingular, or their product, known as the *condition number*,

$$\kappa(A) = \|A\|\,\|A^{-1}\| = \sigma_1/\sigma_n.$$

It is also a step in an extraordinary variety of further computational problems including rank-deficient least-squares, computation of ranges and nullspaces, determination of ranks, "total least-squares," low-rank approximation, and determination of angles between subspaces.

All the discussion above concerns "classical" numerical linear algebra, born in the period 1950–75. The ensuing quarter-century brought in a whole new set of tools: methods for large-scale problems based on *Krylov subspace iterations*. The idea of these iterations is as follows. Suppose a linear algebra problem is given that involves a matrix of large dimension, say $n \gg 1000$. The solution may be characterized as the vector $x \in \mathbb{R}^n$ that satisfies a certain variational property such as minimizing $\frac{1}{2}x^{\mathrm{T}}Ax - x^{\mathrm{T}}b$ (for solving $Ax = b$ if A is symmetric positive definite) or being a stationary point of $(x^{\mathrm{T}}Ax)/(x^{\mathrm{T}}x)$ (for solving $Ax = \lambda x$ if A is symmetric). Now if K_k is a k-dimensional subspace of $\mathbb{R}^n$ with $k \ll n$, then it may be possible to solve the same variational problem much more quickly in that subspace. The magical choice of K_k is a *Krylov subspace*

$$K_k(A, q) = \mathrm{span}(q, Aq, \ldots, A^{k-1}q)$$

for an initial vector q. For reasons that have fascinating connections with approximation theory, solutions in these subspaces often converge very rapidly to the exact solution in $\mathbb{R}^n$ as k increases, if the eigenvalues of A are favorably distributed. For example, it is often possible to solve a matrix problem involving 10^5 unknowns to 10-digit precision in just a few hundred iterations. The speedup as compared with the classical algorithms may be a factor of thousands.

Krylov subspace iterations originated with the conjugate gradient and Lanczos iterations published in 1952, but in those years computers were not powerful enough to solve problems of a large enough scale for the methods to be competitive. They took off in the 1970s with the work of Reid and Paige and especially van der Vorst and Meijerink, who made famous the idea of *preconditioning*. In preconditioning a system $Ax = b$, one replaces it by a mathematically equivalent system such as

$$MAx = Mb$$

for some nonsingular matrix M. If M is well chosen, the new problem involving MA may have favorably distributed eigenvalues and a Krylov subspace iteration may solve it quickly.

Since the 1970s, preconditioned matrix iterations have emerged as an indispensable tool of computational science. As one indication of their prominence we may note that in 2001, Thomson ISI announced that the most heavily cited article in all of mathematics in the 1990s was the 1989 paper by van der Vorst introducing Bi-CGStab, a generalization of conjugate gradients for nonsymmetric matrices.

Finally, we must mention the biggest unsolved problem in numerical analysis. Can an arbitrary $n \times n$ matrix A be inverted in $O(n^\alpha)$ operations for every $\alpha > 2$? (The problems of solving a system $Ax = b$ or computing a matrix product AB are equivalent.) Gaussian elimination has $\alpha = 3$, and the exponent shrinks as far as 2.376 for certain recursive (though impractical) algorithms published by Coppersmith and Winograd in 1990. Is there a "fast matrix inverse" in store for us?

5 Numerical Solution of Differential Equations

Long before much attention was paid to linear algebra, mathematicians developed numerical methods to solve problems of analysis. The problem of numerical integration or *quadrature* goes back to Gauss and NEWTON, and even to ARCHIMEDES. The classic quadrature formulas are derived from the idea of interpolating data at $n + 1$ points by a polynomial of degree n, then integrating the polynomial exactly. Equally spaced interpolation points give the *Newton–Cotes formulas*, which are useful for small degrees but diverge at a rate as high as 2^n as $n \to \infty$: the *Runge phenomenon*. If the points are chosen optimally, then the result is *Gauss quadrature*, which converges rapidly and is numerically stable. It turns out that these optimal points are roots of LEGENDRE POLYNOMIALS, which are clustered near the endpoints. Equally good for most purposes is *Clenshaw–Curtis* quadrature, where the interpolation points become $\cos(j\pi/n)$, $0 \le j \le n$. This quadrature method is also stable and rapidly convergent, and unlike Gauss quadrature can be executed in $O(n \log n)$ operations by the fast Fourier transform. The explanation of why clustered points are necessary for effective quadrature rules is related to the subject of potential theory.

