GUEST EDITORS' INTRODUCTION

the Top 10 Algorithms

In putting together this issue of *Computing in Science & Engineering*, we knew three things: it would be difficult to list just 10 algorithms; it would be fun to assemble the authors and read their papers; and, whatever we came up with in the end, it would be controversial. We tried to assemble the 10 algorithms with the greatest influence on the development and practice of science and engineering in the 20th century. Following is our list (here, the list is in chronological order; however, the articles appear in no particular order):

- Metropolis Algorithm for Monte Carlo
- Simplex Method for Linear Programming
- Krylov Subspace Iteration Methods
- The Decompositional Approach to Matrix Computations
- The Fortran Optimizing Compiler
- QR Algorithm for Computing Eigenvalues
- Quicksort Algorithm for Sorting
- Fast Fourier Transform
- Integer Relation Detection
- Fast Multipole Method

With each of these algorithms or approaches, there is a person or group receiving credit for inventing or discovering the method. Of course, the reality is that there is generally a culmination of ideas that leads to a method. In some cases, we chose authors who had a hand in developing the algorithm, and in other cases, the author is a leading authority.

1521-9615/00/$10.00 © 2000 IEEE

Jack Dongarra
University of Tennessee and Oak Ridge National Laboratory
Francis Sullivan
IDA Center for Computing Sciences

In this issue

Monte Carlo methods are powerful tools for evaluating the properties of complex, many-body systems, as well as nondeterministic processes. Isabel Beichl and Francis Sullivan describe the Metropolis Algorithm. We are often confronted with problems that have an enormous number of dimensions or a process that involves a path with many possible branch points, each of which is governed by some fundamental probability of occurence. The solutions are not exact in a rigorous way, because we randomly sample the problem. However, it is possible to achieve nearly exact results using a relatively small number of samples compared to the problem's dimensions. Indeed, Monte Carlo methods are the only practical choice for evaluating problems of high dimensions.

John Nash describes the Simplex method for solving linear programming problems. (The use of the word *programming* here really refers to scheduling or planning—and not in the way that we tell a computer what must be done.) The Simplex method relies on noticing that the objective function's maximum must occur on a corner of the space bounded by the constraints of the "feasible region."

Large-scale problems in engineering and science often require solution of sparse linear algebra problems, such as systems of equations. The importance of iterative algorithms in linear algebra stems from the simple fact that a direct approach will require $O(N^3)$ work. The Krylov subspace iteration methods have led to a major change in how users deal with large, sparse, nonsymmetric matrix problems. In this article, Henk van der Vorst describes the state of the art in terms of

methods for this problem.

Introducing the decompositional approach to matrix computations revolutionized the field. G.W. Stewart describes the history leading up to the decompositional approach and presents a brief tour of the six central decompositions that have evolved and are in use today in many areas of scientific computation.

David Padua argues that the Fortran I compiler, with its parsing, analysis, and code-optimization techniques, qualifies as one of the top 10 "algorithms." The article describes the language, compiler, and optimization techniques that the first compiler had.

The QR Algorithm for computing eigenvalues of a matrix has transformed the approach to computing the spectrum of a matrix. Beresford Parlett takes us through the history of early eigenvalue computations and the discovery of the family of algorithms referred to as the QR Algorithm.

Sorting is a central problem in many areas of computing so it is no surprise to see an approach to solving the problem as one of the top 10. Joseph JaJa describes Quicksort as one of the best practical sorting algorithm for general inputs. In addition, its complexity analysis and its structure have been a rich source of inspiration for developing general algorithm techniques for various applications.

Daniel Rockmore describes the FFT as an algorithm "the whole family can use." The FFT is perhaps the most ubiquitous algorithm in use today to analyze and manipulate digital or discrete data. The FFT takes the operation count for discrete Fourier transform from $O(N^2)$ to $O(N \log N)$.

Some recently discovered integer relation detection algorithms have become a centerpiece of the emerging discipline of "experimental mathematics"—the use of modern computer technology as an exploratory tool in mathematical research. David Bailey describes the integer relation problem: given n real numbers $x_1, \ldots, x_n$, find the n integers $a_1, \ldots, a_n$ (if they exist) such that $a_1x_1 + \ldots + a_nx_n = 0$. Originally, the algorithm was used to find the coefficients of the minimal integer polynomial an algebraic number satisfied. However, more recently, researchers have used them to discover unknown mathematical identities, as well as to identify some constants that arise in quantum field theory in terms of mathematical constants.

The Fast Multipole Algorithm was developed originally to calculate gravitational and electrostatic potentials. The method utilizes techniques to quickly compute and combine the pair-wise approximation in $O(N)$ operations. This has led to a significant reduction in the computational complexity from $O(N^2)$ to $O(N \log N)$ to $O(N)$ in certain important cases. John Board and Klaus Schulten describe the approach and its importance in the field.

Your thoughts?

We have had fun putting together this issue, and we assume that some of you will have strong feelings about our selection. Please let us know what you think.

Jack Dongarra is a professor of computer science in the Computer Science Department at the University of Tennessee and a scientist in the mathematical science section of Oak Ridge National Lab. He received his BS in mathematics from Chicago State University, his MS in computer science from the Illinois Institute of Technology, and his PhD in applied mathematics from the University of New Mexico. Contact him at dongarra@cs.utk.edu; www.cs.utk.edu/~dongarra.

Francis Sullivan's biography appears in his article on page 69.

THE JOY OF ALGORITHMS

Francis Sullivan, Associate Editor-in-Chief

THE THEME OF THIS FIRST-OF-THE-CENTURY ISSUE OF *COMPUTING IN SCIENCE & ENGINEERING* IS ALGORITHMS. IN FACT, WE WERE BOLD ENOUGH—AND PERHAPS FOOLISH ENOUGH—TO CALL THE 10 EXAMPLES WE'VE SELECTED "THE TOP 10 ALGORITHMS OF THE CENTURY."

Computational algorithms are probably as old as civilization. Sumerian cuneiform, one of the most ancient written records, consists partly of algorithm descriptions for reckoning in base 60. And I suppose we could claim that the Druid algorithm for estimating the start of summer is embodied in Stonehenge. (That's really hard hardware!)

Like so many other things that technology affects, algorithms have advanced in startling and unexpected ways in the 20th century—at least it looks that way to us now. The algorithms we chose for this issue have been essential for progress in communications, health care, manufacturing, economics, weather prediction, defense, and fundamental science. Conversely, progress in these areas has stimulated the search for ever-better algorithms. I recall one late-night bull session on the Maryland Shore when someone asked, "Who first ate a crab? After all, they don't look very appetizing." After the usual speculations about the observed behavior of sea gulls, someone gave what must be the right answer—namely, "A very hungry person first ate a crab."

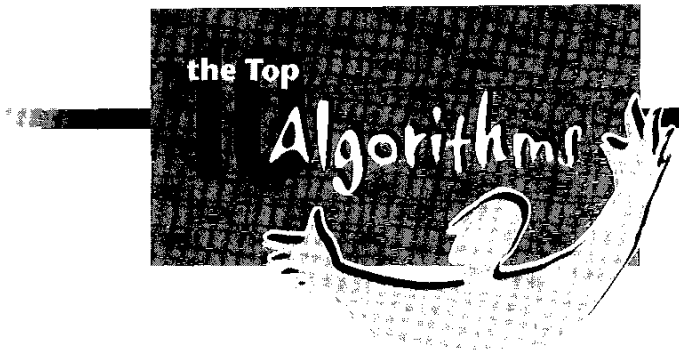
The flip side to "necessity is the mother of invention" is "invention creates its own necessity." Our need for powerful machines always exceeds their availability. Each significant computation brings insights that suggest the next, usually much larger, computation to be done. New algorithms are an attempt to bridge the gap between the demand for cycles and the available supply of them. We've become accustomed to gaining the Moore's Law factor of two every 18 months. In effect, Moore's Law changes the constant in front of the estimate of running time as a function of problem size. Important new algorithms do not come along every 1.5 years, but when they do, they can change the exponent of the complexity!

For me, great algorithms are the poetry of computation. Just like verse, they can be terse, allusive, dense, and even mysterious. But once unlocked, they cast a brilliant new light on some aspect of computing. A colleague recently claimed that he'd done only 15 minutes of productive work in his whole life. He wasn't joking, because he was referring to the 15 minutes during which he'd sketched out a fundamental optimization algorithm. He regarded the previous years of thought and investigation as a sunk cost that might or might not have paid off.

Researchers have cracked many hard problems since 1 January 1900, but we are passing some even harder ones on to the next century. In spite of a lot of good work, the question of how to extract information from extremely large masses of data is still almost untouched. There are still very big challenges coming from more "traditional" tasks, too. For example, we need efficient methods to tell when the result of a large floating-point calculation is likely to be correct. Think of the way that check sums function. The added computational cost is very small, but the added confidence in the answer is large. Is there an analog for things such as huge, multidisciplinary optimizations? At an even deeper level is the issue of reasonable methods for solving specific cases of "impossible" problems. Instances of NP-complete problems crop up in attempting to answer many practical questions. Are there efficient ways to attack them?

I suspect that in the 21st century, things will be ripe for another revolution in our understanding of the foundations of computational theory. Questions already arising from quantum computing and problems associated with the generation of random numbers seem to require that we somehow tie together theories of computing, logic, and the nature of the physical world.

The new century is not going to be very restful for us, but it is not going to be dull either!



THE METROPOLIS ALGORITHM

The Metropolis Algorithm has been the most successful and influential of all the members of the computational species that used to be called the "Monte Carlo Method." Today, topics related to this algorithm constitute an entire field of computational science supported by a deep theory and having applications ranging from physical simulations to the foundations of computational complexity.

The story goes that Stan Ulam was in a Los Angeles hospital recuperating and, to stave off boredom, he tried computing the probability of getting a "perfect" solitaire hand. Before long, he hit on the idea of using random sampling: Choose a solitaire hand at random. If it is perfect, let *count* = *count* + 1; if not, let *count* = *count*. After M samples, take *count*/M as the probability. The hard part, of course, is deciding how to generate a *uniform random* hand. What's the probability distribution to draw from, and what's the algorithm for drawing a hand?

Somewhat later, John von Neumann provided part of the answer, but in a different context. He introduced the *rejection algorithm* for simulating neutron transport. In brief, if you want to sample from some specific probability distribution, simply sample from any distribution you have handy, but keep only the good samples. (Von Neumann discussed his approach in a letter to Bob Richtmeyer [11 Mar. 1947] and in a later letter to Ulam [21 May 1947]. Interestingly, the letter to Richtmeyer contains a fairly detailed program for the Eniac, while the one to Ulam gives an explanation augmented by what we would today call pseudocode.)

Since the rejection method's invention, it has been developed extensively and applied in a wide variety of settings. The Metropolis Algorithm can be formulated as an instance of the rejection method used for generating steps in a Markov chain. This is the approach we will take.

1521-9615/00/$10.00 © 2000 IEEE

ISABEL BEICHL
NIST
FRANCIS SULLIVAN
IDA/Center for Computing Science

The rejection algorithm

First, let's look at the setup for the rejection algorithm itself. We want to sample from a population (for example, solitaire hands or neutron trajectories) according to some probability distribution function, ν, that is known in theory but hard to sample from in practice. However, we can easily sample from some related probability distribution function μ.

So we do this:

1. Use μ to select a sample, x.
2. Evaluate $\nu(x)$. This should be easy, once we have x.

3. Generate a uniform random $\rho \in [0,1)$
   if $\rho < cv(x)/\mu(x)$
   then accept x
   else try again with another x
   Here we choose c so that $cv(x)/\mu(x) < 1$ for all x.

First, the probability of selecting and then accepting some x is

$$c\frac{v(x)}{\mu(x)}\mu(x) = cv(x).$$

Also, if we are collecting samples to estimate the *weighted mean*, $S(f)$, of some function $f(x)$—that is, $S(f) = \Sigma f(x)cv(x)$—we could merely select some large number, M, of samples by using $\mu(x)$, reject none of them, and then compute the *uniform mean*:

$$\frac{1}{M}\sum f(x)c\frac{v(x)}{\mu(x)}.$$

That is, if we don't reject, the ratios give us a sample whose mean converges to the mean for the limiting probability distribution function v. This method for estimating a sum is an instance of *importance sampling*, because it attempts to choose samples according to their importance. The ideal importance function is

$$\mu(x) = \frac{f(x)v(x)}{\sum_y f(y)v(y)}.$$

The alert reader will have noticed that this $\mu(x)$ requires knowledge of the answer. However, importance sampling works for a less-than-perfect $\mu(x)$. This is because the fraction of the samples that equal any particular x will converge to $\mu(x)$, so the sample mean of $f(x)cv(x)/\mu(x)$ will converge to the true mean. For the special case of the constant function, $f(x) = 1$, the quantity S is the probability of a "success" on any particular trial of the rejection method. If we take f to be the function that is identically equal to one, we might know the value of S in advance. In that case, we also know the rejection rate $1/S$, which is the average number of trials before each success. As we shall see, when we use rejection in its formulation as the Metropolis Algorithm, prior knowledge of the rejection rate leads to a more efficient method called *Monte Carlo time*.[1]

Applications: The Metropolis Algorithm

We first look at two important applications of the Metropolis Algorithm—the Ising model and simulated annealing—and then we examine the problem of counting.

The Ising model

This model is one of the most extensively studied systems in statistical physics. It was developed early in the 20th century as a model of magnetization and related phenomena. The model is a 2D or 3D regular array of spins $\sigma_i \in \{-1, 1\}$ and an associated energy $E(\sigma)$ for each configuration. A configuration is any particular choice for the spins, and each configuration has the associated energy

$$E(\sigma) = -\sum_{i,j} \mathcal{J}_{i,j}\sigma_i\sigma_j - B\sum_k \sigma_k.$$

The sum is over those $\{i, j\}$ pairs that interact (usually nearest neighbors). $\mathcal{J}_{i,j}$ is the interaction coefficient (often constant), and B is another constant related to the external magnetic field.

In most applications, we want to estimate a mean of some function $f(\sigma)$ because such quantities give us a first-principles estimate of some fundamental physical quantity. In the Ising model, the mean $\mathcal{F}$ is taken over all configurations:

$$\mathcal{F} = \frac{1}{Z(T)}\sum_\sigma f(\sigma)\exp\left(-E(\sigma)/\kappa T\right).$$

But here, the weights come from the expression for the configuration's energy. The normalizing factor $Z(T)$ is the *partition function*:

$$Z(T) = \sum_\sigma \exp\left(-E(\sigma)/\kappa T\right).$$

T is the temperature and κ is the Boltzmann constant.

A natural importance-sampling approach might be to select configurations from the distribution:

$$\mu(\sigma) = \frac{\exp\left(-E(\sigma)/\kappa T\right)}{Z(T)}$$

so that the sample mean of M samples,

$$F = \frac{\sum_k f(\sigma_k)}{M}$$

will converge rapidly to the true mean, $\mathcal{F}$.

The problem, of course, is finding a way to sample from μ. In this case, sampling from the proposed "easy" distribution μ is not so simple. Nick Metropolis and his colleagues made the following brilliant observation.[2] If we change only one spin, the change in energy, ΔE, is easy to evaluate, because only a few terms of the sum change. This observation gives a way of constructing an

aperiodic symmetric Markov chain converging to the limit distribution μ. The transition probabilities, $p_{\xi,\sigma}$ are such that for each configuration, σ,

$$\mu(\sigma) = \sum_{\xi} \mu(\xi) p_{\xi,\sigma}.$$

The sum is over all configurations ξ that differ from σ by one spin, and

$$p_{\xi,\sigma} = \frac{\mu(\sigma)}{\mu(\xi)} = \exp\left(-\left(E(\sigma) - E(\xi)\right)/\kappa T\right)$$
$$= \exp\left(-\Delta E(\sigma)/\kappa T\right)$$

when $\Delta(E) > 0$, and $p_{\xi,\sigma} = 1$ when $\Delta(E) < 0$.

In other words, if the move lowers the energy, do it, and if it raises the energy, do it with some probability p, meaning reject it with some probability $1 - p$. But how to choose the site for the attempted move? We use rejection yet again. If there are n sites, we use a probability distribution that looks like

$$cv(\sigma) = \frac{\min\left(1, \exp\left(-\Delta E(\sigma)/\kappa T\right)\right)}{n}$$

so that we take $1/n$ as the "easy" probability. That is, we select a site uniformly and accept it according to the Metropolis criterion we just described.

For this case, the expression for the success rate is

$$S = \sum_i cv(\sigma_i) = \sum_i \frac{\min\left(1, \exp\left(-\Delta E(\sigma_i)/\kappa T\right)\right)}{n}.$$

So, the probability of exactly k rejections followed by a success is the same as the probability that a random ρ satisfies $(1 - S)^{k+1} < \rho < (1 - S)^k$, giving this stochastic expression for the waiting time:

$$k = \frac{\log(\rho)}{\log(1 - S)}.$$

We can use this to avoid the rejection steps while still "counting" how many rejections would have occurred.[1]

In principle, this Monte Carlo-time method works with any rejection formulation. However, each stage requires explicit knowledge of all possible next steps. In other words, we need the values for the "difficult" distribution $v(x)$. In the Metropolis Ising case, the Markov chain formulation makes this feasible.

Simulated annealing

Suppose we wish to maximize or minimize some real-valued function defined on a finite (but large) set. The classic example is the traveling salesman's problem. The function is the tour's length, and the set is that of possible tours. One approach is *hill climbing*. That is, given a set of possible changes to a tour, such as permuting the order of some visits, choose the change that decreases the tour length as much as possible. This approach's drawback is that it can get stuck at a local minimum, if all moves from a tour increase that tour's total length.

The Metropolis Algorithm offers a possible method for jumping out of a local minimum. Let the tour's length play the same role that energy plays in the Ising model, and assign a formal "temperature," T, to the system. Then, as long as $T > 0$, there is always a nonzero probability for increasing the tour length so that you needn't get trapped in local minima.

Three questions occur:

1. Does it work?
2. How long does it take?
3. Is it better than merely using hill climbing with many different random starts?

The answers seem to be

1. Yes. A large literature covers both the theory and applications in many different settings. However, if $T > 0$, the limit probability distribution will be nonzero for nonoptimal tours. The way around this is to decrease T as the computation proceeds. Usually, T decreases like $\log(s_k)$ for some positive, decreasing sequence of "cooling schedule" values s_k, so that the acceptance probability decreases linearly until only the true minimum is accepted.
2. It depends. Designing cooling schedules to optimize the solution time is an active research topic.
3. Someone should investigate this carefully.

Counting

Let's reconsider Ulam's original question in a slightly more general form: How many members of a population P have some specific property U? We could do the counting by designing a Markov chain that walks through P and has a limit probability distribution v that is somehow related to our interesting property U. To be more concrete, P might be the set of partial matchings of a bipartite graph G, and U might be the set of matchings that are "perfect," meaning they include every graph node.

To have our Markov chain do what we want, we define a partition function:

$$Z(\lambda) = \sum_k m_k \lambda^k .$$

The partition function is associated with the probability distribution

$$\nu(k) = \frac{m_k \lambda^k}{Z(\lambda)} .$$

Here, m_k is the number of k-matchings, and λ^k plays a role similar to that played by $\exp(-E(\sigma)/(\kappa T))$ in the Ising problem. On each step, if the move selected is from a k-matching to a $(k + 1)$-matching, the probability of doing so is λ. Mark Jerrum and Alistair Sinclair show that the fraction of the samples that are k-matchings can be used to estimate the m_k to whatever accuracy is desired and that, for fixed accuracy, the time for doing so is a polynomial in the problem's size.[3] Physicists call estimating the m_k the *monomer-dimer problem* because having a k-matching means that k pairs have been matched as dimers and the unmatched are monomers.

The limit distribution

The Metropolis Algorithm defines a convergent Markov chain whose limit is the desired probability distribution. But what is the convergence rate? Put differently, does a bound exist on the number of Metropolis steps, τ, required before the sample is close enough to the limit distribution? In some cases, τ can be bounded by a polynomial in the problem size; in other cases, we can show that no such bound exists.

Writers
For detailed information on submitting articles, write to cise@computer.org or visit computer.org/cise/edguide.htm.

Letters to the Editors
Jenny Ferrero, Lead Editor, jferrero@computer.org
Please provide an e-mail address or daytime phone number with your letter.

On the Web
Access computer.org/cise for information about *CiSE*.

Subscribe
Visit www.aip.org/cip/subscrib.htm or computer.org/subscribe.

Missing or Damaged Copies
If you are missing an issue or you received a damaged copy, contact membership@computer.org.

Reprints of Articles
For price information or to order reprints, send e-mail to cise@computer.org or fax +1 714 821 4010.

Reprint Permission
To obtain permission to reprint an article, contact William Hagen, IEEE Copyrights and Trademarks Manager, at whagen@ieee.org.

Rapid mixing

Jerrum and Sinclair have provided convergence results and applications to important combinatorial problems, such as the monomer-dimer problem.[3] To obtain their results, they look for a property they call *rapid mixing* for Markov chains. Jerrum has also proved some "slow convergence" results showing that, in some situations, Metropolis sampling does not mix rapidly and so converges too slowly to be practical.[4]

Coupling from the past

The Metropolis Algorithm and its generalizations have come to be known as the Monte Carlo Markov Chain technique (MCMC) because they simulate a Markov chain in the hope of sampling from the limit distribution. For the Ising model, this limit distribution is

$$\nu(\sigma) = \frac{\exp\left(-F(\sigma)/kT\right)}{Z(T)} .$$

The big question is, when are we at the limit distribution? That is, what is the convergence rate?

In some cases, we can sample directly from the limit distribution. Jim Propp and David Wilson developed a method for this called *coupling from the past* (CFTP).[5]

Think of a single Metropolis move as a map: $f_1 : S \to S$. For example, in the Ising model, choose some site k_1 and generate a random ρ_1 for the rejection test. Depending on the particular state σ, either $f_1(\sigma) = \sigma$ or $f_1(\sigma) = \sigma'$, where σ' differs from σ at one site.

Generally, the image of the set of all states does not cover all states—that is, $f_1[S] \subset S$. And, if we now choose a second map, f_2, we get $f_1 f_2[S] \subset f_1[S] \subset S$. Continuing in this way gives

$$F_k[S] = f_1 f_2 f_3 \ldots f_k[S] \subset f_1 f_2 f_3 \ldots f_{k-1}[S] \subset \ldots \subset S.$$

The functions are composed from the inside out; to add later maps, we must save earlier ones. Because the image is getting smaller, it might become a singleton; that is, F_k is constant. Propp and Wilson show that such a singleton will have been selected from the limit distribution. So, we have a method to sample from the true limit distribution, provided that we are willing to save all the maps and that we can tell when we have enough of them. One of several methods for telling when we have converged is to look for monotonicity. For some systems, there is an order, $<$, for states, there are bottom and top states $\{\perp < \top\}$, and the maps are order-preserving. In this case, we can

apply the iteration to both $\perp$ and T and wait for the two ends to meet. This works, for example, for some instances of the Ising model.

This method's obvious advantage is that, when such a sample can be obtained, it is "perfect." The disadvantages are that not all systems are amenable to this approach and that, when it does apply, the waiting time can be long.

Progress in MCMC has been impressive and seems to be accelerating. Problems that appeared impossible have been solved. For combinatorial counting problems, recent advances have been remarkable. However, two things should be borne in mind.

The first is a famous remark attributed to von Neumann: *Anyone using Monte Carlo is in a state of sin.* We might add that anyone using MCMC is committing an especially grievous offense. Monte Carlo is a last resort, to be used only when no exact analytic method or even finite numerical algorithm is available. And, except for CFTP, the prescription for use always contains the phrase "simulate for a while," meaning until you feel as if you're at the limit distribution. As we mentioned, for the Metropolis method, there are even systems for which convergence is provably slow. The antiferromagnetic Ising model is one such case. In some situations, *no* randomized method, including MCMC, will converge rapidly.

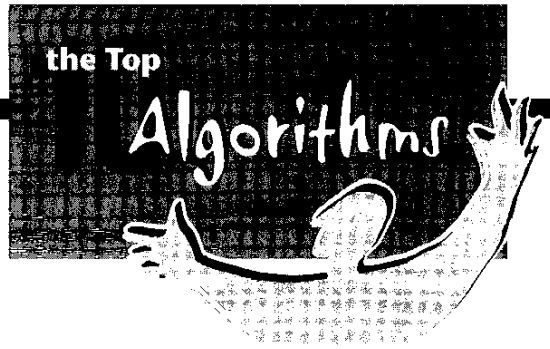
The second thing to bear in mind is that *MCMC is only one of many possible importance-sampling techniques.* For several cases, including the dimer cover problem, the ability to approximate the limit distribution directly results in extremely efficient and accurate importance-sampling methods that are quite different from MCMC.[6] However, a solid theory for these approaches is still almost nonexistent.

References

1. I. Beichl and F. Sullivan, "(Monte-Carlo) Time after Time," *IEEE Computational Science & Eng.*, Vol. 4, No. 3, July–Sept. 1997, pp. 91–95.
2. N. Metropolis et al., "Equation of State Calculations by Fast Computing Machines," *J. Chemical Physics*, Vol. 21, 1953, pp. 1087–1092.
3. M. Jerrum and A. Sinclair, "The Markov Chain Monte Carlo Method: An Approach to Counting and Integration," *Approximation Algorithms for NP-Hard Problems*, Dorit Hochbaum, ed., PWS (Brooks/Cole Publishing), Pacific Grove, Calif., 1996, pp. 482–520.
4. V. Gore and M. Jerrum, "The Swendson-Wang Process Does Not Always Mix Rapidly," *Proc. 29th ACM Symp. Theory of Computing*, ACM Press, New York, 1997, pp. 157–165.
5. J.G. Propp and D.B. Wilson, "Exact Sampling with Coupled Markov Chains and Applications to Statistical Mechanics," *Random Structures and Algorithms*, Vol. 9, Nos. 1 & 2, 1996, pp. 223–252.
6. I. Beichl and F. Sullivan, "Approximating the Permanent via Importance Sampling with Application to the Dimer Covering Problem," *J. Computational Physics*, Vol. 149, No. 1, Feb. 1999, pp. 128–147.

Isabel Beichl is a mathematician in the Information Technology Laboratory at the National Institute of Standards and Technology. Contact her at NIST, Gaithersburg, MD 20899; isabel@cam.nist.gov.

Francis Sullivan is the associate editor-in-chief of *CiSE* and director of the Institute for Defense Analyses' Center for Computing Sciences. Contact him at the IDA/Center for Computing Sciences, Bowie, MD 20715; fran@super.org.



THE (DANTZIG) SIMPLEX METHOD FOR LINEAR PROGRAMMING

George Dantzig created a simplex algorithm to solve linear programs for planning and decision-making in large-scale enterprises. The algorithm's success led to a vast array of specializations and generalizations that have dominated practical operations research for half a century.

George Dantzig, in describing the "Origins of the Simplex Method,"[1] noted that it was the availability of early digital computers that supported and invited the development of LP models to solve real-world problems. He agreed that invention is sometimes the mother of necessity. Moreover, he commented that he initially rejected the simplex method because it seemed intuitively more attractive to pursue the objective function downhill—as in the currently popular interior-point methods—rather than search along the constraint set's edges. It is this latter approach that the simplex method uses, which should not be confused with the J.A. Nelder and R. Mead's function-minimization method,[2] also associated with the word simplex.

When Dantzig introduced his method in 1947, it was somewhat easier to sort out the details of the simplex method than to deal with the "where are we in the domain space" questions that are, in my opinion, the core of interior-point approaches. Easier does not mean simpler, however.

Chapter and verse

The LP problem is, in one of the simplex method's many forms,

$$\begin{aligned} &\text{Minimize (with respect to } \boldsymbol{x}) && \boldsymbol{c}'\boldsymbol{x} && \text{(1a)}\\ &\text{subject to} && \mathbf{A}\boldsymbol{x} = \boldsymbol{b} && \text{(1b)}\\ &\text{and} && \boldsymbol{x} \geq 0 && \text{(1c)} \end{aligned}$$

This is not always the problem we want to solve, though—we might want to have the matrix $\mathbf{A}$ and vector $\boldsymbol{b}$ partitioned row-wise so that

$$\mathbf{A} = \begin{matrix}(\mathrm{A1})\\(\mathrm{A2})\\(\mathrm{A3})\end{matrix} \qquad \boldsymbol{b} = \begin{matrix}(\boldsymbol{b}1)\\(\boldsymbol{b}2)\\(\boldsymbol{b}3)\end{matrix}$$

so that we can write an LP problem as

1521-9615/00/$10.00 © 2000 IEEE

JOHN C. NASH
University of Ottawa

$$\text{Minimize (with respect to } \boldsymbol{x}) \quad \boldsymbol{c}'\boldsymbol{x} \tag{2a}$$

$$\text{subject to} \quad \mathbf{A}1\ \boldsymbol{x} \le \boldsymbol{b}1 \tag{2b}$$
$$\mathbf{A}2\ \boldsymbol{x} = \boldsymbol{b}2 \tag{2c}$$
$$\mathbf{A}3\ \boldsymbol{x} \ge \boldsymbol{b}3 \tag{2d}$$
$$\boldsymbol{x} \ge 0 \tag{2e}$$

However, the use of slack and surplus variables (the $\boldsymbol{x}$ variables we use to augment our problem) can transform the problems from one to the other. In fact, we must add slacks to the left-hand side of Equation 2b and subtract them from Equation 2d to arrive at equalities. Moreover, the special non-negativity conditions in Equations 1c and 2e could be subsumed into the matrix equation or inequation structure, but the LP tradition calls for listing them separately. The number of rows in $\mathbf{A}$ apart from the non-negativity constraints we call m, and the number of variables (including the slacks and surpluses) we call n. Typically, $m \le n$.

The simplex method assumes we have an initial basic feasible solution $\boldsymbol{x}_{\text{init}}$, that is, a solution that is feasible and that has just m of the $\boldsymbol{x}$'s nonzero. The $m\boldsymbol{x}$ values multiply m columns of $\mathbf{A}$, which we call the basis. "Basic" definitely has a technical rather than a general meaning here. We assume this set of basis vectors (columns of $\mathbf{A}$) is linearly independent and that it has full rank. Therefore, the core of the simplex method is to exchange one of the columns of $\mathbf{A}$ that is in the basis for one that is not. This corresponds to increasing one of the nonbasic variables while keeping the constraints satisfied, a process that reduces some of the $\boldsymbol{x}$'s. We choose to increase the $\boldsymbol{x}$ that gives us the most decrease in the objective function, and we continue to increase it until one of the current basic variables goes to zero, which means we now have a new basis. Moreover, the new solution turns out to be a new vertex of the simplex that the constraints define, and it is also a neighboring vertex to that described by our starting basic feasible solution. We have exchanged one vertex of the simplex for one of its neighbors and done so in a way that moves the objective function toward the minimum.

This exchange, or pivoting, seems to be a topic especially devised to torture business school undergraduates. Nevertheless, it is theoretically and practically a relatively simple process, unless, of course, you have a computer without enough memory to hold everything all at once. Additionally, there are some nasty details we have to sort out:

- How do we get an initial feasible solution? Or, more importantly, how do we tell if there is no feasible solution?
- What happens if the constraints don't stop the objective from being reduced? That is, what if the solution is unbounded?
- What happens if several vertices give equal objective values?

These questions, and the memory-management issues, occupied many researchers and generated many journal and book pages over the years. For example, more than half the pages in S. Gass's classic text are devoted to such details;[3] the applications come so late in the book that students of slow lecturers must have wondered what all the fuss was about.

Researchers handled the questions mentioned earlier in a variety of ways. Clever management and programming tactics overcame the memory issue. In early years, storage was so slow that programmers spent considerable effort to arrange computations that would complete in time to read and write data as disks and drums presented the appropriate tracks and sectors. Such complications help to obscure the central ideas of the algorithms in a muddle of detail, mostly irrelevant now.

The initial feasible solution issue was addressed, perhaps surprisingly, by adding m more variables, called artificial variables, which form the initial basis, then minimizing an objective function that drives them all out of the basis.[4] If we cannot drive them out, we have an infeasible problem.

Unbounded solutions turn out to be rather simple to detect, but the "equal objective function" or degeneracy issue worried people a great deal, because they implied the possibility that the algorithm might not terminate. In practice, perturbation or rule-based methods can avoid the cycling.

Nice extras

Modern textbooks spend much less ink on the details.[5] Similarly, recent discussions[6] of the computational techniques have abandoned the tableaux and the jargon of earlier work[3,4] regarding the matrix notations of numerical linear algebra. What remains of great interest, both mathematically and computationally, is the duality feature of LP problems. That is, an LP written as

$$\text{(Primal) minimize } \boldsymbol{c}'\boldsymbol{x} \text{ such that } \mathbf{A}\boldsymbol{x} \ge \boldsymbol{b}, \boldsymbol{x} \ge 0$$

has a dual equivalent

(Dual) maximize $\boldsymbol{b}'\boldsymbol{y}$ such that $\mathbf{A}'\boldsymbol{y} \leq \boldsymbol{c}, \boldsymbol{y} \geq 0$.

Moreover, the optima, if they exist, have the same objective value. Duality also leads to useful interpretations of solutions in terms of prices or values when the LP problems under consideration have economic or related contexts.

Practical importance

The simplex method's importance really lies in the value of the LP applications, even when the LP model is only a crude approximation to the real world. In the 1940s, many organizations were very hungry for solutions to LP problems, even if they did not realize what LP was. Oil and chemical companies led the way, especially for optimizing product mix from multiple sources or multiple sites. Transportation companies and the military recognized quite early that transportation and logistics problems could be formulated as LPs. Large-scale agricultural economic problems were also early applications. Examples also became realized in production planning, staff and resource scheduling, and network or traffic flows. LP can even be applied to maintaining the confidentiality of government statistics.[7]

For example, the Diet Problem has as objective function elements ($\boldsymbol{c}$) the costs per unit of food ingredients. The right-hand side values ($\boldsymbol{b}$) are the required minimum amounts of each of a list of m nutrients, and the constraint coefficients $\mathbf{A}$ give the amount of each nutrient in a unit of each ingredient. Thus we want to have $\mathbf{A}\,\boldsymbol{x} \geq \boldsymbol{b}$ to satisfy the nutrient requirements (we could also put in upper limits of nutrients like vitamin A that can be toxic), but we also want the cheapest blend. Of course, the resulting solution might not be very tasty. It is simply nutritious, and by construction will be the cheapest such recipe. Perhaps this is the origin of institutional cafeteria food.

Although the mathematics and computational algorithms might be fascinating, my view is that the value of the applications is the first reason for the importance of the simplex method. Promotion to the "top 10" category still needs something else, and this is the particular efficiency of the simplex method in finding the best vertex of the simplex. Most constraint matrices in practical LP problems are quite sparse, and the number of iterations where we move from one vertex to another is generally very small relative to the number of variables $\boldsymbol{x}$. Even better, we can explore the sensitivity of the optimal solution to small changes in the constraint and objective very easily. This is not always the case for some other methods.

Did Dantzig realize this efficiency as he first coded the simplex method? My guess is that he did not, although it is clear from his reminiscences[1] that such an understanding was not long in coming. In looking back over half a century, I find it remarkable that so many workers could sort through the jungle of awkward, and to some extent unnecessary, details to see the underlying tool's value.

References

1. S.G. Nash, *A History of Scientific Computing*, ACM Press, New York, 1990, pp. 141–151.
2. J.A. Nelder and R. Mead, "A Simplex Method for Function Minimization," *Computer J.*, Vol. 7, 1965, pp. 308–313.
3. S. Gass, *Linear Programming*, 2nd ed., McGraw-Hill, New York, 1964.
4. G. Hadley, *Linear Programming*, Addison-Wesley, Reading, Mass., 1962.
5. S.G. Nash and A. Sofer, *Linear and Nonlinear Programming*, McGraw-Hill, New York, 1996.
6. J.L. Nazareth, *Computer Solution of Linear Programs*, Oxford Science Publications, New York, 1987.
7. G. Sande, "Automated Cell Suppression to Preserve Confidentiality of Business Statistics," *Statistical J. United Nations ECE*, Vol. 2, 1984, pp. 33–41.

John C. Nash is a professor at the University of Ottawa. He is the author or coauthor of three books on computation, and his research and writings cover a wide range of topics in computers, mathematics, forecasting, risk management, and information science. He has been a mathematics columnist for *Interface Age* and the scientific computing editor for *Byte* magazine. He obtained his BSc in chemistry at the University of Calgary and his DPhil in mathematics from Oxford. Contact him at the Faculty of Administration, Univ. of Ottawa, 136 J-J Lussier Private, Ottawa, Ontario, K1N 6N5, Canada; jcnash@uottawa.ca; macnash.admin.uottawa.ca.

KRYLOV SUBSPACE ITERATION

This survey article reviews the history and current importance of Krylov subspace iteration algorithms.

Since the early 1800s, researchers have considered iteration methods an attractive means for approximating the solutions of large linear systems. They make these solutions possible now that we can do realistic computer simulations. The classical iteration methods typically converge very slowly (and often not at all). Around 1950, researchers realized that these methods lead to solution sequences that span a subspace—the Krylov subspace. It was then evident how to identify much better approximate solutions, without much additional computational effort.

When simulating a continuous event, such as the flow of a fluid through a pipe or of air around an aircraft, researchers usually impose a grid over the area of interest and restrict the simulation to the computation of relevant parameters. An example is the pressure or velocity of the flow or temperature inside the gridpoints. Physical laws lead to approximate relationships between these parameters in neighboring gridpoints. Together with the prescribed behavior at the boundary gridpoints and with given sources, this leads eventually to very large linear systems of equations, $Ax = b$. The vector x is the unknown parameter values in the gridpoints, b is the given input, and the matrix A describes the relationships between parameters in the gridpoints. Because these relationships are often restricted to nearby gridpoints, most matrix elements are zero.

The model becomes more accurate when we refine the grid—that is, when the distance between gridpoints decreases. In a 3D simulation, this easily leads to large systems of equations. Even a few hundred gridpoints in each coordinate direction leads to systems with millions of unknowns. Many other problems also lead to large systems: electric-circuit simulation, magnetic-field computation, weather prediction, chemical processes, semiconductor-device simulation, nuclear-reactor safety problems, mechanical-structure stress, and so on.

The standard numerical-solution methods for these linear systems are based on clever implementations of Gaussian elimination. These

1521-9615/00/$10.00 © 2000 IEEE

HENK A. VAN DER VORST
Utrecht University

methods exploit the sparse linear-system structure as much as possible to avoid computations with zero elements and zero-element storage. But for large systems these methods are often too expensive, even on today's fastest supercomputers, except where A has a special structure. For many of the problems previously listed, we can mathematically show that the standard solution methods will not lead to solutions in any reasonable amount of time. So, researchers have long tried to iteratively approximate the solution x. We start with a good guess for the solution—for instance, by solving a much easier nearby (idealized) problem. We then attempt to improve this guess by reducing the error with a convenient, cheap approximation for A—an *iterative solution method.*

Unfortunately, defining suitable nearby linear systems is difficult—in the sense that each step in the iterative process is cheap, and most important, that the iteration converges sufficiently fast. Suppose that we approximate the $n \times n$ matrix A of the linear system $Ax = b$ by the simpler matrix K. Then, we can formulate the above sketched iteration process as follows: in step $i + 1$, solve the new approximation x_{i+1} for the solution x of $Ax = b$, from

$$Kx_{i+1} = Kx_i + b - Ax_i .$$

For arbitrary initial start x_0, this process's convergence requirement is that the largest eigenvalue, in modulus, of the matrix $I - K^{-1}A$ is less than 1. The smaller this eigenvalue is, the faster the convergence will be (if $K = A$, we have convergence in one step). For most matrices, this is practically impossible. For instance, for the discretized Poisson equation, the choice $K = diag(A)$ leads to a convergence rate $1 - O(h^2)$, where h is the distance between gridpoints. Even for the more modern incomplete LU decompositions, this convergence rate is the same, which predicts a very marginal improvement per iteration step. We get reasonable fast convergence only for strongly diagonally dominant matrices. In the mid 1950s, this led to the observation in Ewald Bodewig's textbook that iteration methods were not useful, except when A approaches a diagonal matrix.[1]

Faster iterative solvers

Despite the negative feelings about iterative solvers, researchers continued to design faster iterative methods.

The developments of modern and more successful method classes started at about the same time, interestingly, in a way not appreciated at the time. The first and truly iterative approach tried to identify a trend in the successive approximants and to extrapolate on the last iteration results. This led to the *successive overrelaxation methods*, in which an overrelaxation (or extrapolation) parameter steered the iteration process. For interesting classes of problems, such as convection-diffusion problems and the neutron-diffusion equation, this led to attractive computational methods that could compete with direct methods (maybe not so much in computing time, but certainly because of the minimal computer memory requirements). David Young[2,3] and Richard Varga[4] were important researchers who helped make these methods attractive. The SOR methods were intensively used by engineers until more successful methods gradually replaced them.

The first and truly iterative approach tried to identify a trend in the successive approximants.

The early computers had relatively small memories that made iterative methods still attractive, because you had to store only the nonzero matrix elements. Also, iterative solution, although slow, was the only way out for many PDE-related linear systems. Including iteration parameters to kill dominant factors in the iteration errors—as in SOR—made the solution of large systems possible.

Varga reports that by 1960, Laplacian-type systems of 20,000 could be solved as a daily routine on a Philco-20000 computer with 32,000 words of core storage.[4] This would have been impossible with a direct method on a similar computer. However, the iterative methods of that time required careful tuning. For example, for the Chebyshev accelerated iteration methods, you needed accurate guesses for the matrix's extremal eigenvalues. Also, for the overrelaxation methods, you needed an overrelaxation parameter that was estimated from the largest eigenvalue of some related iteration matrix.

Another iterative-method class that became popular in the mid 1950s was the Alternating Direction method, which attempted to solve discretized PDEs over grids in more dimensions by successively solving 1D problems in each coordinate direction. Iteration parameters steered this process. Varga's book, *Matrix Iterative Analysis*, gives a good overview of the state of the art in

Compute $r_0 = b - Ax_0$ for some initial guess x_0
for $i = 1,2, \ldots$
  Solve z_{i-1} from $Kz_{i-1} = r_{i-1}$
  $\rho_{i-1} = r_{i-1}^* z_{i-1}$
  if $i = 1$
    $p_1 = z_0$
  else
    $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$;
    $p_i = z_{i-1} + \beta_{i-1} p_{i-1}$
  endif
  $q_i = Ap_i$
  $\alpha_i = \rho_{i-1} / p_i^* q_i$
  $x_i = x_{i-1} + \alpha_i p_i$
  $r_i = r_{i-1} - \alpha_i q_i$
  check convergence; continue if necessary
end;

Figure 1. The conjugate gradient algorithm.

$r = b - Ax_0$, for a given initial guess x_0
for $j = 1, 2, \ldots$
  $\beta = \|r\|_2$, $v_1 = r/\beta$; $\hat{b} = \beta e_1$;
  for $i = 1, 2, \ldots, m$
    $w = Av_i$;
    for $k = 1, \ldots, i$
      $h_{k,i} = v_k^* w$; $w = w - h_{k,i} v_k$;
    $h_{i+1,i} = \|w\|_2$; $v_{i+1} = w/h_{i+1,i}$;
    for $k = 2, \ldots, i$
      $\mu = h_{k-1,i}$
      $h_{k-1,i} = c_{k-1}\mu + s_{k-1} h_{k,i}$
      $h_{k,i} = -s_{k-1}\mu + c_{k-1} h_{k,i}$
    $\delta = \sqrt{h_{i,i}^2 + h_{i+1,i}^2}$; $c_i = h_{i,i}/\delta$; $s_i = h_{i+1,i}/\delta$
    $r_{i,i} = c_i h_{i,i} + s_i h_{i+1,i}$
    $\hat{b}_{i+1} = -s_i \hat{b}_i$; $\hat{b}_i = c_i \hat{b}_i$
    $\rho = |\hat{b}_{i+1}| \left(= \|b - Ax^{(i-1)m+i)}\|_2\right)$
    if ρ is small enough **then**
      ($n_r = i$; goto *SOL*);
  $n_r = m$; $y_{n_r} = \hat{b}_{n_r} / h_{n_r,n_r}$
SOL: **for** $k = n_r - 1, \ldots, 1$
    $y_k = (\hat{b}_k - \sum_{i=k+1}^{n_r} h_{k,i} y_i) / h_{k,k}$
  $x = \sum_{i=1}^{n_r} y_i v_i$; **if** ρ small enough quit
  $r = b - Ax$

Figure 2: GMRES(m) of Saad and Schultz.

1960.[4] It even mentions a system with 108,000 degrees of freedom. Many other problems with a variation in matrix coefficients, such as electronics applications, could not be solved at that time.

Because of the nonrobustness of the early iterative solvers, research focused on more efficient direct solvers. Especially for software used by nonnumerical experts, the direct methods have the advantage of avoiding convergence problems or difficult decisions on iteration parameters. The main problem, however, is that for general PDE-related problems discretized over grids in 3D domains, optimal direct techniques scale $\mathcal{O}(n^{2.3})$ in floating-point operations, so they are of limited use for the larger, realistic 3D problems. The work per iteration for an iterative method is proportional to n, which shows that if you succeed in finding an iterative technique that converges in considerably fewer than n iterations, this technique is more efficient than a direct solver.

For many practical problems, researchers have achieved this goal, but through clever combinations of modern iteration methods with (incomplete) direct techniques: the ILU preconditioned Krylov subspace solvers. With proper ordering techniques and appropriate levels of incompleteness, researchers have realized iteration counts for convection-diffusion problems that are practically independent of the gridsize. This implies that for such problems, the required number of flops is proportional with n (admittedly with a fairly large proportionality constant). The other advantage of iterative methods is that they need modest amounts of computer storage. For many problems, modern direct methods can also be very modest, but this depends on the system's matrix structure.

The Krylov subspace solvers

Cornelius Lanczos[5] and Walter Arnoldi[6] also established the basis for very successful methods in the early 1950s. The idea was to keep all approximants computed so far in the iteration process and to recombine them to a better solution. This task might seem enormous, but Lanczos recognized that the basic iteration (for convenience we will take $K = I$) leads to approximants x_i that are in nicely structured subspaces. Namely, these subspaces are spanned by the vectors r_0, Ar_0, A^2r_0, ..., $A^{i-1}r_0$, where $r_0 = b - Ax_0$. Such a subspace is a Krylov subspace of dimension i for A and r_0.

Lanczos showed that you can generate an orthogonal basis for this subspace with a very simple three-term recurrence relation if the matrix A is symmetric. This simplified the optimal-solution computations in the Krylov subspace. The attractive aspect is that you can obtain these optimal solutions for approximately the same computational costs as the approximants for the original iterative process, which was initially not recognized as a breakthrough in iterative processes. The early observation was that, after $n - 1$ steps, this process must terminate because the Krylov subspace is of dimension n. For that reason, this Lanczos process was regarded as a direct-solution method. Researchers tested the method on tough (although low-dimensional) problems and soon observed that after $n - 1$ steps the approximant x_n could be quite far away from the solution x, with which it should coincide at that point. This made potential users suspicious.

Meanwhile, Magnus Hestenes and Eduard Stiefel[7] had proposed a very elegant method for symmetric positive definite systems, based on the same Krylov subspace principles: the conjugate gradient method. This method suffered from the same lack of exactness as Lanczos' method and did not receive much recognition in its first 20 years.

The conjugate gradient method

It took a few years for researchers to realize that it was more fruitful to consider the conjugate gradient method truly iterative. In 1972, John Reid was one of the first to point in this direction.[8] Meanwhile, analysis had already shown that a factor involving the ratio of the largest and smallest eigenvalue of A dictated this method's convergence and that the actual values of these eigenvalues play no role.

About the same time, researchers recognized that they could construct good approximations K for A with the property that the eigenvalues of $K^{-1}A$ were clustered around 1. This implied that the ratio of these eigenvalues was moderate and so led to fast convergence of conjugate gradients when applied to $K^{-1}Ax = K^{-1}b$ when K is also symmetric positive definite. This process is called *preconditioned conjugate gradients*. Figure 1 describes the algorithm, where $x * y$ denotes the innerproduct of two vectors x and y (complex conjugate if the system is complex).