Around 1850 another problem of analysis began to get attention: the solution of ODEs. The *Adams formulas* are based on polynomial interpolation in equally spaced points, which in practice typically number fewer than 10. These were the first of what are now called *multistep methods* for the numerical solution of ODEs. The idea here is that for an initial value problem $u' = f(t, u)$ with independent variable $t > 0$, we pick a small time step $\Delta t > 0$ and consider a finite set of time values

$$t_n = n\Delta t, \quad n \ge 0.$$

We then replace the ODE by an algebraic approximation that enables us to calculate a succession of approximate values

$$v^n \approx u(t_n), \quad n \ge 0.$$

(The superscript here is just a superscript, not a power.) The simplest such approximate formula, going back to Euler, is

$$v^{n+1} = v^n + \Delta t f(t_n, v^n),$$

or, using the abbreviation $f^n = f(t_n, v^n)$,

$$v^{n+1} = v^n + \Delta t f^n.$$

Both the ODE itself and its numerical approximation may involve one equation or many, in which case $u(t,x)$ and v^n become vectors of an appropriate dimension. The Adams formulas are higher-order generalizations of Euler's formula that are much more efficient at generating accurate solutions. For example, the fourth-order Adams–Bashforth formula is

$$v^{n+1} = v^n + \tfrac{1}{24}\Delta t(55f^n - 59f^{n-1} + 37f^{n-2} - 9f^{n-3}).$$

The term "fourth-order" reflects a new element in the numerical treatment of problems of analysis: the appearance of questions of convergence as $\Delta t \to 0$. The formula above is of fourth order in the sense that it will normally converge at the rate $O((\Delta t)^4)$. The orders employed in practice are most often in the range 3–6, enabling excellent accuracy for all kinds of computations, typically in the range of 3–10 digits, and higher-order formulas are occasionally used when still more accuracy is needed.

Most unfortunately, the habit in the numerical analysis literature is to speak not of the *convergence* of these magnificently efficient methods, but of their *error*, or more precisely their *discretization* or *truncation error* as distinct from rounding error. This ubiquitous language of error analysis is dismal in tone, but seems ineradicable.

At the turn of the twentieth century, the second great class of ODE algorithms known as *Runge–Kutta* or *one-step methods* was developed by Runge, Heun, and Kutta. For example, here are the formulas of the famous fourth-order Runge–Kutta method, which advance a numerical solution (again scalar or system) from time step t_n to t_{n+1} with the aid of four evaluations of the function f:

$$\begin{aligned} a &= \Delta t f(v^n, t_n), \\ b &= \Delta t f(v^n + \tfrac{1}{2}a, t_n + \tfrac{1}{2}\Delta t), \\ c &= \Delta t f(v^n + \tfrac{1}{2}b, t_n + \tfrac{1}{2}\Delta t), \\ d &= \Delta t f(v^n + c, t_n + \Delta t), \\ v^{n+1} &= v^n + \tfrac{1}{6}(a + 2b + 2c + d). \end{aligned}$$

Runge–Kutta methods tend to be easier to implement but sometimes harder to analyze than multistep formulas. For example, for any s, it is a trivial matter to derive the coefficients of the s-step Adams–Bashforth formula, which has order of accuracy $p = s$. For Runge–Kutta methods, by contrast, there is no simple relationship between the number of "stages" (i.e., function evaluations per step) and the attainable order of accuracy. The classical methods with $s = 1, 2, 3, 4$ were known to Kutta in 1901 and have order $p = s$, but it was not until 1963 that it was proved that $s = 6$ stages are required to achieve order $p = 5$. The analysis of such problems involves beautiful mathematics from graph theory and other areas, and a key figure in this area since the 1960s has been John Butcher. For orders $p = 6, 7, 8$ the minimal numbers of stages are $s = 7, 9, 11$, while for $p > 8$ exact minima are not known. Fortunately, these higher orders are rarely needed for practical purposes.