David Kershaw was one of the first to experiment with the conjugate gradient method, with incomplete Cholesky factorization of A as a preconditioner for tough problems related to fusion problems.[9] Table 1 quotes iteration numbers for the basic Gauss-Seidel iteration (that is, the basic iteration for K the lower triangular part of A[4]) the accelerated version SOR (actually, a slightly faster variant, Block SOR[4]), and conjugate gradients preconditioned with incomplete Cholesky (also known as ICCG). The iteration numbers were necessary to reduce the initial-residual norm by a factor of 10^{-6}.

Table 1. Kershaw's results for a fusion problem.

Method	Number of iterations
Gauss Seidel	208,000
Block successive overrelaxation methods	765
Incomplete Cholesky conjugate gradients	25

Table 1 shows the sometimes gigantic improvements from the (preconditioned) conjugate gradients. These and other results also motivated the search for other powerful Krylov subspace methods for a more general equation system.

GMRES

Researchers have proposed quite a few specialized Krylov methods, including Bi-CG and QMR for unsymmetric A; MINRES and SYMMLQ for symmetric-indefinite systems; and Orthomin, Orthodir, and Orthores for general unsymmetric systems. The current de facto unsymmetric-system standard is the GMRES method, proposed in 1986 by Youcef Saad and Martin Schultz.[10] In this method, the x_i in the dimension i Krylov subspace is constructed for which the norm $\|b - Ax_i\|_2$ is minimal. This builds on an algorithm, proposed by Arnoldi,[6] that constructs an orthonormal basis for the Krylov subspace for unsymmetric A.

The price for this ideal situation is that you have to store a full orthogonal basis for the Krylov subspace, which means the more iterations, the more basis vectors you must store. Also, the work per iteration increases linearly, which makes the method attractive only if it converges really fast. For many practical problems, GMRES takes a few tens of iterations; for many other problems it can take hundreds, which makes a full GMRES unfeasable.

Figure 2 shows a GMRES version in which a restart occurs after every m iterations to limit the memory requirements and the work per iteration. The application for a preconditioned system $K^{-1}Ax = K^{-1}b$ is straightforward.

Compute $r_0 = b - Ax_0$ for some initial guess x_0
Choose $\hat{r}_0$, for example $\hat{r}_0 = r_0$
for $i = 1, 2, \ldots$
  $\rho_{i-1} = \hat{r}_0^* r_{i-1}$
  if $\rho_{i-1} = 0$ **method fails**
  if $i = 1$
    $p_i = r_{i-1}$
  else
    $\beta_{i-1} = (\rho_{i-1}/\rho_{i-2})(\alpha_{i-1}/\omega_{i-1})$
    $p_i = r_{i-1} + \beta_{i-1}(p_{i-1} - \omega_{i-1} v_{i-1})$
  endif
  Solve $\hat{p}$ from $K\hat{p} = p_i$
  $v_i = A\hat{p}$
  $\alpha_i = \rho_{i-1} / \hat{r}_0^* v_i$
  $s = r_{i-1} - \alpha_i v_i$
  if $\|s\|$ small enough **then**
    $x^{(i)} = x^{(i-1)} + \alpha_i \hat{p}$ and stop
  Solve z from $Kz = s$
  $t = Az$
  $\omega_i = s^* t / t^* t$
  $x_i = x_{i-1} + \alpha_i \hat{p} + \omega_i z$
  if x_i is accurate enough **then quit**
  $r_i = s - \omega_i t$
  for continuation it is necessary that $\omega_i \neq 0$
end

Figure 3. The Bi-CGSTAB algorithm.

In the mid 1980s, Peter Sonneveld recognized that you can use the A^T operation for a further residual reduction through a minor modification to the Bi-CG scheme, almost without additional computational costs. This CGS method was often faster but significantly more irregular, which led to a precison loss. In 1992, I showed that Bi-CG could be made faster and smoother, at almost no additional cost, with minimal residual steps.[11] Figure 3 schematically shows the resulting Bi-CGSTAB algorithm, for the solution of $Ax = b$ with preconditioner K.

It is difficult to make a general statement about how quickly these Krylov methods converge. Although they certainly converge much faster than the classical iteration schemes and convergence takes place for a much wider class of matrices, many practical systems still cannot be satisfactorily solved. Much depends on whether you are able to define a nearby matrix K that will serve as a preconditioner. Recent research is more oriented in that direction than in trying to improve the Krylov subspace methods, although we might see some improvements for these methods as well. Effective and efficient preconditioner construction is largely problem-dependent; a preconditioner is considered as effective if the number of iteration steps of the preconditioned Krylov subspace method is approximately 100 or less.

Bi-CGSTAB

The GMRES cost per iteration has also led to a search for cheaper near-optimal methods. Vance Faber and Thomas Manteuffel's famous result showed that constructing optimal solutions in the Krylov subspace for unsymmetric A by short recurrences, as in the conjugate gradients method, is generally not possible. The generalization of conjugate gradients for unsymmetric systems, Bi-CG, often displays an irregular convergence behavior, including a possible breakdown. Roland Freund and Noel Nachtigal gave an elegant remedy for both phenomena in their QMR method. BiCG and QMR have the disadvantage that they require an operation with A^T per iteration step. This additional operation does not lead to a further residual reduction.

In this contribution, I have highlighted some of the Krylov subspace methods that researchers have accepted as powerful tools for the iterative solution of very large linear systems with millions of unknowns. These methods are a breakthrough in iterative solution methods for linear systems. I have mentioned a few names that were most directly associated with the development of the most characteristic and powerful methods—CG, GMRES, and Bi-CGSTAB—but these only represent the tip of the iceberg in this lively research area. For more information, see the "Further reading" sidebar.

Another class of acceleration methods that has been developed since around 1980 are the multigrid or multilevel methods. These methods apply to grid-oriented problems, and the idea is to work with coarse and fine grids. Smooth solution components are largely determined on the

coarse grid; the fine grid is for the more locally varying components. When these methods work for regular problems over regular grids for PDEs, they can be very fast and are much more efficient than preconditioned Krylov solvers. However, there is no clear separation between the two camps: you can use multigrid as a preconditioner for Krylov methods for less regular problems and the Krylov techniques as smoothers for multigrid. This is a fruitful direction for further exploration.

Further reading

O. Axelsson, *Iterative Solution Methods*, Cambridge Univ. Press, Cambridge, UK, 1994.

R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, Soc. for Industrial and Applied Mathematics, Philadelphia, 1994.

G.H. Golub and C.F. Van Loan, *Matrix Computations*, Johns Hopkins Univ. Press, Baltimore, 1996.

A. Greenbaum, *Iterative Methods for Solving Linear System*, Soc. for Industrial and Applied Mathematics, Philadelphia, 1997.

G. Meurant, *Computer Solution of Large Linear Systems*, North-Holland, Amsterdam, 1999.

Y. Saad, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, Boston, 1996.

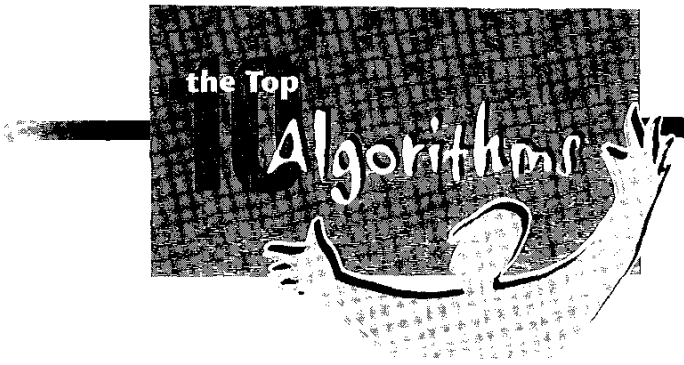
Y. Saad and H.A. van der Vorst, *Iterative Solution of Linear Systems in the 20th Century*, Tech. Report UMSI 99/152, Supercomputing Inst., Univ. of Minnesota, Minneapolis, Minn., 1999.

P. Wesseling, *An Introduction to Multigrid Methods*, John Wiley & Sons, Chichester, 1992.

References

1. E. Bodewig, *Matrix Calculus*, North-Holland, Amsterdam, 1956.
2. D. Young, *Iterative Solution of Large Linear Systems*, Academic Press, San Diego, 1971.
3. D.M. Young, *Iterative Methods for Solving Partial Differential Equations of Elliptic Type*, doctoral thesis, Harvard Univ., Cambridge, Mass., 1950.
4. R.S. Varga, *Matrix Iterative Analysis*, Prentice-Hall, Upper Saddle River, N.J., 1962.
5. C. Lanczos, "Solution of Systems of Linear Equations by Minimized Iterations," *J. Research Nat'l Bureau of Standards*, Vol. 49, 1952, pp. 33–53.
6. W.E. Arnoldi, "The Principle of Minimized Iteration in the Solution of the Matrix Eigenproblem," *Quarterly Applied Mathematics*, Vol. 9, 1951, pp. 17–29.
7. M.R. Hestenes and E. Stiefel, "Methods of Conjugate Gradients for Solving Linear Systems," *J. Research Nat'l Bureau of Standards*, Vol. 49, 1954, pp. 409–436.
8. J.K. Reid, "The Use of Conjugate Gradients for Systems of Equations Possessing 'Property A,'" *SIAM J. Numerical Analysis*, Vol. 9, 1972, pp. 325–332.
9. D.S. Kershaw, "The Incomplete Choleski-Conjugate Gradient Method for the Iterative Solution of Systems of Linear Equations," *J. Computational Physics*, Vol. 26, No. 1, 1978, pp. 43–65.
10. Y. Saad and M.H. Schultz, "GMRES: A Generalized Minimal Residual Algorithm for Solving Nnonsymmetric Linear Systems," *SIAM J. Scientific and Statistical Computing*, Vol. 7, No. 3, 1986, pp. 856–869.
11. H.A. van der Vorst, "Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems," *SIAM J. Scientific and Statistical Computing*, Vol. 13, No. 2, 1992, pp. 631–644.

Henk A. van der Vorst is a professor in numerical analysis at the Mathematical Department of Utrecht University. His research interests include iterative solvers for linear systems, large sparse eigenproblems, and the design of algorithms for parallel and vector computers. He is associate editor of the *SIAM Journal of Scientific Computing, Journal Computational and Applied Mathematics, Applied Numerical Mathematics, Parallel Computing, Numerical Linear Algebra with Applications,* and *Computer Methods in Applied Mechanics and Engineering.* He is a member of the Board of the Center for Mathematics and Informatics, Amsterdam. He has a PhD in applied mathematics from the University of Utrecht. Contact him at the Dept. of Mathematics, Univ. of Utrecht, PO Box 80.010, NL-3508 TA Utrecht, The Netherlands; vorst@math.uu.nl.



THE DECOMPOSITIONAL APPROACH TO MATRIX COMPUTATION

The introduction of matrix decomposition into numerical linear algebra revolutionized matrix computations. This article outlines the decompositional approach, comments on its history, and surveys the six most widely used decompositions.

In 1951, Paul S. Dwyer published *Linear Computations,* perhaps the first book devoted entirely to numerical linear algebra.[1] Digital computing was in its infancy, and Dwyer focused on computation with mechanical calculators. Nonetheless, the book was state of the art. Figure 1 reproduces a page of the book dealing with Gaussian elimination. In 1954, Alston S. Householder published *Principles of Numerical Analysis,*[2] one of the first modern treatments of high-speed digital computation. Figure 2 reproduces a page from this book, also dealing with Gaussian elimination.

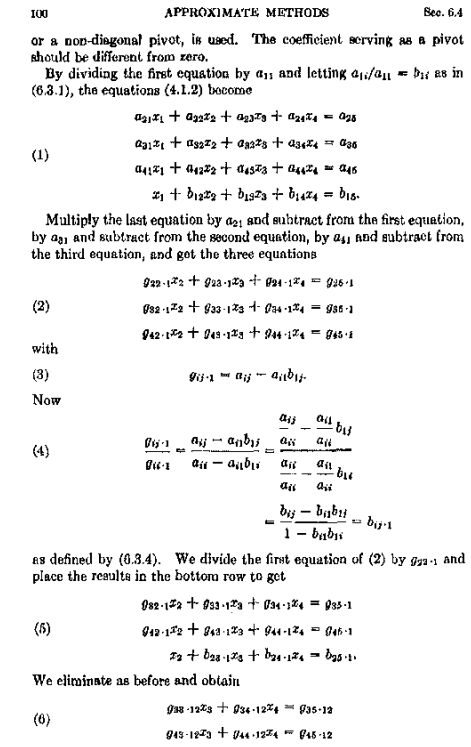
The contrast between these two pages is striking. The most obvious difference is that Dwyer used scalar equations whereas Householder used partitioned matrices. But a deeper difference is that while Dwyer started from a system of equations, Householder worked with a (block) LU decomposition—the factorization of a matrix into the product of lower and upper triangular matrices.

Generally speaking, a decomposition is a factorization of a matrix into simpler factors. The underlying principle of the decompositional approach to matrix computation is that it is not the business of the matrix algorithmists to solve particular problems but to construct computational platforms from which a variety of problems can be solved. This approach, which was in full swing by the mid-1960s, has revolutionized matrix computation.

To illustrate the nature of the decompositional approach and its consequences, I begin with a discussion of the Cholesky decomposition and the solution of linear systems—essentially the decomposition that Gauss computed in his elim-

1521-9615/00/$10.00 © 2000 IEEE

G.W. STEWART
University of Maryland



100 APPROXIMATE METHODS Sec. 6.4

or a non-diagonal pivot, is used. The coefficient serving as a pivot should be different from zero.

By dividing the first equation by a_{11} and letting $a_{1j}/a_{11} = b_{1j}$ as in (6.3.1), the equations (4.1.2) become

$$\begin{aligned} a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 &= a_{25} \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 + a_{34}x_4 &= a_{35} \\ a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4 &= a_{45} \\ x_1 + b_{12}x_2 + b_{13}x_3 + b_{14}x_4 &= b_{15}. \end{aligned} \tag{1}$$

Multiply the last equation by a_{21} and subtract from the first equation, by a_{31} and subtract from the second equation, by a_{41} and subtract from the third equation, and get the three equations

$$\begin{aligned} g_{22\cdot1}x_2 + g_{23\cdot1}x_3 + g_{24\cdot1}x_4 &= g_{25\cdot1} \\ g_{32\cdot1}x_2 + g_{33\cdot1}x_3 + g_{34\cdot1}x_4 &= g_{35\cdot1} \\ g_{42\cdot1}x_2 + g_{43\cdot1}x_3 + g_{44\cdot1}x_4 &= g_{45\cdot1} \end{aligned} \tag{2}$$

with

$$g_{ij\cdot1} = a_{ij} - a_{i1}b_{1j}. \tag{3}$$

Now

$$\frac{g_{ij\cdot1}}{g_{i4\cdot1}} = \frac{a_{ij} - a_{i1}b_{1j}}{a_{i4} - a_{i1}b_{14}} = \frac{\dfrac{a_{ij}}{a_{ii}} - \dfrac{a_{i1}}{a_{ii}}b_{1j}}{\dfrac{a_{i4}}{a_{ii}} - \dfrac{a_{i1}}{a_{ii}}b_{14}} = \frac{b_{ij} - b_{i1}b_{1j}}{1 - b_{i1}b_{14}} = b_{ij\cdot1} \tag{4}$$

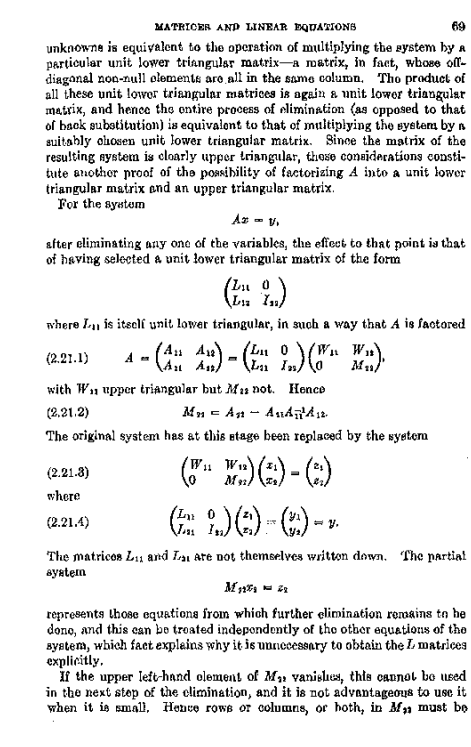
as defined by (6.3.4). We divide the first equation of (2) by $g_{22\cdot1}$ and place the results in the bottom row to get

$$\begin{aligned} g_{32\cdot1}x_2 + g_{33\cdot1}x_3 + g_{34\cdot1}x_4 &= g_{35\cdot1} \\ g_{42\cdot1}x_2 + g_{43\cdot1}x_3 + g_{44\cdot1}x_4 &= g_{45\cdot1} \\ x_2 + b_{23\cdot1}x_3 + b_{24\cdot1}x_4 &= b_{25\cdot1}. \end{aligned} \tag{5}$$

We eliminate as before and obtain

$$\begin{aligned} g_{33\cdot12}x_3 + g_{34\cdot12}x_4 &= g_{35\cdot12} \\ g_{43\cdot12}x_3 + g_{44\cdot12}x_4 &= g_{45\cdot12} \end{aligned} \tag{6}$$

Figure 1. This page from *Linear Computations* shows that Paul Dwyer's approach begins with a system of scalar equations. Courtesy of John Wiley & Sons.



MATRICES AND LINEAR EQUATIONS 69

unknowns is equivalent to the operation of multiplying the system by a particular unit lower triangular matrix—a matrix, in fact, whose off-diagonal non-null elements are all in the same column. The product of all these unit lower triangular matrices is again a unit lower triangular matrix, and hence the entire process of elimination (as opposed to that of back substitution) is equivalent to that of multiplying the system by a suitably chosen unit lower triangular matrix. Since the matrix of the resulting system is clearly upper triangular, these considerations constitute another proof of the possibility of factorizing A into a unit lower triangular matrix and an upper triangular matrix.

For the system

$$Ax = y,$$

after eliminating any one of the variables, the effect to that point is that of having selected a unit lower triangular matrix of the form

$$\begin{pmatrix} L_{11} & 0 \\ L_{12} & I_{22} \end{pmatrix}$$

where L_{11} is itself unit lower triangular, in such a way that A is factored

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 \\ L_{21} & I_{22} \end{pmatrix}\begin{pmatrix} W_{11} & W_{12} \\ 0 & M_{22} \end{pmatrix}, \tag{2.21.1}$$

with W_{11} upper triangular but M_{12} not. Hence

$$M_{22} = A_{22} - A_{21}A_{11}^{-1}A_{12}. \tag{2.21.2}$$

The original system has at this stage been replaced by the system

$$\begin{pmatrix} W_{11} & W_{12} \\ 0 & M_{22} \end{pmatrix}\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} \tag{2.21.3}$$

where

$$\begin{pmatrix} L_{11} & 0 \\ L_{21} & I_{22} \end{pmatrix}\begin{pmatrix} z_1 \\ z_2 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = y. \tag{2.21.4}$$

The matrices L_{11} and L_{21} are not themselves written down. The partial system

$$M_{22}z_2 = z_2$$

represents those equations from which further elimination remains to be done, and this can be treated independently of the other equations of the system, which fact explains why it is unnecessary to obtain the L matrices explicitly.

If the upper left-hand element of M_{22} vanishes, this cannot be used in the next step of the elimination, and it is not advantageous to use it when it is small. Hence rows or columns, or both, in M_{22} must be

Figure 2. On this page from *Principles of Numerical Analysis,* Alston Householder uses partitioned matrices and LU decomposition. Courtesy of McGraw-Hill.

ination algorithm. This article also provides a tour of the five other major matrix decompositions, including the pivoted LU decomposition, the QR decomposition, the spectral decomposition, the Schur decomposition, and the singular value decomposition.

A disclaimer is in order. This article deals primarily with dense matrix computations. Although the decompositional approach has greatly influenced iterative and direct methods for sparse matrices, the ways in which it has affected them are different from what I describe here.

The Cholesky decomposition and linear systems

We can use the decompositional approach to solve the system

$$Ax = b \tag{1}$$

where A is positive definite. It is well known that A can be factored in the form

$$A = R^{\mathrm{T}}R \tag{2}$$

where R is an upper triangular matrix. The factorization is called the *Cholesky decomposition* of A.

The factorization in Equation 2 can be used to solve linear systems as follows. If we write the system in the form $R^{\mathrm{T}}Rx = b$ and set $y = R^{-\mathrm{T}}b$, then x is the solution of the triangular system $Rx = y$. However, by definition y is the solution of the system $R^{\mathrm{T}}y = b$. Consequently, we have reduced the problem to the solution of two triangular systems, as illustrated in the following algorithm:

1. Solve the system $R^{\mathrm{T}}y = b$.
2. Solve the system $Rx = y$. (3)

Because triangular systems are easy to solve, the introduction of the Cholesky decomposition has

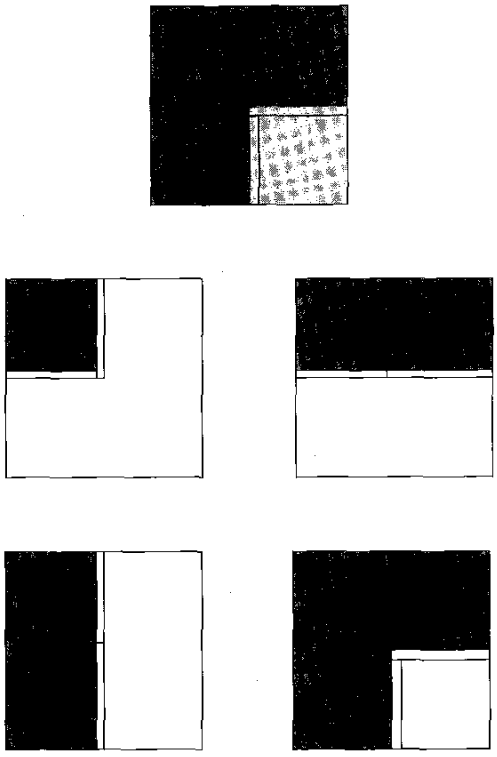


Figure 3. These varieties of Gaussian elimination are all numerically equivalent.

transformed our problem into one for which the solution can be readily computed.

We can use such decompositions to solve more than one problem. For example, the following algorithm solves the system $A^T x = b$:

1. Solve the system $Ry = b$.
2. Solve the system $R^T x = y$. (4)

Again, in many statistical applications we want to compute the quantity $\rho = x^T A^{-1} x$. Because

$$x^T A^{-1} x = x^T (R^T R)^{-1} x = (R^{-T} x)^T (R^{-T} x) \quad (5)$$

we can compute ρ as follows:

1. Solve the system $R^T y = x$.
2. $\rho = y^T y$. (6)

The decompositional approach can also save computation. For example, the Cholesky decomposition requires $O(n^3)$ operations to compute, whereas the solution of triangular systems requires only $O(n^2)$ operations. Thus, if you recognize that a Cholesky decomposition is being used to solve a system at one point in a computation, you can reuse the decomposition to do the same thing later without having to recompute it. Historically, Gaussian elimination and its variants (including Cholesky's algorithm) have solved the system in Equation 1 by reducing it to an equivalent triangular system. This mixes the computation of the decomposition with the solution of the first triangular system in Equation 3, and it is not obvious how to reuse the elimination when a new right-hand side presents itself. A naive programmer is in danger of performing the reduction from the beginning, thus repeating the lion's share of the work. On the other hand, a program that knows a decomposition is in the background can reuse it as needed.

(By the way, the problem of recomputing decompositions has not gone away. Some matrix packages hide the fact that they repeatedly compute a decomposition by providing drivers to solve linear systems with a call to a single routine. If the program calls the routine again with the same matrix, it recomputes the decomposition—unnecessarily. Interpretive matrix systems such as Matlab and Mathematica have the same problem—they hide decompositions behind operators and function calls. Such are the consequences of not stressing the decompositional approach to the consumers of matrix algorithms.)

Another advantage of working with decompositions is unity. There are different ways of organizing the operations involved in solving linear systems by Gaussian elimination in general and Cholesky's algorithm in particular. Figure 3 illustrates some of these arrangements: a white area contains elements from the original matrix, a dark area contains the factors, a light gray area contains partially processed elements, and the boundary strips contain elements about to be processed. Most of these variants were originally presented in scalar form as new algorithms. Once you recognize that a decomposition is involved, it is easy to see the essential unity of the various algorithms.

All the variants in Figure 3 are numerically equivalent. This means that one rounding-error analysis serves all. For example, the Cholesky algorithm, in whatever guise, is backward stable: the computed factor R satisfies

$$(A + E) = R^T R \quad (7)$$

where E is of the size of the rounding unit relative to A. Establishing this backward is usually the most difficult part of an analysis of the use

of a decomposition to solve a problem. For example, once Equation 7 has been established, the rounding errors involved in the solutions of the triangular systems in Equation 3 can be incorporated in E with relative ease. Thus, another advantage of the decompositional approach is that it concentrates the most difficult aspects of rounding-error analysis in one place.

In general, if you change the elements of a positive definite matrix, you must recompute its Cholesky decomposition from scratch. However, if the change is structured, it may be possible to compute the new decomposition directly from the old—a process known as *updating*. For example, you can compute the Cholesky decomposition of $A + xx^T$ from that of A in $O(n^2)$ operations, an enormous savings over the *ab initio* computation of the decomposition.

Finally, the decompositional approach has greatly affected the development of software for matrix computation. Instead of wasting energy developing code for a variety of specific applications, the producers of a matrix package can concentrate on the decompositions themselves, perhaps with a few auxiliary routines to handle the most important applications. This approach has informed the major public-domain packages: the Handbook series,[3] Eispack,[4] Linpack,[5] and Lapack.[6] A consequence of this emphasis on decompositions is that software developers have found that most algorithms have broad computational features in common—features than can be relegated to basic linear-algebra subprograms (such as Blas), which can then be optimized for specific machines.[7–9]

For easy reference, the sidebar "Benefits of the decompositional approach" summarizes the advantages of decomposition.

Benefits of the decompositional approach

- A matrix decomposition solves not one but many problems.
- A matrix decomposition, which is generally expensive to compute, can be reused to solve new problems involving the original matrix.
- The decompositional approach often shows that apparently different algorithms are actually computing the same object.
- The decompositional approach facilitates rounding-error analysis.
- Many matrix decompositions can be updated, sometimes with great savings in computation.
- By focusing on a few decompositions instead of a host of specific problems, software developers have been able to produce highly effective matrix packages.

History

All the widely used decompositions had made their appearance by 1909, when Schur introduced the decomposition that now bears his name. However, with the exception of the Schur decomposition, they were not cast in the language of matrices (in spite of the fact that matrices had been introduced in 1858[10]). I provide some historical background for the individual decompositions later, but it is instructive here to consider how the originators proceeded in the absence of matrices.

Gauss, who worked with positive definite systems defined by the normal equations for least squares, described his elimination procedure as the reduction of a quadratic form $\varphi(x) = x^T Ax$ (I am simplifying a little here). In terms of the Cholesky factorization $A = R^T R$, Gauss wrote $\varphi(x)$ in the form

$$\begin{aligned}\varphi(x) &= \left(r_1^T x\right)^2 + \left(r_2^T x\right)^2 + \cdots + \left(r_n^T x\right)^2 \\ &= \rho_1^2(x) + \rho_2^2(x) + \cdots + \rho_n^2(x)\end{aligned} \tag{8}$$

where r_i^T is the ith row of R. Thus Gauss reduced $\varphi(x)$ to a sum of squares of linear functions ρ_i. Because R is upper triangular, the function $\rho_i(x)$ depends only on the components $x_i, \ldots x_n$ of x. Since the coefficients in the linear forms ρ_i are the elements of R, Gauss, by showing how to compute the ρ_i, effectively computed the Cholesky decomposition of A.

Other decompositions were introduced in other ways. For example, Jacobi introduced the LU decomposition as a decomposition of a bilinear form into a sum of products of linear functions having an appropriate triangularity with respect to the variables. The singular value decomposition made its appearance as an orthogonal change of variables that diagonalized a bilinear form. Eventually, all these decompositions found expressions as factorizations of matrices.[11]

The process by which decomposition became so important to matrix computations was slow and incremental. Gauss certainly had the spirit. He used his decomposition to perform many tasks, such as computing variances, and even used it to update least-squares solutions. But Gauss never regarded his decomposition as a matrix factorization, and it would be anachro-

nistic to consider him the father of the decompositional approach.

In the 1940s, awareness grew that the usual algorithms for solving linear systems involved matrix factorization.[12,13] John Von Neumann and H.H. Goldstine, in their ground-breaking error analysis of the solution of linear systems, pointed out the division of labor between computing a factorization and solving the system:[14]

> We may therefore interpret the elimination method as one which bases the inverting of an arbitrary matrix A on the combination of two tricks: First it decomposes A into the product of two semi-diagonal matrices C, B' ..., and consequently the inverse of A obtains immediately from those of C and B'. Second it forms their inverses by a simple, explicit, inductive process.

In the 1950s and early 1960s, Householder systematically explored the relation between various algorithms in matrix terms. His book *The Theory of Matrices in Numerical Analysis* is the mathematical epitome of the decompositional approach.[15]

In 1954, Givens showed how to reduce a symmetric matrix A to tridiagonal form by orthogonal transformation.[16] The reduction was merely a way station to the computation of the eigenvalues of A, and at the time no one thought of it as a decomposition. However, it and other intermediate forms have proven useful in their own right and have become a staple of the decompositional approach.

In 1961, James Wilkinson gave the first backward rounding-error analysis of the solutions of linear systems.[17] Here, the division of labor is complete. He gives one analysis of the computation of the LU decomposition and another of the solution of triangular systems and then combines the two. Wilkinson continued analyzing various algorithms for computing decompositions, introducing uniform techniques for dealing with the transformations used in the computations. By the time his book *Algebraic Eigenvalue Problem*[18] appeared in 1965, the decompositional approach was firmly established.

The big six

There are many matrix decompositions, old and new, and the list of the latter seems to grow daily. Nonetheless, six decompositions hold the center. The reason is that they are useful and stable—they have important applications and the algorithms that compute them have a satisfactory backward rounding-error analysis (see Equation 7). In this brief tour, I provide references only for details that cannot be found in the many excellent texts and monographs on numerical linear algebra,[18–26] the Handbook series,[3] or the *LINPACK Users' Guide*.[5]

The Cholesky decomposition

Description. Given a positive definite matrix A, there is a unique upper triangular matrix R with positive diagonal elements such that

$$A = R^{\mathrm{T}}R.$$

In this form, the decomposition is known as the Cholesky decomposition. It is often written in the form

$$A = LDL^{\mathrm{T}}$$

where D is diagonal and L is unit lower triangular (that is, L is lower triangular with ones on the diagonal).

Applications. The Cholesky decomposition is used primarily to solve positive definite linear systems, as in Equations 3 and 6. It can also be employed to compute quantities useful in statistics, as in Equation 4.

Algorithms. A Cholesky decomposition can be computed using any of the variants of Gaussian elimination (see Figure 3)—modified, of course, to take advantage of symmetry. All these algorithms take approximately $n^3/6$ floating-point additions and multiplications. The algorithm Cholesky proposed corresponds to the diagram in the lower right of Figure 3.

Updating. Given a Cholesky decomposition $A = R^{\mathrm{T}}R$, you can calculate the Cholesky decomposition of $A + xx^{\mathrm{T}}$ from R and x in $O(n^2)$ floating-point additions and multiplications. The Cholesky decomposition of $A - xx^{\mathrm{T}}$ can be calculated with the same number of operations. The latter process, which is called *downdating*, is numerically less stable than updating.

The pivoted Cholesky decomposition. If P is a permutation matrix and A is positive definite, then $P^{\mathrm{T}}AP$ is said to be a diagonal permutation of A (among other things, it permutes the diagonals of A). Any diagonal permutation of A is positive definite and has a Cholesky factor. Such a factorization is called a *pivoted Cholesky factorization*. There are many ways to pivot a Cholesky decomposition, but the most common one produces a factor satisfying

$$r_{kk}^2 \geq \sum_{j>k} \sum_{i \geq k} r_{ij}^2 \tag{9}$$

In particular, if A is positive semidefinite, this strategy will assure that R has the form

$$R = \begin{pmatrix} R_{11} & R_{12} \\ 0 & 0 \end{pmatrix}$$

where R_{11} is nonsingular and has the same order as the rank of A. Hence, the pivoted Cholesky decomposition is widely used for rank determination.

History. The Cholesky decomposition (more precisely, an LDL^T decomposition) was the decomposition of Gauss's elimination algorithm, which he sketched in 1809[27] and presented in full in 1810.[28] Benoît published Cholesky's variant posthumously in 1924.[29]

The pivoted LU decomposition

Description. Given a matrix A of order n, there are permutations P and Q such that

$$P^T A Q = LU$$

where L is unit lower triangular and U is upper triangular. The matrices P and Q are not unique, and the process of selecting them is known as *pivoting*.

Applications. Like the Cholesky decomposition, the LU decomposition is used primarily for solving linear systems. However, since A is a general matrix, this application covers a wide range. For example, the LU decomposition is used to compute the steady-state vector of Markov chains and, with the inverse power method, to compute eigenvectors.

Algorithms. The basic algorithm for computing LU decompositions is a generalization of Gaussian elimination to nonsymmetric matrices. When and how this generalization arose is obscure (see Dwyer[1] for comments and references). Except for special matrices (such as positive definite and diagonally dominant matrices), the method requires some form of pivoting for stability. The most common form is partial pivoting, in which pivot elements are chosen from the column to be eliminated. This algorithm requires about $n^3/3$ additions and multiplications.

Certain contrived examples show that Gaussian elimination with partial pivoting can be unstable. Nonetheless, it works well for the overwhelming majority of real-life problems.[30,31] Why is an open question.

History. In establishing the existence of the LU decomposition, Jacobi showed[32] that under certain conditions a bilinear form $\varphi(x, y)$ can be written in the form

$$\varphi(x, y) = \rho_1(x)\sigma_1(y) + \rho_2(x)\sigma_2(y) + \ldots + \rho_n(x)\sigma_n(y)$$

where ρ_i and σ_i are linear functions that depend only on the last $(n - i + 1)$ components of their arguments. The coefficients of the functions are the elements of L and U.

The QR decomposition

Description. Let A be an $m \times n$ matrix with $m \geq n$. There is an orthogonal matrix Q such that

$$Q^T A = \begin{pmatrix} R \\ 0 \end{pmatrix}$$

where R is upper triangular with nonnegative diagonal elements (or positive diagonal elements if A is of rank n).

If we partition Q in the form

$$Q = (Q_A \; Q_\perp)$$

where Q_A has n columns, then we can write

$$A = Q_A R. \tag{10}$$

This is sometimes called the *QR factorization* of A.

Applications. When A is of rank n, the columns of Q_A form an orthonormal basis for the column space $\mathcal{R}(A)$ of A, and the columns of $Q_\perp$ form an orthonormal basis of the orthogonal complement of $\mathcal{R}(A)$. In particular, $Q_A Q_A^T$ is the orthogonal projection onto $\mathcal{R}(A)$. For this reason, the QR decomposition is widely used in applications with a geometric flavor, especially least squares.

Algorithms. There are two distinct classes of algorithms for computing the QR decomposition: Gram–Schmidt algorithms and orthogonal triangularization.

Gram–Schmidt algorithms proceed stepwise by orthogonalizing the kth columns of A against the first $(k - 1)$ columns of Q to get the kth column of Q. There are two forms of the Gram–Schmidt algorithm, the classical and the modified, and they both compute only the factorization in Equation 10. The classical Gram–Schmidt is unstable. The modified form can produce a matrix Q_A whose columns deviate from orthogonality. But the deviation is

bounded, and the computed factorization can be used in certain applications—notably computing least-squares solutions. If the orthogonalization step is repeated at each stage—a process known as *reorthogonalization*—both algorithms will produce a fully orthogonal factorization. When $n \gg m$, the algorithms without reorthogonalization require about mn^2 additions and multiplications.

The method of orthogonal triangularization proceeds by premultiplying A by certain simple orthogonal matrices until the elements below the diagonal are zero. The product of the orthogonal matrices is Q, and R is the upper triangular part of the reduced A. Again, there are two versions. The first reduces the matrix by Householder transformations. The method has the advantage that it represents the entire matrix Q in the same amount of memory that is required to hold A, a great savings when $n \gg p$. The second method reduces A by plane rotations. It is less efficient than the first method, but is better suited for matrices with structured patterns of nonzero elements.

Relation to the Cholesky decomposition. From Equation 10, it follows that

$$A^{\mathrm{T}}A = R^{\mathrm{T}}R. \tag{11}$$

In other words, the triangular factor of the QR decomposition of A is the triangular factor of the Cholesky decomposition of the cross-product matrix $A^{\mathrm{T}}A$. Consequently, many problems—particularly least-squares problems—can be solved using either a QR decomposition from a least-squares matrix or the Cholesky decomposition from the normal equation. The QR decomposition usually gives more accurate results, whereas the Cholesky decomposition is often faster.

Updating. Given a QR factorization of A, there are stable, efficient algorithms for recomputing the QR factorization after rows and columns have been added to or removed from A. In addition, the QR decomposition of the rank-one modification $A + xy^{\mathrm{T}}$ can be stably updated.

The pivoted QR decomposition. If P is a permutation matrix, then AP is a permutation of the columns of A, and $(AP)^{\mathrm{T}}(AP)$ is a diagonal permutation of $A^{\mathrm{T}}A$. In view of the relation of the QR and the Cholesky decompositions, it is not surprising that there is a pivoted QR factorization whose triangular factor R satisfies Equation 9. In particular, if A has rank k, then its pivoted QR factorization has the form

$$AP = (Q_1\, Q_2)\begin{pmatrix} R_{11} & R_{12} \\ 0 & 0 \end{pmatrix}.$$

It follows that either Q_1 or the first k columns of AP form a basis for the column space of A. Thus, the pivoted QR decomposition can be used to extract a set of linearly independent columns from A.

History. The QR factorization first appeared in a work by Erhard Schmidt on integral equations.[33] Specifically, Schmidt showed how to orthogonalize a sequence of functions by what is now known as the Gram–Schmidt algorithm. (Curiously, Laplace produced the basic formulas[34] but had no notion of orthogonality.) The name QR comes from the QR algorithm, named by Francis (see the history notes for the Schur algorithm, discussed later). Householder introduced Householder transformations to matrix computations and showed how they could be used to triangularize a general matrix.[35] Plane rotations were introduced by Givens,[16] who used them to reduce a symmetric matrix to tridiagonal form. Bogert and Burris appear to be the first to use them in orthogonal triangularization.[36] The first updating algorithm (adding a row) is due to Golub,[37] who also introduced the idea of pivoting.

The spectral decomposition

Description. Let A be a symmetric matrix of order n. There is an orthogonal matrix V such that

$$A = V\Lambda V^{\mathrm{T}}, \quad \Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n). \tag{12}$$

If v_i denotes the ith column of V, then $Av_i = \lambda_i v_i$. Thus (λ_i, v_i) is an eigenpair of A, and the *spectral decomposition* shown in Equation 12 exhibits the eigenvalues of A along with complete orthonormal system of eigenvectors.

Applications. The spectral decomposition finds applications wherever the eigensystem of a symmetric matrix is needed, which is to say in virtually all technical disciplines.

Algorithms. There are three classes of algorithms to compute the spectral decomposition: the QR algorithm, the divide-and-conquer algorithm, and Jacobi's algorithm. The first two require a preliminary reduction to tridiagonal form by orthogonal similarities. I discuss the QR algorithm in the next section on the Schur decomposition. The divide-and-conquer algo-

rithm[38,39] is comparatively recent and is usually faster than the QR algorithm when both eigenvalues and eigenvectors are desired; however, it is not suitable for problems in which the eigenvalues vary widely in magnitude. The Jacobi algorithm is much slower than the other two, but for positive definite matrices it may be more accurate.[40] All these algorithms require $O(n^3)$ operations.

Updating. The spectral decomposition can be updated. Unlike the Cholesky and QR decompositions, the algorithm does not result in a reduction in the order of the work—it still remains $O(n^3)$, although the order constant is lower.

History. The spectral decomposition dates back to an 1829 paper by Cauchy,[41] who introduced the eigenvectors as solutions of equations of the form $Ax = \lambda x$ and proved the orthogonality of eigenvectors belonging to distinct eigenvalues. In 1846, Jacobi[42] gave his famous algorithm for spectral decomposition, which iteratively reduces the matrix in question to diagonal form by a special type of plane rotations, now called Jacobi rotations. The reduction to tridiagonal form by plane rotations is due to Givens[16] and by Householder transformations to Householder.[43]

The Schur decomposition

Description. Let A be a matrix of order n. There is a unitary matrix U such that

$$A = UTU^{H}$$

where T is upper triangular and H means conjugate transpose. The diagonal elements of T are the eigenvalues of A, which, by appropriate choice of U, can be made to appear in any order. This decomposition is called a *Schur decomposition* of A.

A real matrix can have complex eigenvalues and hence a complex Schur form. By allowing T to have real 2×2 blocks on its diagonal that contain its complex eigenvalues, the entire decomposition can be made real. This is sometimes called a *real Schur form.*

Applications. An important use of the Schur form is as an intermediate form from which the eigenvalues and eigenvectors of a matrix can be computed. On the other hand, the Schur decomposition can often be used in place of a complete system of eigenpairs, which, in fact, may not exist. A good example is the solution of Sylvester's equation and its relatives.[44,45]

Algorithms. After a preliminary reduction to Hessenberg form, which is usually done with Householder transformations, the Schur from is computed using the QR algorithm.[46] Elsewhere in this issue, Beresford Parlett discusses the modern form of the algorithm. It is one of the most flexible algorithms in the repertoire, having variants for the spectral decomposition, the singular values decomposition, and the generalized eigenvalue problem.

History. Schur introduced his decomposition in 1909.[47] It was the only one of the big six to have been derived in terms of matrices. It was largely ignored until Francis's QR algorithm pushed it into the limelight.

The singular value decomposition

Description. Let A be an $m \times n$ matrix with $m \geq n$. There are orthogonal matrices U and V such that

$$U^{T}AV = \begin{pmatrix} \Sigma \\ 0 \end{pmatrix}$$

where

$$\Sigma = \mathrm{diag}(\sigma_1, \ldots, \sigma_n), \qquad \sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n \geq 0.$$

This decomposition is called the *singular value decomposition* of A. If U_A consists of the first n columns of U, we can write

$$A = U_A \Sigma V^{T} \tag{13}$$

which is sometimes called the *singular value factorization* of A.

The diagonal elements of σ are called the *singular values* of A. The corresponding columns of U and V are called *left and right singular vectors* of A.

Applications. Most of the applications of the QR decomposition can also be handled by the singular value decomposition. In addition, the singular value decomposition gives a basis for the row space of A and is more reliable in determining rank. It can also be used to compute optimal low-rank approximations and to compute angles between subspaces.

Relation to the spectral decomposition. The singular value factorization is related to the spectral decomposition in much the same way as the QR factorization is related to the Cholesky decomposition. Specifically, from Equation 13 it follows that

$$A^{T}A = V\Sigma^{2}V^{T}.$$

Thus the eigenvalues of the cross-product ma-

trix A^TA are the squares of the singular vectors of A and the eigenvectors of A^TA are right singular vectors of A.

Algorithms. As with the spectral decomposition, there are three classes of algorithms for computing the singular value decomposition: the QR algorithm, a divide-and-conquer algorithm, and a Jacobi-like algorithm. The first two require a reduction of A to bidiagonal form. The divide-and-conquer algorithm[49] is often faster than the QR algorithm, and the Jacobi algorithm is the slowest.

History. The singular value decomposition was introduced independently by Beltrami in 1873[50] and Jordan in 1874.[51] The reduction to bidiagonal form is due to Golub and Kahan,[52] as is the variant of the QR algorithm. The first Jacobi-like algorithm for computing the singular value decomposition was given by Kogbetliantz.[53]

The big six are not the only decompositions in use; in fact, there are many more. As mentioned earlier, certain intermediate forms—such as tridiagonal and Hessenberg forms—have come to be regarded as decompositions in their own right. Since the singular value decomposition is expensive to compute and not readily updated, rank-revealing alternatives have received considerable attention.[54,55] There are also generalizations of the singular value decomposition and the Schur decomposition for pairs of matrices.[56,57] All crystal balls become cloudy when they look to the future, but it seems safe to say that as long as new matrix problems arise, new decompositions will be devised to solve them.

Acknowledgment

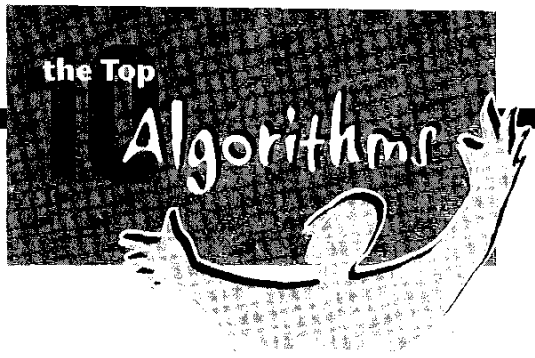
This work was supported by the National Science Foundation under Grant No. 970909-8562.

References

1. P.S. Dwyer, *Linear Computations*, John Wiley & Sons, New York, 1951.
2. A.S. Householder, *Principles of Numerical Analysis*, McGraw-Hill, New York, 1953.
3. J.H. Wilkinson and C. Reinsch, *Handbook for Automatic Computation, Vol. II, Linear Algebra*, Springer-Verlag, New York, 1971.
4. B.S. Garbow et al., "Matrix Eigensystem Routines—Eispack Guide Extension," *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1977.
5. J.J. Dongarra et al., *LINPACK User's Guide*, SIAM, Philadelphia, 1979.
6. E. Anderson et al., *LAPACK Users' Guide*, second ed., SIAM, Philadelphia, 1995.
7. J.J. Dongarra et al., "An Extended Set of Fortran Basic Linear Algebra Subprograms," *ACM Trans. Mathematical Software*, Vol. 14, 1988, pp. 1–17.
8. J.J. Dongarra et al., "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Trans. Mathematical Software*, Vol. 16, 1990, pp. 1–17.
9. C.L. Lawson et al., "Basic Linear Algebra Subprograms for Fortran Usage," *ACM Trans. Mathematical Software*, Vol. 5, 1979, pp. 308–323.
10. A. Cayley, "A Memoir on the Theory of Matrices," *Philosophical Trans. Royal Soc. of London*, Vol. 148, 1858, pp. 17–37.
11. C.C. Mac Duffee, *The Theory of Matrices*, Chelsea, New York, 1946.
12. P.S. Dwyer, "A Matrix Presentation of Least Squares and Correlation Theory with Matrix Justification of Improved Methods of Solution," *Annals Math. Statistics*, Vol. 15, 1944, pp. 82–89.
13. H. Jensen, *An Attempt at a Systematic Classification of Some Methods for the Solution of Normal Equations*, Report No. 18, Geodætisk Institut, Copenhagen, 1944.
14. J. von Neumann and H.H. Goldstine, "Numerical Inverting of Matrices of High Order," *Bull. Am. Math. Soc.*, Vol. 53, 1947, pp. 1021–1099.
15. A.S. Householder, *The Theory of Matrices in Numerical Analysis*, Dover, New York, 1964.
16. W. Givens, *Numerical Computation of the Characteristic Values of a Real Matrix*, Tech. Report 1574, Oak Ridge Nat'l Laboratory, Oak Ridge, Tenn., 1954.
17. J.H. Wilkinson, "Error Analysis of Direct Methods of Matrix Inversion," *J. ACM*, Vol. 8, 1961, pp. 281–330.
18. J.H. Wilkinson, *The Algebraic Eigenvalue Problem*, Clarendon Press, Oxford, U.K., 1965.
19. Å. Björck, *Numerical Methods for Least Squares Problems*, SIAM, Philadelphia, 1996.
20. B.N. Datta, *Numerical Linear Algebra and Applications*, Brooks/Cole, Pacific Grove, Calif., 1995.
21. J.W. Demmel, *Applied Numerical Linear Algebra*, SIAM, Philadelphia, 1997.
22. N.J. Higham, *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, 1996.
23. B.N. Parlett, *The Symmetric Eigenvalue Problem*, Prentice-Hall, Englewood Cliffs, N.J., 1980; reissued with revisions by SIAM, Philadelphia, 1998.
24. G.W. Stewart, *Introduction to Matrix Computations*, Academic Press, New York, 1973.
25. G.W. Stewart, *Matrix Algorithms I: Basic Decompositions*, SIAM, Philadelphia, 1998.
26. D.S. Watkins, *Fundamentals of Matrix Computations*, John Wiley & Sons, New York, 1991.
27. C.F. Gauss, *Theoria Motus Corporum Coelestium in Sectionibus Conicis Solem Ambientium [Theory of the Motion of the Heavenly Bodies Moving about the Sun in Conic Sections]*, Perthes and Besser, Hamburg, Germany, 1809.
28. C.F. Gauss, "Disquisitio de elementis ellipticis Palladis [Disquisition on the Elliptical Elements of Pallas]," *Commentatines societatis regiae scientarium Gottingensis recentiores*, Vol. 1, 1810.
29. C. Benoît, "Note sur une méthode de résolution des équations normales provenant de l'application de la méthode des moindres cárres a un systém d'équations linéares en nombre inférieur 'a celui des inconnues. — application de la méthode á la resolution d'un systéme défini d'équations linéares (Procédé du Commandant Cholesky) [Note on a Method for Solving the Normal Equations Arising from the Application of the Method of Least Squares to a System of Linear Equations whose Number Is Less than the Number of Unknowns—Application of the Method to the Solution of a Definite System of Linear Equations]," *Bullétin Géodésique (Toulouse)*, Vol. 2, 1924, pp. 5–77.