When computers began to be used to solve differential equations after World War II, a phenomenon of the greatest practical importance appeared: once again, *numerical instability.* As before, this phrase refers to the unbounded amplification of local errors by a computational process, but now the dominant local errors are usually those of discretization rather than rounding. Instability typically manifests itself as an oscillatory error in the computed solution that blows up exponentially as more numerical steps are taken. One mathematician concerned with this effect was Germund Dahlquist. Dahlquist saw that the phenomenon could be analyzed with great power and generality, and some people regard the appearance of his 1956 paper as one of the events marking the birth of modern numerical analysis. This landmark paper introduced what might be called the *fundamental theorem of numerical analysis*:

$$\text{consistency} + \text{stability} = \text{convergence}.$$

The theory is based on precise definitions of these three notions along the following lines. *Consistency* is the property that the discrete formula has locally positive order of accuracy and thus models the right ODE. *Stability* is the property that errors introduced at one time step cannot grow unboundedly at later times. *Convergence* is the property that as $\Delta t \to 0$, in the absence of rounding errors, the numerical solution converges to the correct result. Before Dahlquist's paper, the idea of an equivalence of stability and convergence was perhaps in the air in the sense that practitioners realized that if a numerical scheme was not unstable, then it would probably give a good approximation to the right answer. His theory gave rigorous form to that idea for a wide class of numerical methods.

As computer methods for ODEs were being developed, the same was happening for the much bigger subject of PDEs. Discrete numerical methods for solving PDEs had been invented around 1910 by Richardson for applications in stress analysis and meteorology, and further developed by Southwell; in 1928 there was also a theoretical paper on finite-difference methods by Courant, Friedrichs, and Lewy. But although the Courant–Friedrichs–Lewy work later became famous, the impact of these ideas before computers came along was limited. After that point the subject developed quickly. Particularly influential in the early years were the group of researchers around von Neumann at the Los Alamos laboratory, including the young Peter Lax.

Just as for ODEs, von Neumann and his colleagues discovered that some numerical methods for PDEs were subject to catastrophic instabilities. For example, to solve the linear wave equation $u_t = u_x$ numerically we may pick space and time steps Δx and Δt for a regular grid,

$$x_j = j\Delta x, \quad t_n = n\Delta t, \quad j, n \geq 0,$$

and replace the PDE by algebraic formulas that compute a succession of approximate values:

$$v_j^n \approx u(t_n, x_j), \quad j, n \geq 0.$$

A well-known discretization for this purpose is the *Lax–Wendroff* formula:

$$v_j^{n+1} = v_j^n + \tfrac{1}{2}\lambda(v_{j+1}^n - v_{j-1}^n) + \tfrac{1}{2}\lambda^2(v_{j+1}^n - 2v_j^n + v_{j-1}^n),$$

where $\lambda = \Delta t/\Delta x$, which can be generalized to nonlinear systems of hyperbolic conservation laws in one dimension. For $u_t = u_x$, if λ is held fixed at a value less than or equal to 1, the method will converge to the correct solution as $\Delta x, \Delta t \to 0$ (ignoring rounding errors). If λ is greater than 1, on the other hand, it will explode. Von Neumann and others realized that the presence or absence of such instabilities could be tested, at least for linear constant-coefficient problems, by discrete FOURIER ANALYSIS in x: "von Neumann analysis." Experience indicated that, as a practical matter, a method would succeed if it was not unstable. A theory soon appeared that gave rigor to this observation: the *Lax equivalence theorem*, published by Lax and Richtmyer in 1956, the same year as Dahlquist's paper. Many details were different—this theory was restricted to linear equations whereas Dahlquist's theory for ODEs also applied to nonlinear ones—but broadly speaking the new result followed the same pattern of equating convergence to consistency plus stability. Mathematically, the key point was the uniform boundedness principle.

In the half-century since von Neumann died, the Lax–Wendroff formula and its relatives have grown into a breathtakingly powerful subject known as *computational fluid dynamics.* Early treatments of linear and nonlinear equations in one space dimension soon moved to two dimensions and eventually to three. It is now a routine matter to solve problems involving millions of variables on computational grids with hundreds of points in each of three directions. The equations are linear or nonlinear; the grids are uniform or nonuniform, often adaptively refined to give special attention to boundary layers and other fast-changing features; the applications are everywhere. Numerical methods were used first to model airfoils, then whole wings, then whole aircraft. Engineers still use wind tunnels, but they rely more on computations.