30. L.N. Trefethen and R.S. Schreiber, "Average-Case Stability of Gaussian Elimination," *SIAM J. Matrix Analysis and Applications*, Vol. 11, 1990, pp. 335–360.
31. L.N. Trefethen and D. Bau III, *Numerical Linear Algebra*, SIAM, Philadelphia, 1997, pp. 166–170.
32. C.G.J. Jacobi, "Über einen algebraischen Fundamentalsatz und seine Anwendungen [On a Fundamental Theorem in Algebra and Its Applications]," *Journal für die reine und angewandte Mathematik*, Vol. 53, 1857, pp. 275–280.
33. E. Schmidt, "Zur Theorie der linearen und nichtlinearen Integralgleichungen. I Teil, Entwicklung willkürlichen Funktionen nach System vorgeschriebener [On the Theory of Linear and Nonlinear Integral Equations. Part I, Expansion of Arbitrary Functions by a Prescribed System]," *Mathematische Annalen*, Vol. 63, 1907, pp. 433–476.
34. P.S. Laplace, *Theorie Analytique des Probabilites [Analytical Theory of Probabilities]*, 3rd ed., Courcier, Paris, 1820.
35. A.S. Householder, "Unitary Triangularization of a Nonsymmetric Matrix," *J. ACM*, Vol. 5, 1958, pp. 339–342.
36. D. Bogert and W.R. Burris, *Comparison of Least Squares Algorithms*, Tech. Report ORNL-3499, V.1, SS.5, Neutron Physics Div., Oak Ridge Nat'l Laboratory, Oak Ridge, Tenn., 1963.
37. G.H. Golub, "Numerical Methods for Solving Least Squares Problems," *Numerische Mathematik*, Vol. 7, 1965, pp. 206–216.
38. J.J.M. Cuppen, "A Divide and Conquer Method for the Symmetric Eigenproblem," *Numerische Mathematik*, Vol. 36, 1981, pp. 177–195.
39. M. Gu and S.C. Eisenstat, "Stable and Efficient Algorithm for the Rank-One Modification of the Symmetric Eigenproblem," *SIAM J. Matrix Analysis and Applications*, Vol. 15, 1994, pp. 1266–1276.
40. J. Demmel and K. Veselic, "Jacobi's Method Is More Accurate than QR," *SIAM J. Matrix Analysis and Applications*, Vol. 13, 1992, pp. 1204–1245.
41. A.L. Cauchy, "Sur l'equation á l'aide de laquelle on détermine les inégalites séculaires des mouvements des planétes [On the Equation by which the Inequalities for the Secular Movement of Planets Is Determined]," *Oeuvres Completes (II, e Séie)*, Vol. 9, 1829.
42. C.G.J. Jacobi, "Über ein leichtes Verfahren die in der Theorie der Säculärstörungen vorkommenden Gleichungen numerisch aufzulösen [On an Easy Method for the Numerical Solution of the Equations that Appear in the Theory of Secular Perturbations]," *Journal für die reine und angewandte Mathematik*, Vol. 30, 1846, pp. 51–s94.
43. J.H. Wilkinson, "Householder's Method for the Solution of the Algebraic Eigenvalue Problem," *Computer J.*, Vol. 3, 1960, pp. 23–27.
44. R.H. Bartels and G.W. Stewart, "Algorithm 432: The Solution of the Matrix Equation $AX - BX = C$," *Comm. ACM*, Vol. 8, 1972, pp. 820–826.
45. G.H. Golub, S. Nash, and C. Van Loan, "Hessenberg–Schur Method for the Problem $AX + XB = C$," *IEEE Trans. Automatic Control*, Vol. AC-24, 1979, pp. 909–913.
46. J.G.F. Francis, "The QR Transformation, Parts I and II," *Computer J.*, Vol. 4, 1961, pp. 265–271, 1962, pp. 332–345.
47. J. Schur, "Über die charakteristischen Würzeln einer linearen Substitution mit einer Anwendung auf die Theorie der Integral gleichungen [On the Characteristic Roots of a Linear Transformation with an Application to the Theory of Integral Equations]," *Mathematische Annalen*, Vol. 66, 1909, pp. 448–510.
49. M. Gu and S.C. Eisenstat, "A Divide-and-Conquer Algorithm for the Bidiagonal SVD," *SIAM J. Matrix Analysis and Applications*, Vol. 16, 1995, pp. 79–92.
50. E. Beltrami, "Sulle funzioni bilineari [On Bilinear Functions]," *Giornale di Matematiche ad Uso degli Studenti Delle Universita*, Vol. 11, 1873, pp. 98–106. An English translation by D. Boley is available in Tech. Report 90–37, Dept. of Computer Science, Univ. of Minnesota, Minneapolis, 1990.
51. C. Jordan, "Memoire sur les formes bilineaires [Memoir on Bilinear Forms]," *Journal de Mathematiques Pures et Appliquees, Deuxieme Serie*, Vol. 19, 1874, pp. 35–54.
52. G.H. Golub and W. Kahan, "Calculating the Singular Values and Pseudo-Inverse of a Matrix," *SIAM J. Numerical Analysis*, Vol. 2, 1965, pp. 205–224.
53. E.G. Kogbetliantz, "Solution of Linear Systems by Diagonalization of Coefficients Matrix," *Quarterly of Applied Math.*, Vol. 13, 1955, pp. 123–132.
54. T.F. Chan, "Rank Revealing QR Factorizations," *Linear Algebra and Its Applications*, Vol. 88/89, 1987, pp. 67–82.
55. G.W. Stewart, "An Updating Algorithm for Subspace Tracking," *IEEE Trans. Signal Processing*, Vol. 40, 1992, pp. 1535–1541.
56. Z. Bai and J.W. Demmel, "Computing the Generalized Singular Value Decomposition," *SIAM J. Scientific and Statistical Computing*, Vol. 14, 1993, pp. 1464–1468.
57. C.B. Moler and G.W. Stewart, "An Algorithm for Generalized Matrix Eigenvalue Problems," *SIAM J. Numerical Analysis*, Vol. 10, 1973, pp. 241–256.

G.W. Stewart is a professor in the Department of Computer Science and in the Institute for Advanced Computer Studies at the University of Maryland. His research interests focus on numerical linear algebra, perturbation theory, and rounding-error analysis. He earned his PhD from the University of Tennessee, where his advisor was A.S. Householder. Contact Stewart at the Dept. of Computer Science, Univ. of Maryland, College Park, MD 20742; stewart@cs.umd.edu.



THE FORTRAN I COMPILER

The Fortran I compiler was the first demonstration that it is possible to automatically generate efficient machine code from high-level languages. It has thus been enormously influential. This article presents a brief description of the techniques used in the Fortran I compiler for the parsing of expressions, loop optimization, and register allocation.

During initial conversations about the topic of this article, it became evident that we can't identify *the* top compiler algorithm of the century if, as the *CiSE* editors originally intended, we consider only the parsing, analysis, and code optimization algorithms found in undergraduate compiler textbooks and the research literature. Making such a selection seemed, at first, the natural thing to do, because fundamental compiler algorithms belong to the same class as the other algorithms discussed in this special issue. In fact, fundamental compiler algorithms, like the other algorithms in this issue, are often amenable to formal descriptions and, as a result, to mathematical treatment.

However, in the case of compilers the difficulty is that, paraphrasing John Donne, no algorithm is an island, entire of itself. A compiler's components are designed to work together to complement each other. Furthermore, next to this conceptual objection, there is the very practical issue that we don't have enough information to decide whether any of the fundamental compiler algorithms have had a determinant impact on the quality of compilers.

1521-9615/00/$10.00 © 2000 IEEE

DAVID PADUA
University of Illinois at Urbana-Champaign

At the same time, it is almost universally agreed that the most important event of the 20th century in compiling—and in computing—was the development of the first Fortran compiler between 1954 and 1957. By demonstrating that it is possible to automatically generate quality machine code from high-level descriptions, the IBM team led by John Backus opened the door to the Information Age.

The impressive advances in scientific computing, and in computing in general, during the past half century would not have been possible without high-level languages. Although the word *algorithm* is not usually used in that sense, from the definition it follows that a compiler *is* an algorithm and, therefore, we can safely say that the Fortran I translator is the 20th century's top compiler algorithm.

The language

The IBM team not only developed the compiler but also designed the Fortran language, and today, almost 50 years later, Fortran is still the language of choice for scientific programming. The language has evolved, but there is a clear family resemblance between Fortran I and today's Fortran 77, 90, and 95. Fortran's influence is also evident in the most popular languages today, including numerically oriented languages

such as Matlab as well as general-purpose languages such as C and Java.

Ironically, Fortran has been the target of criticism almost from the beginning, and even Backus voiced serious objections: "'von Neuman languages' [like Fortran] create enormous, unnecessary intellectual roadblocks in thinking about programs and in creating the higher-level combining forms required in a powerful programming methodology."[1]

Clearly, some language features, such as implicit typing, were not the best possible choices, but Fortran's simple, direct design enabled the development of very effective compilers. Fortran I was the first of a long line of very good Fortran compilers that IBM and other companies developed. These powerful compilers are perhaps the single most important reason for Fortran's success.

The compiler

The Fortran I compiler was fairly small by today's standards. It consisted of 23,500 assembly language instructions and required 18 person-years to develop. Modern commercial compilers might contain 100 times more instructions and require many more person-years to develop. However, its size not withstanding, the compiler was a very sophisticated and complex program. It performed many important optimizations—some quite elaborate even by today's standards—and it "produced code of such efficiency that its output would startle the programmers who studied it."[1]

However, as expected, the success was not universal.[2] The compiler seemingly generated very good code for regular computations; however, irregular computations, including sparse and symbolic computations, are generally more difficult to analyze and transform. Based on my understanding of the techniques used in the Fortran I compiler, I believe that it did not do as well on these types of computations. A manifestation of the difficulties with irregular computations is that subscripted subscripts, such as `A(M(I,J),N(I,J))`, were not allowed in Fortran I.

The compiler's sophistication was driven by the need to produce efficient object code. The project would not have succeeded otherwise. According to Backus:

> It was our belief that if Fortran, during its first months, were to translate any reasonable scientific source program into an object program only half as fast as its hand-coded counterpart, the acceptance of our system would be in serious danger.[1]

The flip side of using novel and sophisticated compiler algorithms was implementation and debugging complexity. Late delivery and many bugs created more than a few Fortran skeptics, but Fortran eventually prevailed:

> It gradually got to the point where a program in Fortran had a reasonable expectancy of compiling all the way through and maybe even of running. This gradual change in status from an experimental to a working system was true of most compilers. It is stressed here in the case of Fortran only because Fortran is now almost taken for granted, as if it were built into the computer hardware.[2]

Optimization techniques

The Fortran I compiler was the first major project in code optimization. It tackled problems of crucial importance whose general solution was an important research focus in compiler technology for several decades. Many classical techniques for compiler analysis and optimization can trace their origins and inspiration to the Fortran I compiler. In addition, some of the terminology the Fortran I implementers used almost 50 years ago is still in use today. Two of the terms today's compiler writers share with the 1950s IBM team are *basic block* ("a stretch of program which has a single entry point and a single exit point"[3]) and *symbolic/real registers*. Symbolic registers are variable names the compiler uses in an intermediate form of the code to be generated. The compiler eventually replaces symbolic registers with real ones that represent the target machine's registers.

Although more general and perhaps more powerful methods have long since replaced those used in the Fortran I compiler, it is important to discuss Fortran I methods to show their ingenuity and to contrast them with today's techniques.

Parsing expressions

One of the difficulties designers faced was how to compile arithmetic expressions taking into account the precedence of operators. That is, in the absence of parentheses, exponentiation should be evaluated first, then products and divisions, followed by additions and subtractions. Operator precedence was needed to avoid extensive use of parentheses and the problems associated with them. For example, IT, an experimental compiler completed by A. Perlis and J.W. Smith in 1956 at the Carnegie Institute of Technology,[4] did not assume operator precedence. As

Donald Knuth pointed out: "The lack of operator priority (often called precedence or hierarchy) in the IT language was the most frequent single cause of errors by the users of that compiler."[5]

The Fortran I compiler would expand each operator with a sequence of parentheses. In a simplified form of the algorithm, it would

- replace + and - with `))+((` and `))-((`, respectively;
- replace * and / with `)*(` and `)/(`, respectively;
- add `((` at the beginning of each expression and after each left parenthesis in the original expression; and
- add `))` at the end of the expression and before each right parenthesis in the original expression.

It is interesting to contrast the parsing algorithm of Fortran I with more advanced parsing algorithms.

Although not obvious, the algorithm was correct, and, in the words of Knuth, "The resulting formula is properly parenthesized, believe it or not."[5] For example, the expression `A + B * C` was expanded as `((A))+((B)*(C))`. The translation algorithm then scanned the resulting expression from left to right and inserted a temporary variable for each left parenthesis. Thus, it translated the previous expression as follows:

u_1=u_2+u_4; u_2=u_3; u_3=A; u_4=u_5*u_6; u_5=B; u_6=C.

Here, variable u_i ($2 \le i \le 5$) is generated when the $(i-1)$th left parenthesis is processed. Variable u_1 is generated at the beginning to contain the expression's value. These assignment statements, when executed from right to left, will evaluate the original expression according to the operator precedence semantics. A subsequent optimization eliminates redundant temporaries. This optimization reduces the code to only two instructions:

u_1=A+u_4; u_4=B*C.

Here, variables A, B, and C are propagated to where they are needed, eliminating the instructions rendered useless by this propagation.

In today's terminology, this optimization was equivalent to applying, at the expression level, *copy propagation* followed by *dead-code elimination*.[6] Given an assignment `x = a`, copy propagation substitutes `a` for occurrences of `x` whenever it can determine it is safe to do so. Dead-code elimination deletes statements that do not affect the program's output. Notice that if `a` is propagated to all uses of `x`, `x = a` can be deleted.

The Fortran I compiler also identified permutations of operations, which reduced memory access and eliminated redundant computations resulting from common subexpressions.[3] It is interesting to contrast the parsing algorithm of Fortran I with more advanced parsing algorithms developed later on. These algorithms, which are much easier to understand, are based on syntactic representation of expressions such as:[7]

expression = *term* [[+ | −] *term*]...
term = *factor* [[* | /] *factor*]...
factor = *constant* | *variable* | (*expression*)

Here, a *factor* is a *constant*, *variable*, or *expression* enclosed by parentheses. A *term* is a factor possibly followed by a sequence of factors separated by * or /, and an expression is a term possibly followed by a sequence of terms separated by + or −. The precedence of operators is implicit in the notation: terms (sequences of products and divisions) must be formed before expressions (sequences of additions and subtractions). When represented in this manner, it is easy to build a recursive descent parser with a routine associated with each type of object, such as term or factor. For example, the routine associated with term will be something like

```
procedure term(){
        call factor()
        while token is * or / {
                get next token
                call factor()
                }
        }
```

Multiplication and division instructions could be generated inside the `while` loop (and addition or subtraction in a similar routine written to represent expressions) without redundancy, thus avoiding the need for copy-propagation or dead-code-elimination optimization within an expression.

DO loop optimizations and subscript computations

One of the Fortran I compiler's main objectives was "to analyze the entire structure of the program in order to generate optimal code from DO state-

ments and references to subscripted variables."[1] For example, the address of the Fortran array element `A(I,J,c3*K+6)` could take the form

$$\texttt{base\_A+I-1+(J-1)*}d_i\texttt{+(}c_3\texttt{*K+6-1)*}d_i\texttt{*}d_j$$

where d_i and d_j are the length of the first two dimensions of `A`, and these two values as well as the coefficient c_3 are assumed to be constant. Clearly, address expressions such as this can slow down a program if not computed efficiently.

It is easy to see that there are constant subexpressions in the address expression that can be incorporated in the address of the instruction that makes the reference.[3] Thus, an instruction making reference to the previous array element could incorporate the constant $\texttt{base\_A+(6-1)*}d_i\texttt{*}d_j\texttt{-}d_i\texttt{-1}$. It is also important to evaluate the variant part of the expression as efficiently as possible. The Fortran I compiler used a pattern-based approach to achieve this goal. For the previous expression, every time "`K` is increased by `n` (under control of a `DO`), the index quantity is increased by $c_3d_id_jn$, giving the correct new value."[3]

Today's compilers apply *removal of loop invariants*, *induction-variable detection*, and *strength reduction* to accomplish similar results.[6,8] The idea of induction-variable detection is to identify those variables within a loop that assume a sequence of values forming an arithmetic sequence. After identifying these induction variables, strength reduction replaces multiplications of induction-variable and loop-invariant values with additions.

The Fortran I compiler applied, instead, a single transformation that simultaneously moved subexpressions to the outermost possible level and applied strength reduction. A limitation of the Fortran I compiler, with respect to modern methods, was that it only recognized loop indices as induction variables:

> It was decided that it was not practical to track down and identify linear changes in subscripts resulting from assignment statements. Thus, the sole criterion for linear changes, and hence for efficient handling of array references, was to be that the subscripts involved were being controlled by `DO` statements.[1]

Register allocation

The IBM 704, the Fortran I compiler's target machine, had three index registers. The compiler applied register allocation strategies to reduce the number of load instructions needed to bring values to the index registers. The compiler section that Sheldon Best designed, which performed index-register allocation, was extremely complex and probably had the greatest influence on later compilers.[1] Indeed, seven years after Fortran I was delivered, Saul Rosen wrote:

> Part of the index register optimization fell into disuse quite early, but much of it was carried along into Fortran II and is still in use on the 704/9/90. In many programs it still contributes to the production of better code than can be achieved on the new Fortran IV compiler.[2]

The register allocation section was preceded by another section whose objective was to create what today is called a control-flow graph. The nodes of this graph are basic blocks and its arcs represent the flow of execution. Absolute execution frequencies were computed for each basic block using a Monte Carlo method and the information provided by Frequency statements. Fortran I programmers had to insert the Frequency statements in the source code to specify the branching probability of `IF` statements, computed `GOTO` statements, and average iteration counts for `DO` statements that had variable limits.[9]

Although the Monte Carlo algorithm delivered the necessary results, not everybody liked the strategy.

Compilers have used Frequency information for register allocation and for other purposes. However, modern compilers do not rely on programmers to insert information about frequency in the source code. Modern register allocation algorithms usually estimate execution frequency using syntactic information such as the level of nesting. When compilers used actual branching frequencies, as was the case with the Multiflow compiler,[10] they obtained the information from actual executions of the source program.

Although the Monte Carlo Algorithm delivered the necessary results, not everybody liked the strategy:

> The possibility of solving the simultaneous equations determining path frequency in terms of transition frequency using known methods for solving sparse matrix equations was considered, but no methods which would work in the presence of `DO`-loops and assigned `GOTO` statements [were] hit upon, although `IF`-type branches alone could

> have been handled without explicit interpretation. The frequency estimating simulation traced the flow of control in the program through a fixed number of steps, and was repeated several times in an effort to secure reliable frequency statistics. Altogether an odd method![9]

With the estimated value of execution frequency at hand, the compiler proceeded to create *connected regions*, similar to the traces used many years later in the Multiflow compiler. Regions were created iteratively. In each iteration, the control flow graph was scanned one at a time by finding at each step the basic block with the highest absolute execution frequency. Then, working backwards and forward, a chain was formed by following the branches with the highest probability of execution as specified in the Frequency statements. Then, registers were allocated in the new region

> ... by simulating the action of the program. Three cells are set aside to represent the object machine index registers. As each new tagged instruction is encountered, these cells are examined to see if one of them contains the required tag; if not, the program is searched ahead to determine which of the index registers is the least undesirable to replace.[3]

The new regions could connect with old regions and subsume them into larger regions.

> In processing a new path connecting two previously disconnected regions, register usage was matched by permuting all the register designations of one region to match those of the other as necessary.[9]

The process of dealing with loops was somewhat involved.

> In processing a new path linking a block to itself and thus defining a loop, the loop was first considered to be concatenated with a second copy of itself, and straight-line register allocation carried out in normal fashion through the first of the two copies, with look-ahead extending into the second copy. ...Straight-line allocation was carried out for a second loop copy in essentially normal fashion.[9]

The only difference was that the look-ahead procedure employed during this allocation was a modified version of the original look-ahead procedure to account for register reuse across loop iterations. Finally, "the allocation produced for the second loop was that ultimately used in generating machine code."[9]

The "least undesirable" register the look-ahead procedure identified was one whose value was dead or, if all registers were live, the one reused most remotely within the region. This strategy is the same as that proved optimal by Laszlo A. Belady in a 1965 paper for page replacement strategies.[11] Belady's objective was to minimize the number of page faults; as a result, the algorithm is optimal "as long as one is concerned only with minimizing the number of loads of symbolic indexes into actual registers and not with minimizing the stores of modified indexes."[9]

The goal, of course, was not to prove or even achieve optimality of the register allocation algorithm. In fact,

> [i]n order to simplify the index register allocation, it was implicitly assumed that calculations were not to be reordered. The contrary assumption would have introduced a new order of difficulty into the allocation process, and required the abstraction of additional information from the program to be processed.[9]

This assumption meant that the result is not always optimal because, in some cases, "... there is much advantage to be had by reordering computations."[7] Nevertheless, "... empirically, Best's 1955–1956 procedure appeared to be optimal."[1]

During the last decade, the relative importance of traditional programming languages as the means to interact with computers has rapidly declined. The availability of powerful interactive applications has made it possible for many people to use computers without needing to write a single line of code.

Although traditional programming languages and their compilers are still necessary to implement these applications, this is bound to change. I do not believe that 100 years hence computers will still be programmed the same way they are today. New applications-development technology will supersede our current strategies that are based on conventional languages. Then, the era of compilers that Fortran I initiated will come to an end.

Technological achievements are usually of interest for a limited time only. New techniques or devices rapidly replace old ones in an endless

cycle of progress. All the techniques used in the Fortran I compiler have been replaced by more general and effective methods. However, Fortran I remains an extraordinary achievement that will forever continue to impress and inspire.

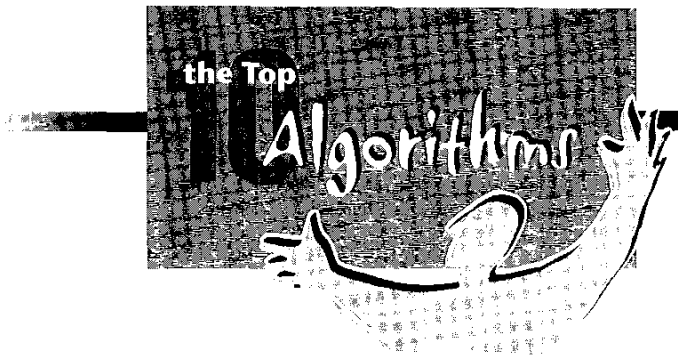
Acknowledgments

This work was supported in part by US Army contract N66001-97-C-8532; NSF contract ACI98-70687; and Army contract DABT63-98-1-0004. This work is not necessarily representative of the positions or policies of the Army or Government.

References

1. J. Backus, "The History of Fortran I, II, and III," *IEEE Annals of the History of Computing*, Vol. 20, No. 4, 1998.
2. S. Rosen, "Programming Systems and Languages—A Historical Survey," *Proc. Eastern Joint Computer Conf.*, Vol. 25, 1964, pp. 1–15, 1964; reprinted in *Programming Systems and Languages*, S. Rosen, ed., McGraw-Hill, New York, 1967, pp. 3–22.
3. J.W. Backus et al., "The Fortran Automatic Coding System," *Proc. Western Joint Computer Conf.*, Vol. 11, 1957, pp. 188–198; reprinted in *Programming Systems and Languages*, S. Rosen, ed., McGraw-Hill, New York, 1967, pp. 29–47.
4. D.E. Knuth and L.T. Pardo, "Early Development of Programming Languages," *Encyclopedia of Computer Science and Technology*, Vol. 7, Marcel Dekker, New York, 1977, p. 419.
5. D.E. Knuth, "A History of Writing Compilers," 1962; reprinted in *Compiler Techniques*, B.W. Pollack, ed., Auerbach Publishers, Princeton, N.J., 1972, pp. 38–59.
6. A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers, Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass., 1988.
7. W.M. McKeeman, "Compiler Construction," *Compiler Construction: An Advanced Course*, F.L. Bauer and J.Eickel, eds., Lecture Notes in Computer Science, Vol. 21, Springer-Verlag, Berlin, 1976.
8. S.S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan Kaufmann, San Francisco, 1997.
9. J. Cocke and J.T. Schwartz, *Programming Languages and Their Compilers, Courant Inst. of Mathematical Sciences*, New York Univ., New York, 1970.
10. J.A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Trans. Computers*, Vol. C-30, No. 7, July 1981, pp. 478–490.
11. L.A. Belady, "A Study of Replacement Algorithms for Virtual Storage Computers," *IBM Systems J.*, Vol. 5, No. 2, 1966, pp. 78–101.

David Padua is a professor of computer science at the University of Illinois, Urbana-Champaign. His interests are in compiler technology, especially for parallel computers, and machine organization. He is a member of the ACM and a fellow of the IEEE. Contact him at the Dept. of Computer Science, 3318 Digital Computer Laboratory, 1304 W. Springfield Ave., Urbana, IL 61801; padua@uiuc.edu; polaris.cs.uiuc.edu/~padua.



THE QR ALGORITHM

After a brief sketch of the early days of eigenvalue hunting, the author describes the QR Algorithm and its major virtues. The symmetric case brings with it guaranteed convergence and an elegant implementation. An account of the impressive discovery of the algorithm brings the article to a close.

I assume you share the view that the rapid computation of a square matrix's eigenvalues is a valuable tool for engineers and scientists.[1] The QR Algorithm solves the eigenvalue problem in a very satisfactory way, but this success does not mean the QR Algorithm is necessarily the last word on the subject. Machines change and problems specialize. What makes the experts in matrix computations happy is that this algorithm is a genuinely new contribution to the field of numerical analysis and not just a refinement of ideas given by Newton, Gauss, Hadamard, or Schur.

Early history of eigenvalue computations

Matrix theory dramatically increased in importance with the arrival of matrix mechanics and quantum theory in the 1920s and 1930s. In the late 1940s, some people asked themselves how the digital computer might be employed to solve the matrix eigenvalue problem. The obvious approach was to use a two-stage method. First, compute the coefficients of the characteristic polynomial, and then compute the zeros of the characteristic polynomial.

There are several ingenious ways to accomplish the first stage in a number of arithmetic operations proportional to n^3 or n^4, where n is the order of the matrix.[2] The second stage was a hot topic of research during this same time. Except for very small values of n, $n \leq 10$, this two-stage approach is a disaster on a computer with fixed word length. The reason is that the zeros of a polynomial are, in general, incredibly sensitive to tiny changes in the coefficients, whereas a matrix's eigenvalues are often, but not always, insensitive to small uncertainties in the n^2 entries of the matrix. In other words, the replacement of those n^2 entries by the characteristic polynomial's n coefficients is too great a condensation of the data.

A radical alternative to the characteristic polynomial is the use of similarity transformations to obtain a nicer matrix with the same eigenvalues. The more entries that are zero, the nicer the matrix. Diagonal matrices are perfect, but a triangular matrix is good enough for our purposes be-

1521-9615/00/$10.00 © 2000 IEEE

BERESFORD N. PARLETT
University of California, Berkeley

cause the eigenvalues lie on the main diagonal. For deep theoretical reasons, a triangular matrix is generally not attainable in a finite number of arithmetic operations. Fortunately, one can get close to triangular form with only $O(n^3)$ arithmetic operations. More precisely, a matrix is said to be *upper Hessenberg* if it is upper triangular with an extra set of nonzero entries just below the diagonal in positions $(i+1, i)$, $i = 1, 2, \ldots, n-1$, and these are called the *subdiagonal entries*. What is more, the similarity transformations needed to obtain a Hessenberg form (it is not unique) can be chosen so that no computed matrix entry ever exceeds the norm of the original matrix. Any proper norm, such as the square root of the sum of the squares of the entries, will do.

This property of keeping intermediate quantities from becoming much bigger than the original data is important for computation with fixed-length numbers and is called *stability*. The hard task is to find similarity transformations that both preserve Hessenberg form and eliminate those subdiagonal entries. This is where the QR Algorithm comes to the rescue. The subroutine DHSEQR in the Lapack library embodies the latest implementation.[3]

Let us try to put the improvement based on QR in perspective. It has reduced the time for standard eigenvalue computations to the time required for a few matrix multiplies.

The LU and QR Algorithms

Suppose $B = XY$, with X invertible. Then the new matrix $C := YX$ is similar to B, because $C = YX = X^{-1}BX$. Although intriguing, this observation does not appear to be of much use to eigenvalue hunters. Let us recall two well-known ways of factoring a matrix:

- Triangular factorization (or Gaussian elimination), $B = LU$, where L is lower triangular with 1's along the main diagonal and U is upper triangular. This factorization is not always possible. The multipliers in the reduction are stored in L, the reduced matrix in U.
- QR (or orthogonal triangular) factorization, $B = QR$, where Q is unitary, $Q^{-1} = Q^*$ (conjugate transpose of Q), and R is upper triangular with nonnegative diagonal entries. This factorization always exists. Indeed, the columns of Q are the outputs of the Gram-Schmidt orthonormalizing process when it is executed in exact arithmetic on the columns of $B = (\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n)$.

Each factorization leads to an algorithm by iteration.

The LU transform of B is $UL = L^{-1}BL$, and the QR transform of B is $RQ = Q^{-1}BQ = Q^*BQ$. In general, the LU or QR transform of B will not have more zero entries than B. The rewards of using this transform come only by repetition.

Theorem 1 (Fundamental Theorem). If B's eigenvalues have distinct absolute values and the QR transform is iterated indefinitely starting from $B_1 = B$,

$$\begin{array}{lcll} \text{Factor} & B_j & = & Q_jR_j, \\ \text{Form} & B_{j+1} & = & R_jQ_j, \qquad j = 1, 2, \ldots. \end{array}$$

then, under mild conditions on the eigenvector matrix of B, the sequence $\{B_j\}$ converges to the upper triangular form (commonly called the Schur form) of B with eigenvalues in monotone decreasing order of absolute value down the diagonal.

This result is not obvious; see James Hardy Wilkinson's article for proofs.[4] The procedure that generates $\{B_j\}$ is called the *basic* QR Algorithm.

A similar theorem holds for the basic LU algorithm, provided all the transforms exist. It takes several clever observations to turn this simple theory into the highly successful QR Algorithm of today.

Invariance of the Hessenberg form

If B is an upper Hessenberg matrix (entry (i, j) vanishes if $i > j + 1$), then so are all its QR iterates and LU iterates. This useful result is easy to see because it depends only on the pattern of zeros in the matrices and not on the values of the nonzero entries. If B_j is Hessenberg, then so is Q_j, because R_j^{-1} is triangular. Consequently, $R_jQ_j = (B_{j+1})$ is also Hessenberg. The cost of computing the QR factorization falls from $O(n^3)$ to $O(n^2)$ when the matrix is Hessenberg and $n \times n$.[5] A similar result holds for LU iterates.

Fortunately, any matrix can be reduced by similarities to Hessenberg form in $O(n^3)$ arithmetic operations in a stable way, and from this point on we will assume this reduction has been performed as an initial phase. The mild conditions mentioned in the Fundamental Theorem are satisfied by upper Hessenberg matrices with nonzero subdiagonals.[6]

Accelerating convergence

The Hessenberg sequences $\{B_j\}$ and $\{C_j\}$ produced by the basic algorithms converge linearly, and that is too slow for impatient customers. We can improve the situation by making a subtle

change in our goal. Instead of looking at the matrix sequence $\{B_j\}$, we can focus on the $(n, n-1)$ entry of each matrix, the last subdiagonal entry. When the $(n, n-1)$ entry is negligible, the (n, n) entry is an eigenvalue—to within working precision—and column n does not influence the remaining eigenvalues. Consequently, the variable n can be reduced by 1, and computation continues on the smaller matrix. We say that the nth eigenvalue has been *deflated*. The top of the matrix need not be close to triangular form. Thus, convergence refers to the scalar sequence of $(n, n-1)$ entries. The rate of convergence of this sequence can be vastly improved, from linear to quadratic, by using the shifted QR Algorithm: Let $B_1 = B$. For $i = 1, 2, \ldots$ until convergence,

Select a shift s_i
Factor $B_i - s_i I = Q_i R_i$
Form $B_{i+1} = R_i Q_i + s_i I = Q_i^* B_i Q_i$.

In principle, each shift strategy has a different convergence theory. It is rare that more than $2n$ QR iterations are needed to compute all the eigenvalues of B.

The double shift implementation for real matrices

There is a clever variation on the shifted QR Algorithm that I should mention. In many applications, the initial matrix is real, but some of the eigenvalues are complex. The shifted algorithm must then be implemented in complex arithmetic to achieve quadratic convergence. The man who first presented the QR Algorithm, J.G.F Francis,[7] showed how to keep all arithmetic in the real field and still retain quadratic convergence.

Let us see how it is done. Without loss, take $j = 1$. Consider two successive steps:

$$\begin{aligned} B_1 - s_1 I &= Q_1 R_1 \\ B_2 &= R_1 Q_1 + s_1 I \\ B_2 - s_2 I &= Q_2 R_2 \\ B_3 &= R_2 Q_2 + s_2 I. \end{aligned}$$

It turns out, after some manipulation, that

$$(Q_1 Q_2)(R_2 R_1) = (B_1 - s_1 I)(B_1 - s_2 I)$$

and

$$B_3 = (Q_1 Q_2)^* B_1 (Q_1 Q_2).$$

Suppose s_1 and s_2 are either both real or a complex conjugate pair. Then $(B_1 - s_1 I)(B_1 - s_2 I)$ is real. By the uniqueness of the QR factorization, $Q_1 Q_2$ is the Q factor of $(B_1 - s_1 I)(B_1 - s_2 I)$ and so is real orthogonal, not just unitary. Hence, B_3 is a product of three real matrices and thus real.

The next challenge is to compute B_3 from B_1 and s (complex) without constructing B_2. The solution is far from obvious and brings us to the concept of *bulge chasing*, a significant component of the QR success story.

Bulge chasing

The theoretical justification comes from the Implicit Q or Uniqueness of Reduction property.

Theorem 2 (Uniqueness). If Q is orthogonal, B is real, and $H = Q^* B Q$ is a Hessenberg matrix in which each subdiagonal entry $h_{i+1,i} > 0$, then H and Q are determined uniquely by B and q_1, the first column of Q.

Now return to the equations above and suppose $s_1 = s$, $s_2 = \bar{s} \neq s$. If B_1 is Hessenberg, then so are all the B_i's. Suppose that B_3 has positive subdiagonal entries. By Theorem 2, both B_3 and $Q_1 Q_2$ are determined by column 1 of $Q_1 Q_2$, which we'll call $\boldsymbol{q}$. Because $R_2 R_1$ is upper triangular, $\boldsymbol{q}$ is a multiple of the first column of

$$B_1^2 - 2(\mathrm{Re}\, s) B_1 + |s|^2 I.$$

Because B_1 is Hessenberg, the vector $\boldsymbol{q}$ is zero except in its top three entries.

The following three-stage algorithm computes B_3. It uses orthogonal matrices $H_j, j = 1, 2, \ldots, n-1$, such that H_j is the identity except for a 3×3 submatrix in rows and columns $j, j+1, j+2$. H_{n-1} differs from I only in its trailing 2×2 matrix.

1. Compute the first three entries of $\boldsymbol{q}$.
2. Compute a matrix H_1 such that $H_1^t \boldsymbol{q}$ is a multiple of $\boldsymbol{e}_1$, the first column of I. Form $C_1 = H_1^t B_1 H_1$. It turns out that C_1 is upper Hessenberg except for nonzeros in positions (3, 1), (4, 1), and (4, 2). This little submatrix is called the *bulge*.
3. Compute a sequence of matrices $H_2, \ldots, H_{n-1}$ and $C_j = H_j^t C_{j-1} H_j, j = 2, \ldots, n-1$ such that $C_{n-1} = H_{n-1}^t \ldots H_2^t C_1 H_2 \ldots H_{n-1}$ is a Hessenberg matrix with positive subdiagonal entries. More on H_j below.

We claim that $C_{n-1} = B_3$. Recall that column 1 of H_j is $\mathbf{e}_1$ for $j > 1$. Thus, $H_1 H_2 \ldots H_{n-1} \mathbf{e}_1 = H_1 \mathbf{e}_1 = \mathbf{q}/\|\mathbf{q}\| = (Q_1 Q_2)\mathbf{e}_1$. Now $C_{n-1} = (H_1 \ldots H_{n-1})^t B_1 (H_1 \ldots H_{n-1})$, and $B_3 = (Q_1 Q_2)^t B_1 (Q_1 Q_2)$. The Implicit Q Theorem ensures that B_3 and C_{n-1} are the same. Moreover, if s is not an eigen-

value, then C_{n-1} must have positive subdiagonal entries in exact arithmetic.

Step 3 involves $n-2$ minor steps, and at each one only three rows and columns of the array are altered. The code is elegant, and the cost of forming C_{n-1} is about $5n^2$ operations. The transformation $C_2 \rightarrow H_2{}^t C_1 H_2$ pushes the bulge into positions (4, 2), (5, 2), and (5, 3), while creating zeros in positions (3, 1) and (4, 1). Subsequently, each operation with an H matrix pushes the bulge one row lower until it falls off the bottom of the matrix and the Hessenberg form is restored. The transformation $B_1 \rightarrow B_3$ is called a double step.

It is now necessary to inspect entry $(n-1, n-2)$ as well as $(n, n-1)$ to see whether a deflation is warranted. For complex conjugate pairs, the $(n-1, n-2)$ entry, not $(n, n-1)$, becomes negligible.

The current shift strategies for QR do not guarantee convergence in all cases. Indeed, examples are known where the sequence $\{B_j\}$ can cycle. To guard against such misfortunes, an ad hoc exceptional shift is forced from time to time. A more complicated choice of shifts might produce a nicer theory, but practical performance is excellent.[8]

The symmetric case

The QR transform preserves symmetry for real matrices and preserves the Hermitian property for complex matrices: $B \rightarrow Q^* B Q$. It also preserves Hessenberg form. Because a symmetric Hessenberg matrix is tridiagonal (that means entry (i, j) vanishes if $|i-j| > 1$), the QR Algorithm preserves symmetric tridiagonal form and the cost of a transform plunges from $O(n^2)$ to $O(n)$ operations. In fact, the standard estimate for the cost of computing all the eigenvalues of an $n \times n$ symmetric tridiagonal matrix is $10n^2$ arithmetic operations. Recall that all the eigenvalues are real.

One reason this case is worthy of a section to itself is its convergence theory. Theorem 1 tells us that the basic algorithm (all shifts are zero) pushes the large eigenvalues to the top and the small ones to the bottom. A shift strategy Wilkinson suggested in the 1960s *always* makes the $(n, n-1)$ entry converge to zero. Moreover convergence is rapid. Everyone believes the rate is cubic (very fast), although our proofs only guarantee a quadratic rate (such as Newton's iteration for polynomial zeros).[9]

The implementation for symmetric tridiagonals is particularly elegant, and we devote a few lines to evoke the procedure. The bulge-chasing method described earlier simplifies, because the bulge consists of a single entry on each side of the diagonal.

There is one more twist to the implementation that is worth mentioning. The code can be rearranged so that no square roots need to be computed.[9]

The discovery of the algorithms

There is no obvious benefit in factoring a square matrix B into $B = QR$ and then forming a new matrix $RQ = Q^* B Q$. Indeed, some structure in B might be lost in $Q^* B Q$.

So how did someone come up with the idea of iterating this transformation? Major credit is due to the Swiss mathematician and computer scientist H. Rutishauser. His doctoral thesis was not concerned with eigenvalues but rather with a more general algorithm he invented, which he called the Quotient-Difference (QD) Algorithm.[10] This procedure can be used to find zeros of polynomials or poles of rational functions or to manipulate continued fractions. The algorithm transforms an array of numbers, which Rutishauser writes as

$$Z = (q_1, e_1, q_2, e_2, \ldots, q_{n-1}, e_{n-1}, q_n)$$

into another one, $\hat{Z}$, of the same form.

Let us define two bidiagonal matrices associated with Z. For simplicity, take $n = 5$; then

$$L = \begin{pmatrix} 1 & & & & \\ e_1 & 1 & & & \\ & e_2 & 1 & & \\ & & e_3 & 1 & \\ & & & e_4 & 1 \end{pmatrix}, U = \begin{pmatrix} q_1 & 1 & & & \\ & q_2 & 1 & & \\ & & q_3 & 1 & \\ & & & q_4 & 1 \\ & & & & q_5 \end{pmatrix}.$$

Rutishauser observed that the rhombus rules he discovered for the QD transformation, namely

$$\begin{aligned} \hat{e}_i + \hat{q}_{i+1} &= q_{i+1} + e_{i+1} \\ \hat{e}_i \hat{q}_i &= q_{i+1} e_i \end{aligned},$$

admit the following remarkable interpretation:

$$\hat{L}\hat{U} = UL.$$

Note that UL and LU are tridiagonal matrices with all superdiagonal entries equal to one. In other words, the QD Algorithm is equivalent to the following procedure on tridiagonals $\mathcal{J}$ with unit superdiagonals:

$$\begin{aligned} &\text{Factor} \quad \mathcal{J} = LU, \\ &\text{Form} \quad \hat{\mathcal{J}} = UL. \end{aligned}$$

Thus was the LU transform born. It did not take Rutishauser a moment to see that the idea of reversing factors could be applied to a dense

matrix or a banded matrix. Although the QD Algorithm appeared in 1953 or 1954, Rutishauser did not publish his LU algorithm until 1958.[10,11] He called it LR, but I use LU to avoid confusion with QR.

Unfortunately, the LU transform is not always stable, so the hunt was begun for a stable variant. This variant was found by a young computer scientist J.G.F Francis,[7] greatly assisted by his mentor Christopher Strachey, the first professor of computation at Oxford University. Independent of Rutishauser and Francis, Vera Kublanovskaya in the USSR presented the basic QR Algorithm in 1961.[12] However, Francis not only gave us the basic QR Algorithm, but at the same time exploited the invariance of the Hessenberg form and gave the details of a double step to avoid the use of complex arithmetic.

In 1955, the calculation of the eigenvalues and eigenvectors of a real matrix that was not symmetric was considered a formidable task for which there was no reliable method. By 1965, the task was routine, thanks to the QR Algorithm. However, there is more to be done, because of accuracy issues. The eigenvalues delivered by QR have errors that are tiny, like the errors in rounding a number to 15 decimals, with respect to the size of the numbers in the original matrix. That is good news for the large eigenvalues but disappointing for any that are tiny. In many applications, the tiny eigenvalues are important because they are the dominant eigenvalues of the inverse.

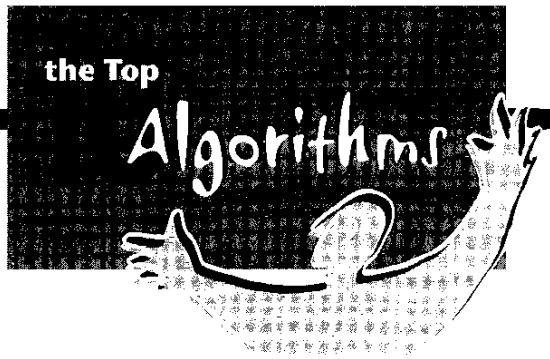
For a long time, this limitation on accuracy was regarded as an intrinsic limitation caused by the limited number of digits for each number. Now we know that there are important cases where the data in the problem do define all the eigenvalues to high relative accuracy, and current work seeks algorithms that can deliver answers to that accuracy.

The QR Algorithm was aimed at matrices with at most a few hundred rows and columns. Its success has raised hopes. Now customers want to find a few eigenpairs of matrices with thousands, even hundreds of thousands, of rows. Most of the entries of these matrices are zero, and they are called sparse. The QR Algorithm is not well suited for such cases. It destroys sparsity and has difficulty finding selected eigenpairs. Since 1970, research has turned toward these challenging cases.[13]

References

1. A. Jennings and J.J. McKeown, *Matrix Computations*, 2nd ed., John Wiley, New York., 1977.
2. D.K. Faddeev and V.N. Faddeeva, *Computational Methods of Linear Algebra*, W.H. Freeman and Co., San Francisco, 1963.
3. E. Anderson et al., *LAPACK Users' Guide*, 2nd ed., SIAM, Philadelphia, 1995.
4. J.H. Wilkinson, "Convergence of the LR, QR, and Related Algorithms," *The Computer J.*, No. 4, 1965, pp. 77–84.
5. G.H. Golub and C.F. Van Loan, *Matrix Computations*, 3rd ed., The Johns Hopkins Univ. Press, Baltimore, 1996.
6. B.N. Parlett, "Global Convergence of the Basic QR Algorithm on Hessenberg Matrices," *Mathematics of Computation*, No. 22, 1968, pp. 803–817.
7. J.G.F Francis, "The QR Transformation, Parts I and II," *The Computer J.*, No. 4, 1961–1962, pp. 265–271 and 332–345.
8. S. Batterson and D. Day, "Linear Convergence in the Shifted QR Algorithm," *Mathematics of Computation*, No. 59, 1992, pp. 141–151.
9. B.N. Parlett, *The Symmetric Eigenvalue Problem*, 2nd ed., SIAM, Philadelphia, 1998.
10. H. Rutishauser, "Der Quotienten-Differenzen-Algorithmus," *Zeitung der Angewandte Mathematik und Physik*, No. 5, 1954, pp. 233–251.
11. H. Rutishauser, "Solution of Eigenvalue Problems with the LR-Transformation," *Nat'l Bureau Standards Applied Mathematics Series*, No. 49, 1958, pp. 47–81.
12. V.N. Kublanovskaya, "On Some Algorithms for the Solution of the Complete Eigenvalue Problem," *USSR Computational Mathematics and Physics*, No. 3, 1961, pp. 637–657.
13. Y. Saad, *Numerical Methods for Large Eigenvalue Problems*, Manchester Univ. Press, Manchester, UK, 1992.

Beresford N. Parlett is an emeritus professor of mathematics and of computer science at the University of California, Berkeley. His professional interest is in matrix computations, especially eigenvalue problems. He received his BA in mathematics from Oxford University and his PhD from Stanford University. He is a member of the AMS and SIAM. Contact him at the Mathematics Dept. and Computer Science Division, EECS Dept., Univ. of California, Berkeley, CA 94720; parlett@math.berkeley.edu.