Many of these successes have been facilitated by another numerical technology for solving PDEs that emerged in the 1960s from diverse roots in engineering and mathematics: finite elements. Instead of approximating a differential operator by a difference quotient, finite-element methods approximate the solution itself by functions f that can be broken up into simple pieces. For instance, one might partition the domain of f into elementary sets such as triangles or tetrahedra and insist that the restriction of f to each piece is a polynomial of small degree. The solution is obtained by solving a variational form of the PDE within the corresponding finite-dimensional subspace, and there is often a guarantee that the computed solution is optimal within that subspace. Finite-element methods have taken advantage of tools of functional analysis to develop to a very mature state. These methods are known for their flexibility in handling complicated geometries, and in particular they are entirely dominant in applications in structural mechanics and civil engineering. The number of books and articles that have been published about finite-element methods is in excess of 10 000.

In the vast and mature field of numerical solution of PDEs, what aspect of the current state of the art would most surprise Richardson or Courant, Friedrichs and Lewy? I think it is the universal dependence on exotic algorithms of linear algebra. The solution of a large-scale PDE problem in three dimensions may require a system of a million equations to be solved at each time step. This may be achieved by a GMRES matrix iteration that utilizes a finite-difference preconditioner implemented by a Bi-CGStab iteration relying on another multigrid preconditioner. Such stacking of tools was surely not imagined by the early computer pioneers. The need for it ultimately traces to numerical instability, for as Crank and Nicolson first noted in 1947, the crucial tool for combating instability is the use of *implicit formulas*, which couple together unknowns at the new time step t_{n+1}, and it is in implementing this coupling that solutions of systems of equations are required.

Here are some examples illustrating the successful reliance of today's science and engineering on the numerical solution of PDEs: chemistry (Schrödinger equation); structural mechanics (equations of elasticity); weather prediction (geostrophic equations); turbine design (Navier–Stokes equations); acoustics (Helmholtz equation); telecommunications (Maxwell equations); cosmology (Einstein equations); oil discovery (migration equations); groundwater remediation (Darcy's law); integrated circuit design (drift diffusion equations); tsunami modelling (shallow-water equations); optical fibers (nonlinear wave equations); image enhancement (Perona–Malik equation); metallurgy (Cahn–Hilliard equation); pricing financial options (Black–Scholes equation).

6 Numerical Optimization

The third great branch of numerical analysis is optimization, that is, the minimization of functions of several variables and the closely related problem of solution of nonlinear systems of equations. The development of optimization has been somewhat independent of that of the rest of numerical analysis, carried forward in part by a community of scholars with close links to operations research and economics.

Calculus students learn that a smooth function may achieve an extremum at a point of zero derivative, or at a boundary. The same two possibilities characterize the two big strands of the field of optimization. At one end there are problems of finding interior zeros and minima of unconstrained nonlinear functions by methods related to multivariate calculus. At the other are problems of linear programming, where the function to be minimized is linear and therefore easy to understand, and all the challenge is in the boundary constraints.

Unconstrained nonlinear optimization is an old subject. Newton introduced the idea of approximating functions by the first few terms of what we now call their Taylor series; indeed, Arnol'd has argued that Taylor series were Newton's "main mathematical discovery." To find a zero x_* of a function F of a real variable x, everyone knows the idea of *Newton's method*: at the kth step, given an estimate $x^{(k)} \approx x_*$, use the derivative $F'(x^{(k)})$ to define a linear approximation from which to derive a better estimate $x^{(k+1)}$:

$$x^{(k+1)} = x^{(k)} - F(x^{(k)})/F'(x^{(k)}).$$

Newton (1669) and Raphson (1690) applied this idea to polynomials, and Simpson (1740) generalized it to other functions F and to systems of two equations. In today's language, for a system of n equations in n unknowns, we regard F as an n-vector whose derivative at a point $x^{(k)} \in \mathbb{R}^n$ is the $n \times n$ Jacobian matrix with entries

$$J_{ij}(x^{(k)}) = \frac{\partial F_i}{\partial x_j}(x^{(k)}), \quad 1 \leq i, j \leq n.$$