A PERSPECTIVE ON QUICKSORT

This article introduces the basic Quicksort algorithm and gives a flavor of the richness of its complexity analysis. The author also provides a glimpse of some of its generalizations to parallel algorithms and computational geometry.

Sorting is arguably the most studied problem in computer science, because of both its use in many applications and its intrinsic theoretical importance. The basic sorting problem is the process of rearranging a given collection of items into ascending or descending order. The items can be arbitrary objects with a linear ordering. For example, the items in a typical business data-processing application are records, each containing a special identifier field called a key, and the records must be sorted according to their keys. Sorting can greatly simplify searching for or updating a record. This article focuses only on the case where all the items to be sorted fit in a machine's main memory. Otherwise, the main objective of sorting, referred to as *external sorting*, is to minimize the amount of I/O communication, an issue this article does not address.

Although researchers have developed and analyzed many sorting algorithms, the Quicksort algorithm stands out. This article shows why.

1521-9615/00/$10.00 © 2000 IEEE

JOSEPH JAJA
University of Maryland

Staying power

Tony Hoare presented the original algorithm and several of its variations in 1962,[1] yet Quicksort is still the best-known practical sorting algorithm. This is quite surprising, given the amount of research done to develop faster sorting algorithms during the last 38 years or so. Quicksort remains the sorting routine of choice except when we have more detailed information about the input, in which case other algorithms could outperform Quicksort.

Another special feature of the algorithm is the richness of its complexity analysis and structure, a feature that has inspired the development of many algorithmic techniques for various combinatorial applications. Although the worst-case complexity of Quicksort is poor, we can rigorously prove that its average complexity is quite good. In fact, Quicksort's average complexity is the best possible under certain general complexity models for sorting. More concretely, Quicksort performs poorly for almost sorted inputs but extremely well for almost random inputs.

In retrospective, Quicksort follows the divide-and-conquer strategy (likely one of the oldest military strategies!), one of the most powerful algorithmic paradigms for designing efficient algorithms for combinatorial problems. In particular, a variation of Quicksort, which Hoare also mentioned in his original paper, is based on a randomization step using a random-number generator.

Researchers cultivated this idea years later and significantly enriched the class of randomized algorithms, one of the most interesting areas in theoretical computer science. You could argue that these developments are somewhat independent of Quicksort, but many of the randomized combinatorial algorithms are based on the same randomized divide-and-conquer strategy described in Hoare's original paper. Indeed, many randomized algorithms in computational geometry can be viewed as variations of Quicksort.

What makes Quicksort so interesting from the complexity-analysis point of view is its far superior average complexity.

Finally, a generalization of the randomized version of the Quicksort algorithm results in a parallel sample-sorting strategy, the best-known strategy for sorting on parallel machines, both from a theoretical perspective and from extensive experimental studies.[2]

The basic algorithm

The divide-and-conquer strategy has three phases. First, divide the problem into several subproblems (typically two for sequential algorithms and more for parallel algorithms) of almost equal sizes. Second, solve independently the resulting subproblems. Third, merge the solutions of the subproblems into a solution for the original problem. This strategy's efficiency depends on finding efficient procedures to partition the problem during the initial phase and to merge the solutions during the last phase. For example, the fast Fourier transform follows this strategy, as does Quicksort. Here is a high-level description of Quicksort applied to an array $A[0 : n - 1]$:

1. Select an element from $A[0 : n - 1]$ to be the pivot.
2. Rearrange the elements of A to partition A into a left subarray and a right subarray, such that no element in the left subarray is larger than the pivot and no element in the right subarray is smaller than the pivot.
3. Recursively sort the left and the right subarrays.

Ignoring the implementation details for now, it is intuitively clear that the previous algorithm will correctly sort the elements of A. Equally clear is that the algorithm follows the divide-and-conquer strategy. In this case, Steps 1 and 2 correspond to the partitioning phase. Step 3 amounts to solving the induced subproblems independently, which sorts the entire array, making a merging phase unnecessary.

Two crucial implementation issues are missing from our initial description of the Quicksort algorithm. The first concerns the method for selecting the pivot. The second concerns the method for partitioning the input once the pivot is selected.

Selecting the pivot

Because a critical assumption for an efficient divide-and-conquer algorithm is to partition the input into almost equal-size pieces, we should select the pivot so as to induce almost equal-size partitions of the array A. Assuming A contains n distinct elements, the best choice would be to select the median of A as the pivot. Although some good theoretical algorithms can find the median without first sorting the array, they all incur too much overhead to be useful for a practical implementation of Quicksort.

In reality, there are three basic methods to select the pivot. The first and simplest is to select an element from a fixed position of A, typically the first, as the pivot. In general, this choice of the pivot does not work well unless the input is random. If the input is almost sorted, the input will be partitioned extremely unevenly during each iteration, resulting in a very poor performance of the overall algorithm. The second method for selecting the pivot is to try to approximate the median of A by computing the median of a small subset of A.

One commonly used method is to select the pivot to be the median of the first, the middle, and the last elements of A. Such a method works well, even though we may still end up with very uneven partitions during each iteration for certain inputs. Finally, we can randomly select an element from A to be the pivot by using a random-number generator. In this case, we can rigorously prove that each step will result in an almost even partition with very high probability, regardless of the initial input distribution.

Partitioning the input

Because the algorithm is recursive, I describe a simple partitioning procedure for a general subarray $A[l, r]$, where $l < r$. The procedure manipulates two pointers i and j, where i moves from left to right starting at position l, and j moves from right to left starting at position r. The pointer i is continually incremented until an element $A[i]$ larger than the pivot is encountered. Similarly, j is decremented until an element $A[j]$ smaller than the pivot is encountered. At that time, $A[i]$ and $A[j]$ are exchanged, and the process continues until the two pointers cross each other, which indicates the completion of the partitioning phase. More precisely, Figure 1 gives the following pseudocode procedure for partitioning $A[l, r]$, where all the elements of A are assumed to be distinct, and the left-most element is chosen as the pivot.

```
procedure partition(A, l, u)

begin

Step 1. Set i = l; j = r + 1; pivot = A[l];

Step 2. while(true){
          while(A[++i] < pivot);
          while(A[--j] > pivot);
          if i < j then exchange A[i] and A[j];
            else break;
            }
          A[l] = A[j];
          A[j] = pivot;

end procedure
```

Figure 1. The partitioning procedure for $A[l, r]$.

The partition procedure starts by initializing the pointers i and j and selecting the left-most element to be the pivot. Note that j is initially set to the value $r + 1$ so that the `while` loop for j in Step 2 will start by examining the subarray's right-most element. The last two assignments in Step 2 ensure that the pivot is placed in its correct position in the final sorted order of A. The partition procedure takes linear time with a very small constant factor. Donald Knuth[3] attributes this particular partitioning procedure to Robert Sedgewick.[4]

Quicksort initially calls the partition procedure with the values $l = 0$ and $r = n - 1$, followed by recursive calls to handle the subarrays $A[l, j - 1]$ and $A[j + 1, r]$. We can eliminate the recursive calls by using a stack to keep track of the partitions yet to be sorted. We run Quicksort until the number of elements in the subarray is small (say, fewer than 30); then we use a simpler sorting procedure, such as insertion sort.

Complexity analysis

We can use several measures to analyze a sorting algorithm's performance. We can count the number of comparisons the algorithm makes, because comparisons seem to be the major operations performed by any general sorting algorithm. Another important parameter is the number of swaps the algorithm incurs. Yet another measure, one that is perhaps more relevant for today's processors and their use of memory hierarchies (typically two levels of cache and main memory), is the data movement and its spatial locality. Here, I concentrate on the number of comparisons, for which there are well-known models (for example, algebraic decision trees[5]) that can derive nonlinear lower bounds for sorting.

Worst-case analysis

In general, an algorithm's most commonly used performance metric is the asymptotic estimate of the maximum amount of resources (for example, the number of operations, the number of memory accesses, and memory size) required by any instance of a problem of size n. In sorting, we determine the asymptotic number of comparisons required in the worst case. Let $T(n)$ be the number of comparisons required by our Quicksort algorithm, using any of the methods just described for selecting the pivot. Then the following recurrence equations express $T(n)$:

$$T(n) = \max_{1 \le i \le n-1} \{T(i) + T(n-i)\} + cn$$

$$T(1) = \Theta(1)$$

where cn is an upper bound on the time required to partition the input. The parameter i is the size of the left partition resulting after the first iteration. Because we are focusing on the worst-case scenario, we take the maximum over all possible values of i. It is intuitively clear that the worst case occurs when $i = 1$ (or equivalently, $i = n - 1$), which can happen for each of our three pivot-selection methods. Therefore, $T(n) = \Theta(n^2)$, which is inferior to the worst-case complexity of several of the other known sorting algorithms. However, what makes Quicksort so interesting from the complexity-analysis point of view is its far superior average complexity.

Average-case analysis

Although performing a worst-case analysis on an algorithm is usually easy, it often results in a very pessimistic performance estimate. Known alternative performance measures seem more useful, but deriving their asymptotic behavior is significantly more difficult. One such measure is to establish the average complexity under a certain probability distribution of the input. Not only is deriving such a bound difficult, but also this approach fails to offer adequate ways for defining the probability distribution of the input in most cases.

An alternative approach is to let the algorithm use a random-number generator (such as our third method for randomly selecting the pivot from the input array). In this case, a probabilistic analysis of the algorithm does not require any assumptions about the input distribution, and hence holds for any input distribution. This approach gives rise to the so-called *randomized algorithms*.

Here, I consider the case when the array's first element is selected as the pivot and assume random input. So, the pivot is equally likely to be of any rank between 1 and n. Therefore, the average complexity is given by the recurrence equations,

$$T(n) = \frac{1}{n}\left(\sum_{i=0}^{n-1}\left(T(i)+T(n-i)\right)\right)+cn$$

$$T(1) = \Theta(1)$$

The first equation is equivalent to

$$T(n) = \frac{2}{n}\left(\sum_{i=0}^{n-1} T(i)\right)+cn\ .$$

Multiplying both sides by n gives

$$nT(n) = 2\left(\sum_{i=0}^{n-1} T(i)\right)+cn^2\ .$$

Subtracting the recurrence equation corresponding to $n-1$ from the one corresponding to n, we get

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn - c.$$

Rearranging the terms and dropping c (it is asymptotically irrelevant) gives

$$nT(n) = (n+1)T(n-1) + 2cn.$$

Dividing both sides by $n(n+1)$ gives

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}\ .$$

In particular, we have this sequence of equations:

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

$$\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2c}{n}$$

$$\frac{T(n-2)}{n-1} = \frac{T(n-3)}{n-2} \frac{2c}{n-1}$$

M

$$\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2c}{3}$$

Adding the above equations and considering that

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

where $\gamma \approx 0.577$ is Euler's constant, we find that $T(n) = O(n \log n)$.

Under a fairly general model for sorting that even allows algebraic operations,[5] no algorithm can beat this bound. This analysis formally justifies the superior performance of Quicksort in most practical situations.

Analysis of randomized Quicksort

Consider the case where the pivot is randomly selected from the input array. With high probability—that is, with probability $1 - n^{-c}$ for some positive constant c—the resulting Quicksort algorithm's complexity is $O(n \log n)$, regardless of the initial input distribution.

Our strategy is as follows. We view the Quicksort algorithm as a sequence of iterations such that the first iteration partitions the input into two *buckets*, the second iteration partitions the previous two buckets into four buckets, and so on, until each bucket is small enough (say, ≤ 30 elements). We then use insertion sort to sort these buckets. Another way of looking at this is to unfold the recursion into a tree of partitions, where each level of the tree corresponds to an iteration's partitions. Partitioning the buckets during each iteration takes a deterministic $O(n)$ time. So, our goal is to show that the number of iterations is $O(\log n)$ with high probability.

For any specific element e, the sizes of any two consecutive buckets containing e decrease by a constant factor with a certain probability (Claim 1). Then, with probability $1 - O(n^{-7})$, the bucket containing e will be of size 30 or less after $O(\log n)$ iterations (Claim 2). Using Boole's inequality, we conclude that, with probability $1 - O(n^{-6})$, the bucket of every element has ≤ 30 elements after $O(\log n)$ iterations.[6]

Let e be an arbitrary element of our input array A. Let n_j be the size of the bucket containing e at the end of the jth partitioning step, where $j \geq 1$.

We set $n_0 = n$. Then, this claim holds:

Claim 1. $Pr\{n_{j+1} \geq 7n_j/8\} \leq 1/4$, for any $j \geq 0$.

Proof. An element a partitions the jth bucket into two buckets, one of size at least $7n_j/8$ if and only if $rank(a : B_j) \leq n_j/8$ or $rank(a : B_j) \geq 7n_j/8$. The probability that a random element is among the smallest or largest $n_j/8$ elements of B_j is at most 1/4; hence, Claim 1 follows.

We fix the element e of A, and we consider the sizes of the buckets containing e during various partitioning steps. We call the jth partitioning step successful if $n_j < 7n_{j-1}/8$. Because $n_0 = n$, the size of the bucket containing e after k successful partitioning steps is smaller than $(7/8)^k n$. Therefore, e can participate in at most $c \log(n/30)$ successful partitioning steps, where $c = 1/\log(8/7)$. For the remainder of this proof, all logarithms are to the base 8/7; so the constant c is equal to 1.

Claim 2. Among 20 log n partitioning steps, the probability that an element e goes through 20 log n – log(n/30) unsuccessful partitioning steps is $O(n^{-7})$.

Proof. The random choices made at various partitioning steps are independent; therefore, the events that constitute the successful partitioning steps are independent. So, we can model these events as *Bernoulli trials*. Let X be a random variable denoting the number of unsuccessful partitioning steps among the 20 log n steps. Therefore,

$$Pr\{X > 20\log n - \log(n/30)\} \leq Pr\{X > 19\log n\}$$

$$\leq \sum_{j>19\log n} \binom{20\log n}{j}\left(\frac{1}{4}\right)^j\left(\frac{3}{4}\right)^{20\log n - j}$$

Given that

$$\binom{n}{k} \leq \left(\frac{en}{k}\right)^k$$

we obtain that our probability is

$$\leq \sum_{j>19\log n}\left(\frac{20e\log n}{j}\right)^j\left(\frac{1}{4}\right)^j = \sum_{j>19\log n}\left(\frac{5e\log n}{j}\right)^j$$

$$\leq \sum_{j>19\log n}\left(\frac{5e\log n}{19\log n}\right)^j = \sum_{j>19\log n}\left(\frac{5e}{19}\right)^j = O\left(n^{-7}\right) .$$

Therefore, Claim 2 follows.

By Boole's inequality, the probability that one or more elements of A go through 20 log n – log(n/30) unsuccessful steps is at most $O(n \times n^{-7}) = O(n^{-6})$. Thus, with probability $1 - O(n^{-6})$, the algorithm terminates within 20 log n iterations. So, we have proven that the algorithm takes $O(n \log n)$ time with high probability.

Extension to parallel processing

For our purposes, a parallel machine is simply a collection of processors interconnected to allow the coordination of their activities and the exchange of data. Two types of parallel machines currently dominate. The first is symmetric multiprocessors (SMPs), which are the main choice for the high-end server market, and soon will be on most desktop computers. The second type clusters high-end processors through proprietary interconnect (for example, the IBM SP High-Performance Switch) or through off-the-shelf interconnect (the ATM switch, gigabit Ethernet, and so on).

One way to choose the pivots is by randomly sampling the input elements—hence, the name sample sort.

A parallel sorting algorithm tries to exploit the various resources on the parallel machine to speed up the execution time. In addition to distributing the load almost evenly among the various processors, a good parallel algorithm should minimize the communication and coordination among the different processors. Parallel algorithms often perform poorly (and sometimes execution time increases with the number of processors) primarily because of the amount of communication and coordination required between the different processors.

Quicksort's strategy lends itself well to parallel machines, resulting in *sample sorting*:

1. Select $p - 1$ pivots, where p is the number of processors available.
2. Partition the input array into p subarrays, such that every element in the ith subarray is smaller than each element in the $(i + 1)$th subarray.
3. Solve the problem by getting the ith processor to sort the ith subarray.

One way to choose the pivots is by randomly sampling the input elements—hence, the name *sample sort*. (A generalization of the bucket-sorting method, also called sample sort, involves sampling as well.[7]) As in the sequential case, the algorithm's efficiency depends on how the pivots are selected and on the method used to par-

tition the input into the p partitions.

Before addressing these issues, let's define our problem more precisely. We have n distinct elements that are distributed evenly among the p processors of a parallel machine, where we assume that p divides n evenly. The purpose of sorting is to rearrange the elements so that the smallest n/p elements appear in processor P_1 in sorted order, the second smallest n/p elements appear in processor P_2 in sorted order, and so on.

A simple way to realize the sample sorting strategy is to have each processor choose s samples from its n/p input elements (In the next section, I'll show you one way to do this.), route the ps samples into a single processor, sort the samples on that processor, and select every sth element as a pivot.[8] Each processor partitions its input elements into p groups using the pivots. Then, the first group in each processor is sent to P_1, the second to P_2, and so on. Finally, we sort the overall input by sorting the elements in each processor separately.

The first difficulty with this approach is the work involved in gathering and sorting the samples. A larger value of s improves load balancing but increases overhead. The second difficulty is that the communication required for routing the elements to the appropriate processors can be extremely inefficient because of large variations in the number of elements destined for different processors.

Consider a variation that scales optimally with high probability and has performed extremely well on several distributed-memory parallel machines using various benchmarks.[2] Here are the algorithm's steps:

1. *Randomization.* Each processor P_i $(1 \le i \le p)$ randomly assigns each of its n/p elements to one of p buckets. With high probability, no bucket will receive more than $c_1(n/p^2)$ elements for some constant c_1. Each processor then routes the contents of bucket j to P_j.
2. *Local sorting.* Each processor sorts the elements received in Step 1. The first processor then selects $p - 1$ pivots that partition its sorted sublist evenly and broadcasts the pivots to the other $p - 1$ processors.
3. *Local partitioning.* Each processor uses binary search on its local sorted list to partition it into p subsequences using the pivots received from Step 2. Then the jth subsequence is sent to P_j.
4. *Local merging.* Each processor merges the p sorted subsequences received to produce the ith column of the sorted array.

Our randomized sampling algorithm runs in $O((n \log n)/p)$ with high probability, using only two rounds of balanced communication and a broadcast of $p - 1$ elements from one processor to the remaining processors.[2]

Applications to computational geometry

Computational geometry is the study of designing efficient algorithms for computational problems dealing with objects in Euclidean space. This rich class of problems arises in many applications, such as computer graphics, computer-aided design, robotics, pattern recognition, and statistics. Randomization techniques play an important role in computational geometry, and most of these techniques can be viewed as higher-dimensional generalizations of Quicksort.[9] We'll look at the simplest example of such techniques, which is also related to the sample sorting procedure I just described.

Let N be a set of points on the real line R such that $|N| = n$. Sorting the n points amounts to partitioning R into $n + 1$ regions, each defined by an (open) interval. Let S be a random point of N that divides R into two halves. Let N_1 and N_2 be the subsets of N contained in these halves. Then, we expect the sizes of N_1 and N_2 to be roughly equal. As in the Quicksort algorithm, we can sort N_1 and N_2 recursively.

As a generalization, let S be a random sample of N of size s. One way to construct S is to choose the first element of S randomly from N and delete it from N. Continue the process, each time selecting a random element from the remaining set N and deleting it from N, until we have s elements in our set S. Such a procedure is called *sampling without replacement.*

Does S divide the real line R into regions of roughly equal size? As I stated earlier, the partition is defined by the set of open intervals. The *conflict size* of any such interval I is the number of points of N and I, but not in S. We would like to determine whether each interval's conflict size is $O(n/s)$ with high probability. Unfortunately, we can only show a weaker result—namely, that with probability greater than 1/2, each interval's conflict size is $O((n \log s)/s)$.

We can apply the same random sampling technique to the case where N is a set of lines in the plane. A random sample S generates an arrangement (that is, the convex regions in the plane induced by the lines in S). The conflict size is the number of lines in N, but not in S, that intersect a region of the arrangement of S. If we refine the

arrangement of S so that each region has a bounded number of sides, then with high probability, the conflict size of every region is indeed $O((n \log s)/s)$.

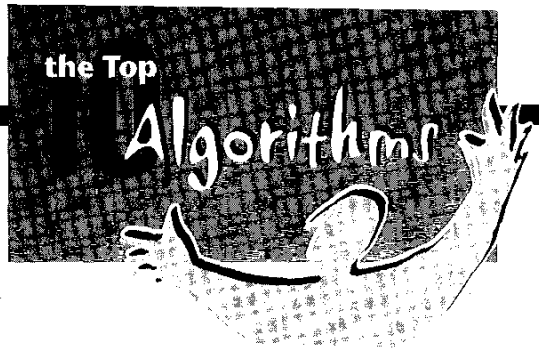
These two simple generalizations of the randomized Quicksort can be used to develop efficient algorithms for various problems in computational geometry.[9]

The basic strategy of Quicksort—randomized divide-and-conquer—represents a fundamental approach for designing efficient combinatorial algorithms that will remain a source of inspiration for researchers for many years to come. In particular, the full potential of the technique in handling higher-dimensional problems in computational geometry and in developing parallel algorithms for combinatorial problems is yet to be fully exploited. In the future, we can anticipate vigorous research progress along these directions.

References

1. C.A.R. Hoare, "Quicksort," *The Computer J.*, Vol. 5, No. 1, Apr. 1962, pp. 10–15.
2. D. Helman, D. Bader, and J. JaJa, "A Randomized Parallel Sorting Algorithm with an Experimental Study," *J. Parallel and Distributed Computing*, Vol. 52, No. 1, 10 July 1998, pp.1–23.
3. D.E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.
4. R. Sedgewick, "Implementing Quicksort Programs," *Comm. ACM*, Vol. 21, No. 10, Oct. 1978, pp. 847–857.
5. M. Ben-Or, "Lower Bounds for Algebraic Computation Trees," *Proc. 15th Ann. ACM Symp. Theory of Computing*, ACM Press, New York, 1983, pp. 80–86.
6. P. Raghaven, *Lecture Notes on Randomized Algorithms*, tech. report, IBM Research Division, Yorktown Heights, N.Y., 1990.
7. W. Frazer and A. Mckellar, "Samplesort: A Sampling Approach to Minimal Storage Tree Sorting," *J. ACM*, Vol. 17, No. 3, July 1970, pp. 496–507.
8. G. Blelloch et al., "A Comparison of Sorting Algorithms for the Connection Machine CM-2," *Proc. ACM Symp. Parallel Algorithms and Architectures*, ACM Press, New York, 1991, pp. 3–16.
9. K. Mulmuley, *Computational Geometry: An Introduction through Randomized Algorithms*, Prentice Hall, Upper Saddle River, N.J., 1994.

Joseph JaJa is the director of the University of Maryland Institute of Advanced Computed Studies and a professor of electrical engineering at the university. His research interests are in parallel and distributed computing, combinatorial optimization, and earth-science applications. He received his MS and PhD in applied mathematics from Harvard University. He is a fellow of the IEEE. Contact him at the Inst. for Advanced Computer Studies, A.V. Williams Bldg., Univ. of Maryland, College Park, MD 20742; joseph@umiacs.umd.edu; www.umiacs.umd.edu/~joseph.



THE FFT: AN ALGORITHM THE WHOLE FAMILY CAN USE

The fast Fourier transform is one of the fundamental algorithm families in digital information processing. The author discusses its past, present, and future, along with its important role in our current digital revolution.

> A paper by Cooley and Tukey described a recipe for computing Fourier coefficients of a time series that used many fewer machine operations than did the straightforward procedure ... What lies over the horizon in digital signal processing is anyone's guess, but I think it will surprise us all.
> – Bruce P. Bogert, *IEEE Trans. Audio Electronics*, AU-15, No. 2, 1967, p. 43.