This matrix defines a linear approximation to $F(x)$ that is accurate for $x \approx x^{(k)}$. Newton's method then takes the matrix form

$$x^{(k+1)} = x^{(k)} - (J(x^{(k)}))^{-1} F(x^{(k)}),$$

which in practice means that to get $x^{(k)}$ from $x^{(k+1)}$, we solve a linear system of equations:

$$J(x^{(k)})(x^{(k+1)} - x^{(k)}) = -F(x^{(k)}).$$

As long as J is Lipschitz continuous and nonsingular at x_* and the initial guess is good enough, the convergence of this iteration is quadratic:

$$\|x^{(k+1)} - x_*\| = O(\|x^{(k)} - x_*\|^2). \tag{1}$$

Students often think it might be a good idea to develop formulas to enhance the exponent in this estimate to 3 or 4. However, this is an illusion. Taking two steps at a time of a quadratically convergent algorithm yields a quartically convergent one—so the difference in efficiency between quadratic and quartic is at best a constant factor. The same goes if the exponent 2, 3, or 4 is replaced by any other number greater than 1. The true distinction is between all of these algorithms that converge *superlinearly*, of which Newton's method is the prototype, and those that converge *linearly* or *geometrically*, where the exponent is just 1.

From the point of view of multivariate calculus, it is a small step from solving a system of equations to minimizing a scalar function f of a variable $x \in \mathbb{R}^n$: to find a (local) minimum, we seek a zero of the gradient $g(x) = \nabla f(x)$, an n-vector. The derivative of g is the Jacobian matrix known as the *Hessian* of f, with entries

$$H_{ij}(x^{(k)}) = \frac{\partial^2 f}{\partial x_i \partial x_j}(x^{(k)}), \quad 1 \leq i, j \leq n,$$

and one may utilize it just as before in a Newton iteration to find a zero of $g(x)$, the new feature being that a Hessian is always symmetric.

Though the Newton formulas for minimization and finding zeros were were already established, the arrival of computers created a new field of numerical optimization. One of the obstacles quickly encountered was that Newton's method often fails if the initial guess is not good. This problem has been comprehensively addressed both practically and theoretically by the algorithmic technologies known as *line searches* and *trust regions*.

For problems with more than a few variables, it also quickly became clear that the cost of evaluating Jacobians or Hessians at every step could be exorbitant. Faster methods were needed that might make use of inexact Jacobians or Hessians and/or inexact solutions of the associated linear equations, while still achieving superlinear convergence. An early breakthrough of this kind was the discovery of *quasi-Newton methods* in the 1960s by Broyden, Davidon, Fletcher, and Powell, in which partial information is used to generate steadily improving estimates of the true Jacobian or Hessian or its matrix factors. An illustration of the urgency of this subject at the time is the fact that in 1970 the optimal rank-two symmetric positive-definite quasi-Newton updating formula was published independently by no fewer than four different authors, namely Broyden, Fletcher, Goldfarb, and Shanno; their discovery has been known ever since as the *BFGS formula*. In subsequent years, as the scale of tractable problems has increased exponentially, new ideas have also become important, including *automatic differentiation*, a technology that enables derivatives of computed functions to be determined automatically: the computer program itself is "differentiated" so that as well as producing numerical outputs, it also produces their derivatives. The dream of automatic differentiation is an old one, but for various reasons, partly related to advances in sparse linear algebra, it did not become practical until the work of Bischof, Carle, and Griewank in the 1990s.

Unconstrained optimization problems are relatively easy, but they are not typical; the true depth of this field is revealed by the methods that have been developed for dealing with constraints. Suppose a function $f : \mathbb{R}^n \to \mathbb{R}$ is to be minimized subject to certain equality constraints $c_j(x) = 0$ and inequality constraints $d_j(x) \geq 0$, where $\{c_j\}$ and $\{d_j\}$ are also functions from $\mathbb{R}^n$ to $\mathbb{R}$. Even the problem of stating local optimality conditions for solutions to such problems is nontrivial, a matter involving LAGRANGE MULTIPLIERS and a distinction between active and inactive constraints. This problem was solved by what are now known as the *KKT conditions*, introduced by Kuhn and Tucker in 1951 and also 12 years earlier, it was subsequently realized, by Karush. Development of algorithms for constrained nonlinear optimization continues to be an active research topic today.

The problem of constraints brings us to the other strand of numerical optimization, linear programming. This subject was born in the 1930s and 1940s with Kantorovich in the Soviet Union and Dantzig in the United States. As an outgrowth of his work for the U.S. Air Force in the war, Dantzig invented the famous SIMPLEX ALGORITHM for solving linear programs in 1947. A linear program is nothing more than a problem of minimizing a linear function of n variables subject to m linear equality and/or inequality constraints. How can this be a challenge? One answer is that m and n may be large. Large-scale problems may arise through discretization of continuous problems and also in their own right. A famous early example was Leontiev's theory of input–output models in economics, which won him the Nobel Prize in 1973. Even in the 1970s, the Soviet Union used an input–output computer model involving thousands of variables as a tool for planning the economy.

The simplex algorithm made medium- and large-scale linear programming problems tractable. Such a problem is defined by its *objective function*, the function

$f(x)$ to be minimized, and its *feasible region*, the set of vectors $x \in \mathbb{R}^n$ that satisfy all the constraints. For a linear program the feasible region is a polyhedron, a closed domain bounded by hyperplanes, and the optimal value of f is guaranteed to be attained at one of the vertex points. (A point is called a *vertex* if it is the unique solution of some subset of the equations that define the constraints.) The simplex algorithm proceeds by moving systematically downhill from one vertex to another until an optimal point is reached. All of the iterates lie on the boundary of the feasible region.

In 1984, an upheaval occurred in this field, triggered by Narendra Karmarkar at AT&T Bell Laboratories. Karmarkar showed that one could sometimes do much better than the simplex algorithm by working in the interior of the feasible region instead. Improvements of Karmarker's method soon came along, especially a crucial idea of working with a pair of primal and dual formulations of a problem in tandem; one now speaks of *primal-dual interior-point methods.* These algorithms are very powerful, and they have the intriguing feature that they do not depend strongly on linearity. Thus, in addition to changing the field of linear programming, Karmarkar and his successors have had the healthy effect of bringing the linear and nonlinear sides of the field of optimization closer together. Today, this field is far advanced, dealing with linear or nonlinear problems involving millions of variables and constraints.

7 The Future

Numerical analysis sprang from mathematics; then it spawned the field of computer science. When universities began to found computer science departments in the 1960s, numerical analysts were often in the lead. Now, two generations later, most of them are to be found in mathematics departments. What happened? A part of the answer is that numerical analysts deal with continuous mathematical problems, whereas computer scientists prefer discrete ones, and it is remarkable how wide a gap this can be.

Nevertheless, the computer science side of numerical analysis is of crucial importance, and I would like to end with a prediction that emphasizes this aspect of the subject. Traditionally one might think of a numerical algorithm as a cut-and-dried procedure, a loop of some kind to be executed until a well-defined termination criterion is satisfied. For some computations this picture is accurate. On the other hand, beginning with the work of de Boor, Lyness, Rice and others in the 1960s, a less deterministic kind of numerical computing began to appear: *adaptive algorithms.* In an adaptive quadrature program of the simplest kind, two estimates of the integral are calculated on each portion of a certain mesh and then compared to produce an estimate of the local error. Based on this estimate, the mesh may then be refined locally to improve the accuracy. This process is carried out iteratively until a final answer is obtained that aims to be accurate to a tolerance specified in advance by the user. Most such computations come with no guarantee of accuracy, but an exciting ongoing development is the advance of more sophisticated techniques of *a posteriori* error control that sometimes do provide guarantees.

When combined with techniques of *interval arithmetic*, there is even the prospect of accuracy guaranteed with respect to rounding as well as discretization error.

First, computer programs for quadrature became adaptive; then programs for ODEs. For PDEs, the move to adaptive programs is happening on a longer timescale. More recently there have been related developments in the computation of Fourier transforms, optimization, and large-scale numerical linear algebra, and some of the new algorithms adapt to the computer architecture as well as the mathematical problem. In a world where several algorithms are known for solving every problem, we increasingly find that the most robust computer program will be one that has diverse capabilities at its disposal and deploys them adaptively on the fly. In other words, numerical computation is increasingly embedded in intelligent control loops. I believe this process will continue, just as has happened in so many other areas of technology, removing scientists further from the details of their computations but offering steadily growing power in exchange. I expect that most of the numerical computer programs of 2050 will be 99% intelligent "wrapper" and just 1% actual "algorithm," if such a distinction makes sense. Hardly anyone will know how they work, but they will be extraordinarily powerful and reliable, and will often deliver results of guaranteed accuracy.

This story will have a mathematical corollary. One of the fundamental distinctions in mathematics is between linear problems, which can be solved in one step, and nonlinear ones, which usually require iteration. A related distinction is between forward problems (one step) and inverse problems (iteration). As numerical algorithms are increasingly embedded in intelligent control loops, almost every problem will be handled by iteration, regardless of its philosophical status. Problems of algebra will be solved by methods of analysis; and between linear and nonlinear, or forward and inverse, the distinctions will fade.

Appendix: Some Major Numerical Algorithms

The list in Table 1 attempts to identify some of the most significant algorithmic (as opposed to theoretical) developments in the history of numerical analysis. In each case some of the key early figures are cited, more or less chronologically, and a key early date is given. Of course, any brief sketch of history like this must be an oversimplification. Distressing omissions of names occur throughout the list—including more than half of the authors of the EISPACK, LINPACK, and LAPACK libraries. Even the dates can be questioned; the fast Fourier transform is listed as 1965, for example, since that is the year of the paper that brought it to the world's attention, though Gauss made the same discovery 160 years earlier. Similarly, linear programming interior-point methods are listed with Karmarkar's breakthrough of 1984, though Khachiyan's important related work came five years earlier. Nor should one imagine that the years from 1991 to the present have been a blank! No doubt in the future we shall identify developments from this period that deserve a place in the table.

Table 1: Some algorithmic developments in the history of numerical analysis.

Year	Development	Key early figures
263	Gaussian elimination	Hui, Lagrange, Gauss, Jacobi
1671	Newton's method	Newton, Raphson, Simpson
1795	Least-squares fitting	Gauss, Legendre
1814	Gauss quadrature	Gauss, Jacobi
1855	Adams ODE formulas	Adams, Bashforth
1895	Runge–Kutta ODE formulas	Runge, Heun, Kutta
1910	Finite differences for PDE	Richardson, Southwell, Courant, von Neumann, Lax
1936	Floating-point arithmetic	Zuse
1943	Finite elements for PDE	Courant, Feng, Argyris, Clough
1946	Splines	Schoenberg, de Casteljau, Bezier, de Boor
1947	Monte Carlo simulation	Ulam, von Neumann, Metropolis
1947	Simplex algorithm	Kantorovich, Dantzig
1952	Conjugate gradient iteration	Hestenes, Stiefel, Lanczos
1952	Stiff ODE solvers	Curtiss, Hirschfelder, Dahlquist, Gear
1954	Fortran	Backus
1958	Orthogonal linear algebra	Givens, Householder, Wilkinson, Golub
1959	Quasi-Newton iterations	Davidon, Fletcher, Powell, Broyden
1961	QR algorithm for eigenvalues	Rutishauser, Kublanovskaya, Francis, Wilkinson
1965	Fast Fourier transform	Gauss, Cooley, Tukey
1971	Spectral methods for PDE	Lanczos, Clenshaw, Orszag, Gottlieb
1971	Radial basis functions	Hardy, Askey, Duchon, Micchelli
1973	Multigrid iterations	Fedorenko, Brandt, Hackbusch
1976	EISPACK, LINPACK, LAPACK	Moler, Stewart, Smith, Dongarra, Demmel, Bai
1976	Nonsymmetric Krylov iterations	Vinsome, Saad, van der Vorst, Sorensen
1977	Preconditioned matrix iterations	van der Vorst, Meijerink
1977	MATLAB	Moler
1977	IEEE arithmetic	Kahan
1982	Wavelets	Morlet, Grossmann, Meyer, Daubechies
1984	LP interior-point methods	Khachiyan, Karmarkar, Megiddo
1987	Fast multipole method	Rokhlin, Greengard
1991	Automatic differentiation	Bischof, Carle, Griewank

Further Reading

Ciarlet, P. G. 1978. *The Finite Element Method for Elliptic Problems.* Amsterdam: North-Holland.

Golub, G. H., and C. F. Van Loan. 1996. *Matrix Computations*, 3rd edn. Baltimore, MD: Johns Hopkins University Press.

Hairer, E., S. P. Nørsett (for Volume I), and G. Wanner. 1993, 1996. *Solving Ordinary Differential Equations*, Volumes I and II. Springer.

Iserles, A. (ed.). 1992–. *Acta Numerica* (annual volumes). Cambridge University Press.

Nocedal, J., and S. J. Wright. 1999. *Numerical Optimization.* Springer.

Powell, M. J. D. 1981. *Approximation Theory and Methods.* Cambridge University Press.

Richtmyer, R. D., and K. W. Morton. 1967. *Difference Methods for Initial-Value Problems.* Wiley Interscience.