These days, it is almost beyond belief that there was a time before digital technology. It seems almost everyone realizes that the data whizzing over the Internet, bustling through our modems, or crashing into our cell phones is ultimately just a sequence of 0's and 1's—a digital sequence—that magically makes the world the convenient, high-speed place it is today. Much of this magic is due to a family of algorithms that collectively go by the name the *fast Fourier transform*. Indeed, the FFT is perhaps the most ubiquitous algorithm used today to analyze and manipulate digital or discrete data.

My own research experience with various flavors of the FFT is evidence of its wide range of applicability: electroacoustic music and audio-signal processing, medical imaging, image processing, pattern recognition, computational chemistry, error-correcting codes, spectral methods for partial differential equations, and last but not least, mathematics. Of course, I could list many more applications, notably in radar and communications, but space and time restrict. E. Oran Brigham's book is an excellent place to start, especially pages two and three, which contain a (nonexhaustive) list of 77 applications![1]

1521-9615/00/$10.00 © 2000 IEEE

DANIEL N. ROCKMORE
Dartmouth College

History

We can trace the FFT's first appearance, like so much of mathematics, back to Gauss.[2] His interests were in certain astronomical calculations (a recurrent area of FFT application) that dealt with the interpolation of asteroidal orbits from a finite set of equally spaced observations. Undoubtedly, the prospect of a huge, laborious hand calculation provided good motivation to develop

Discrete Fourier transforms

The fast Fourier transform efficiently computes the discrete Fourier transform. Recall that the DFT of a complex input vector of length N, $X = (X(0) \ldots, X(N-1))$, denoted $\hat{X}$, is another vector of length N given by the collection of sums

$$\hat{X}(k) = \sum_{j=0}^{N-1} X(j) W_N^{jk} \tag{1}$$

where $W_N = \exp\left(2\pi\sqrt{-1}/N\right)$. Equivalently, we can view this as the matrix-vector product $F_N \cdot X$, where

$$F_N = \left(\left(W_N^{jk}\right)\right)$$

is the so-called *Fourier matrix*. The DFT is an invertible transform with inverse given by

$$X(j) = \frac{1}{N} \sum_{k=0}^{N-1} \hat{X}(k) W_N^{-jk} . \tag{2}$$

Thus, if computed directly, the DFT would require N^2 operations. Instead, the FFT is an algorithm for computing the DFT in $O(N \log N)$ operations. Note that we can view the inverse as the DFT of the function

$$\frac{1}{N} \hat{X}(-k),$$

so that we can also use the FFT to invert the DFT.

One of the DFT's most useful properties is that it converts circular or cyclic convolution into pointwise multiplication, for example,

$$\widehat{X * Y}(k) = \hat{X}(k) \hat{Y}(k) \tag{3}$$

where

$$X * Y(j) = \sum_{l=0}^{n} X(l) Y(j-l) . \tag{4}$$

Consequently, the FFT gives an $O(N \log N)$ (instead of an N^2) algorithm for computing convolutions: First compute the DFTs of both X and Y, then compute the inverse DFT of the sequence obtained by multiplying pointwise $\hat{X}$ and $\hat{Y}$.

In retrospect, the idea underlying the Cooley-Tukey FFT is quite simple. If $N = N_1 N_2$, then we can turn the 1D equation (Equation 1) into a 2D equation with the change of variables

$$\begin{aligned} j &= j(a,b) = aN_1 + b,\ 0 \le a < N_2,\ 0 \le b < N_1 \\ k &= k(c,d) = cN_2 + d,\ 0 \le c < N_1,\ 0 \le d < N_2 \end{aligned} \tag{5}$$

Using the fact $W_N^{m+n} = W_N^m W_N^n$, it follows quickly from Equation 5 that we can rewrite Equation 1 as

$$\hat{X}(c,d) = \sum_{b=0}^{N_1-1} W_N^{b(cN_2+d)} \sum_{a=0}^{N_2-1} X(a,b) W_{N_2}^{ad} \tag{6}$$

The computation is now performed in two steps. First, compute for each b the inner sums (for all d)

$$\tilde{X}(b,d) = \sum_{a=0}^{N_2-1} X(a,b) W_{N_2}^{ad} \tag{7}$$

which is now interpreted as a subsampled DFT of length N_2. Even if computed directly, at most $N_1 N_2^2$ arithmetic operations are required to compute all of the $\tilde{X}(b,d)$. Finally, we compute $N_1 N_2$ transforms of length N_1:

$$\sum_{b=0}^{N_1-1} W^{b(cN_2+d)} \tilde{X}(b,d) \tag{8}$$

which requires at most an additional $N_1 N_1^2$ operations. Thus, instead of $(N_1 N_2)^2$ operations, this two-step approach uses at most $(N_1 N_2)(N_1 + N_2)$ operations. If we had more factors in Equation 6, then this approach would work even better, giving Cooley and Tukey's result. The main idea is that we have converted a 1D algorithm, in terms of indexing, into a 2D algorithm. Furthermore, this algorithm has the advantage of an in-place implementation, and when accomplished this way, concludes with data reorganized according to the well-known bit-reversal shuffle.

This "decimation in time" approach is one of a variety of FFT techniques. Also notable is the dual approach of "decimation in frequency" developed simultaneously by Gordon Sande, whose paper with W. Morven Gentleman also contains an interesting discussion on memory consideration as it relates to implementational issues.[1] Charles Van Loan's book discusses some of the other variations and contains an extensive bibliography.[2] Many of these algorithms rely on the ability to factor N. When N is prime, we can use a different idea in which the DFT is effectively reduced to a cyclic convolution instead.[3]

References

1. W.M. Gentleman and G. Sande, "Fast Fourier Transforms—For Fun and Profit," *Proc. Fall Joint Computer Conf. AFIPS*, Vol. 29, Spartan, Washington, D.C., 1966, pp. 563–578.
2. C. Van Loan, *Computational Framework for the Fast Fourier Transform*, SIAM, Philadelphia, 1992.
3. C.M. Rader, "Discrete Fourier Transforms When the Number of Data Points is Prime," *Proc. IEEE*, IEEE Press, Piscataway, N.J., Vol. 56, 1968, pp. 1107–1108.

a fast algorithm. Fewer calculations also imply less opportunity for error and therefore lead to numerical stability. Gauss observed that he could break a Fourier series of bandwidth $N = N_1 N_2$ into a computation of N_2 subsampled discrete Fourier transforms of length N_1, which are combined as N_1 DFTs of length N_2. (See the "Discrete Fourier transforms" sidebar for detailed information.) Gauss's algorithm was never published outside of his collected works.

The statistician Frank Yates published a less general but still important version of the FFT in 1932, which we can use to efficiently compute the Hadamard and Walsh transforms.[3] Yates's "interaction algorithm" is a fast technique designed to compute the analysis of variance for a 2^n-factorial design and is described in almost any text on statistical design and analysis of experiments.

Another important predecessor is the work of G.C. Danielson and Cornelius Lanczos, performed in the service of x-ray crystallography, another area for applying FFT technology.[4] Their "doubling trick" showed how to reduce a DFT on $2N$ points to two DFTs on N points using only N extra operations. Today, it's amusing to note their problem sizes and timings: "Adopting these improvements, the approximate times for Fourier analysis are 10 minutes for 8 coefficients, 25 minutes for 16 coefficients, 60 minutes for 32 coefficients, and 140 minutes for 64 coefficients."[4] This indicates a running time of about .37 $N \log N$ minutes for an N-point DFT!

Despite these early discoveries of an FFT, it wasn't until James W. Cooley and John W. Tukey's article that the algorithm gained any notice. The story of their collaboration is an interesting one. Tukey arrived at the basic reduction while in a meeting of President Kennedy's Science Advisory Committee. Among the topics discussed were techniques for offshore detection of nuclear tests in the Soviet Union. Ratification of a proposed United States–Soviet Union nuclear test ban depended on the development of a method to detect the tests without actually visiting Soviet nuclear facilities. One idea was to analyze seismological time-series data obtained from offshore seismometers, the length and number of which would require fast algorithms to compute the DFT. Other possible applications to national security included the long-range acoustic detection of nuclear submarines.

Richard Garwin of IBM was another participant at this meeting, and when Tukey showed him the idea, he immediately saw a wide range of potential applicability and quickly set to getting the algorithm implemented. He was directed to Cooley, and, needing to hide the national security issues, told Cooley that he wanted the code for another problem of interest: the determination of the spin-orientation periodicities in a 3D crystal of He^3. Cooley was involved with other projects, and sat down to program the Cooley-Tukey FFT only after much prodding. In short order, he and Tukey prepared a paper which, for a mathematics or computer science paper, was published almost instantaneously (in six months).[5] This publication, as well as Garwin's fervent proselytizing, did a lot to publicize the existence of this (apparently) new fast algorithm.[6]

The timing of the announcement was such that usage spread quickly. The roughly simultaneous development of analog-to-digital converters capable of producing digitized samples of a time-varying voltage at rates of 300,000 samples per second had already initiated something of a digital revolution. This development also provided scientists with heretofore unimagined quantities of digital data to analyze and manipulate (just as is the case today). The "standard" applications of FFT as an analysis tool for waveforms or for solving PDEs generated a tremendous interest in the algorithm a priori. But moreover, the ability to do this analysis quickly let scientists from new areas try the algorithm without having to invest too much time and energy.

Its effect

It's difficult for me to overstate FFT's importance. Much of its central place in digital signal and image processing is due to the fact that it made working in the frequency domain equally computationally feasible as working in the temporal or spatial domain. By providing a fast algorithm for convolution, the FFT enabled fast, large-integer and polynomial multiplication, as well as efficient matrix-vector multiplication for Toeplitz, circulant, and other kinds of structured matrices. More generally, it plays a key role in most efficient sorts of filtering algorithms. Modifications of the FFT are one approach to fast algorithms for discrete cosine or sine transforms, as well as Chebyshev transforms. In particular, the discrete cosine transform is at the heart of MP3 encoding, which gives life to real-time audio streaming. Last but not least, it's also one of the few algorithms to make it into the movies—I can still recall the scene in *No Way Out* where the image-processing guru declares that he will need to "Fourier transform the image" to help Kevin Costner see the detail in a photograph!

Even beyond these direct technological applications, the FFT influenced the direction of academic research, too. The FFT was one of the first instances of a less-than-straightforward algorithm with a high payoff in efficiency used to compute something important. Furthermore, it raised the natural question, "Could an even faster algorithm be found for the DFT?" (the answer is no[7]), thereby raising awareness of and heightening interest in the subject of lower bounds and the analysis and development of efficient algorithms in general. With respect to Shmuel Winograd's

lower-bound analysis, Cooley writes in the discussion of the 1968 Arden House Workshop on FFT, "These are the beginnings, I believe, of a branch of computer science which will probably uncover and evaluate other algorithms for high speed computers."[8]

Ironically, the FFT's prominence might have slowed progress in other research areas. It provided scientists with a big analytic hammer, and, for many, the world suddenly looked as though it were full of nails—even if this wasn't always so. Researchers sometimes massaged problems that might have benefited from other, more appropriate techniques into a DFT framework, simply because the FFT was so efficient. One example that comes to mind is some of the early spectral-methods work to solve PDEs in spherical geometry. In this case, the spherical harmonics are a natural set of basis functions. Discretization for numerical solutions implies the computation of discrete Legendre transforms (as well as FFTs). Many of the early computational approaches tried instead to approximate these expansions completely in terms of Fourier series, rather than address the development of an efficient Legendre transform.

Even now there are still lessons to learn from the FFT's development. In this day and age, where any new technological idea seems fodder for Internet venture capitalists and patent lawyers, it is natural to ask, "Why didn't IBM patent the FFT?" Cooley explained that because Tukey wasn't an IBM employee, IBM worried that it might not be able to gain a patent. Consequently, IBM had a great interest in putting the algorithm in the public domain. The effect was that then nobody else could patent it either. This did not seem like such a great loss because at the time, the prevailing attitude was that a company made money in hardware, not software. In fact, the FFT was designed as a tool to analyze huge time series, in theory something only supercomputers tackled. So, by placing in the public domain an algorithm that would make time-series analysis feasible, more big companies might have an interest in buying supercomputers (like IBM mainframes) to do their work.

Whether having the FFT in the public domain had the effect IBM hoped for is moot, but it certainly provided many scientists with applications on which to apply the algorithm. The breadth of scientific interests at the Arden workshop (held only two years after the paper's publication) is truly impressive. In fact, the rapid pace of today's technological developments is in many ways a testament to this open development's advantage. This is a cautionary tale in today's arena of proprietary research, and we can only wonder which of the many recent private technological discoveries might have prospered from a similar announcement.

The future FFT

As torrents of digital data continue to stream into our computers, it seems that the FFT will continue to play a prominent role in our analysis and understanding of this river of data. What follows is a brief discussion of future FFT challenges, as well as a few new directions of related research.

Even bigger FFTs

Astronomy continues to be a chief consumer of large FFT technology. The needs of projects like MAP (Microwave Anisotropy Project) or LIGO (Laser InterFerometer Gravitational-Wave Observatory) require FFTs of several (even tens of) gigapoints. FFTs of this size do not fit in the main memory of most machines, and these so-called *out-of-core* FFTs are an active area of research.[9]

As computing technology evolves, undoubtedly, versions of the FFT will evolve to keep pace and take advantage of it. Different kinds of memory hierarchies and architectures present new challenges and opportunities.

Approximate and nonuniform FFTs

For a variety of applications (such as fast MRI), we need to compute DFTs for nonuniformly spaced grid points and frequencies. Multipole-based approaches efficiently compute these quantities in such a way that the running time increases by a factor of

$$\log\left(\frac{1}{\epsilon}\right)$$

where ϵ denotes the approximation's precision.[10] Algebraic approaches based on efficient polynomial evaluation are also possible.[11]

Group FFTs

The FFT might also be explained and interpreted using the language of group representation theory—working along these lines raises some interesting avenues for generalization. One approach is to view a 1D DFT of length N as computing the expansion of a function defined on C_N, the cyclic group of length N (the group of integers mod N) in terms of the basis of irreducible matrix elements of C_N, which are precisely the familiar sampled exponentials: $e_k(m) = \exp(2\pi\sqrt{-1}km/N)$. The FFT is a highly efficient algorithm for computing the expansion in this basis. More generally, a function on

any compact group (cyclic or not) has an expansion in terms of a basis of irreducible matrix elements (which generalize the exponentials from the point of view of group invariance). It's natural to wonder if efficient algorithms for performing this change of basis exist. For example, the problem of efficiently computing spherical harmonic expansions falls into this framework.

The first FFT for a noncommutative finite group seems to have been developed by Alan Willsky in the context of analyzing certain Markov processes.[12] To date, fast algorithms exist for many classes of compact groups.[11] Areas of applications of this work include signal processing, data analysis, and robotics.[13]

Quantum FFTs

One of the first great triumphs of the quantum-computing model is Peter Shor's fast algorithm for integer factorization on a quantum computer.[14] At the heart of Shor's algorithm is a subroutine that computes (on a quantum computer) the DFT of a binary vector representing an integer. The implementation of this transform as a sequence of one- and two-bit quantum gates, now called the quantum FFT, is effectively the Cooley-Tukey FFT realized as a particular factorization of the Fourier matrix into a product of matrices composed as certain tensor products of two-by-two unitary matrices, each of which is a so-called local unitary transform. Similarly, the quantum solution to the Modified Deutsch-Josza problem uses the matrix factorization arising from Yates's algorithm.[15] Extensions of these ideas to the more general group transforms mentioned earlier are currently being explored.

That's the FFT—both parent and child of the digital revolution, a computational technique at the nexus of the worlds of business and entertainment, national security and public communication. Although it's anyone's guess as to what lies over the next horizon in digital signal processing, the FFT will most likely be in the thick of it.

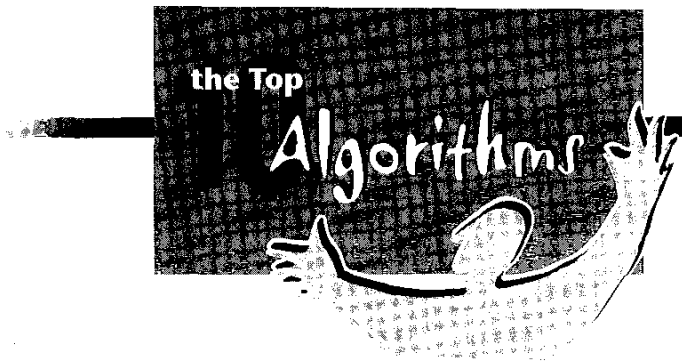
Acknowledgment

Special thanks to Jim Cooley, Shmuel Winograd, and Mark Taylor for helpful conversations. The Santa Fe Institute provided partial support and a very friendly and stimulating environment in which to write this paper. NSF Presidential Faculty Fellowship DMS-9553134 supported part of this work.

References

1. E.O. Brigham, *The Fast Fourier Transform and Its Applications*, Prentice Hall Signal Processing Series, Englewood Cliffs, N.J., 1988.
2. M.T. Heideman, D.H. Johnson, and C.S. Burrus, "Gauss and the History of the Fast Fourier Transform," *Archive for History of Exact Sciences*, Vol. 34, No. 3, 1985, pp. 265–277.
3. F. Yates, "The Design and Analysis of Factorial Experiments," *Imperial Bureau of Soil Sciences Tech. Comm.*, Vol. 35, 1937.
4. G.C. Danielson and C. Lanczos, "Some Improvements in Practical Fourier Analysis and Their Application to X-Ray Scattering from Liquids," *J. Franklin Inst.*, Vol. 233, Nos. 4 and 5, 1942, pp. 365–380 and 432–452.
5. J.W. Cooley and J.W. Tukey, "An Algorithm for Machine Calculation of Complex Fourier Series," *Mathematics of Computation*, Vol. 19, Apr., 1965, pp. 297–301.
6. J.W. Cooley, "The Re-Discovery of the Fast Fourier Transform Algorithm," *Mikrochimica Acta*, Vol. 3, 1987, pp. 33–45.
7. S. Winograd, "Arithmetic Complexity of Computations," *CBMS-NSF Regional Conf. Series in Applied Mathematics, Vol. 33*, SIAM, Philadelphia, 1980.
8. "Special Issue on Fast Fourier Transform and Its Application to Digital Filtering and Spectral Analysis," *IEEE Trans. Audio Electronics*, AU-15, No. 2, 1969.
9. T.H. Cormen and D.M. Nicol, "Performing Out-of-Core FFTs on Parallel Disk Systems," *Parallel Computing*, Vol. 24, No. 1, 1998, pp. 5–20.
10. A. Dutt and V. Rokhlin, "Fast Fourier Transforms for Nonequispaced Data," *SIAM J. Scientific Computing*, Vol. 14, No. 6, 1993, pp. 1368–1393; continued in *Applied and Computational Harmonic Analysis*, Vol. 2, No. 1, 1995, pp. 85–100.
11. D.K. Maslen and D.N. Rockmore, "Generalized FFTs—A Survey of Some Recent Results," *Groups and Computation, II, DIMACS Ser. Discrete Math. Theoret. Comput. Sci., Vol. 28*, Amer. Math. Soc., Providence, R.I., 1997, pp. 183–237.
12. A.S. Willsky, "On the Algebraic Structure of Certain Partially Observable Finite-State Markov Processes," *Information and Control*, Vol. 38, 1978, pp. 179–212.
13. D.N. Rockmore, "Some Applications of Generalized FFTs (An Appendix with D. Healy)," *Groups and Computation II, DIMACS Series on Discrete Math. Theoret. Comput. Sci.*, Vol. 28, American Mathematical Society, Providence, R.I., 1997, pp. 329–369.
14. P.W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer," *SIAM J. Computing*, Vol. 26, No. 5, 1997, pp. 1484–1509.
15. D. Simon, "On the Power of Quantum Computation," *Proc. 35th Annual ACM Symp. on Foundations of Computer Science*, ACM Press, New York, 1994, pp. 116–123.

Daniel N. Rockmore is an associate professor of mathematics and computer science at Dartmouth College, where he also serves as vice chair of the Department of Mathematics. His general research interests are in the theory and application of computational aspects of group representations, particularly to FFT generalizations. He received his BA and PhD in mathematics from Princeton University and Harvard University, respectively. In 1995, he was one of 15 scientists to receive a five-year NSF Presidential Faculty Fellowship from the White House. He is a member of the American Mathematical Society, the IEEE, and SIAM. Contact him at the Dept. of Mathematics, Bradley Hall, Dartmouth College, Hanover, NH 03755; rockmore@cs.dartmouth.edu; www.cs.darthmouth.edu/~rockmore.



INTEGER RELATION DETECTION

Practical algorithms for integer relation detection have become a staple in the emerging discipline of experimental mathematics—using modern computer technology to explore mathematical questions. After briefly discussing the problem of integer relation detection, the author describes several recent, remarkable applications of these techniques in both mathematics and physics.

For many years, researchers have dreamt of a facility that lets them recognize a numeric constant in terms of the mathematical formula that it satisfies. With the advent of efficient integer relation detection algorithms, that time has arrived.

Integer relation detection

Let $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ be a vector of real or complex numbers. $\mathbf{x}$ is said to possess an integer relation if there exist integers a_i (not all zero), such that $a_1x_1 + a_2x_2 + \ldots + a_nx_n = 0$. An integer relation algorithm is a practical computational scheme that can recover the vector of integers a_i (if it exists), or can produce bounds within which no integer relation exists.

The problem of finding integer relations is not new. Euclid (whose Euclidean algorithm solves this problem in the case $n = 2$) first studied it about 300 BC. Leonard Euler, Karl Jacobi, Jules Poincaré, and other well-known 18th, 19th, and 20th century mathematicians pursued solutions for a larger n. The first integer relation algorithm for general n was discovered in 1977 by Helaman Ferguson and Rodney Forcade.[1]

There is a close connection between integer relation detection and integer-lattice reduction. Indeed, one common solution to the integer relation problem is to apply the Lenstra-Lenstra-Lovasz lattice-reduction algorithm. However, there are some difficulties with this approach, notably the somewhat arbitrary selection of a required multiplier—if it is too small or too large, the LLL solution will not determine the desired integer relation. The HJLS algorithm,[2] which is based on the LLL algorithm, addresses these dif-

1521-9615/00/$10.00 © 2000 IEEE

DAVID H. BAILEY
Lawrence Berkeley Laboratory

ficulties, but, unfortunately, it suffers from numerical instability and thus fails in many cases of practical interest.

The PSLQ algorithm

The most effective algorithm for integer relation detection is Ferguson's recently discovered PSLQ algorithm;[3] the name derives from its usage of a partial sum-of-squares vector and an LQ (lower-diagonal-orthogonal) matrix factorization. In addition to possessing good numerical stability, PSLQ finds a relation in a polynomially bounded number of iterations. A simple statement of the PSLQ algorithm, equivalent to the original formulation, is as follows: Let $\mathbf{x}$ be the n-long input real vector, and let nint denote the nearest integer function. Select $\gamma \geq \sqrt{4/3}$ Then perform the following operations.

1. Set the $n \times n$ matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ to the identity.
2. Compute the n-long vector $\mathbf{s}$ as
$$\mathbf{s}_k := \sqrt{\sum_{j=k}^{n} \mathbf{x}_j^2}$$
and set $\mathbf{y}$ to the $\mathbf{x}$ vector, normalized by $\mathbf{s}_1$.
3. Compute the initial $n \times (n-1)$ matrix $\boldsymbol{H}$ as $\boldsymbol{H}_{ij} = 0$ if $i < j$, $\boldsymbol{H}_{jj} := \mathbf{s}_{j+1}/\mathbf{s}_j$, and $\boldsymbol{H}_{ij} := -\mathbf{y}_i\mathbf{y}_j/(\mathbf{s}_j\mathbf{s}_{j+1})$ if $i > j$.
4. Reduce $\boldsymbol{H}$: For $i := 2$ to n: for $j := i - 1$ to 1 step -1 : set $t :=$ nint $(\boldsymbol{H}_{ij}/\boldsymbol{H}_{jj})$; and $\mathbf{y}_j := \mathbf{y}_j + t\mathbf{y}_i$; for $k := 1$ to j: set $\boldsymbol{H}_{ik} := \boldsymbol{H}_{ik} - t\boldsymbol{H}_{jk}$; endfor; for $k := 1$ to n: set $\boldsymbol{A}_{ik} := \boldsymbol{A}_{ik} - t\boldsymbol{A}_{jk}$ and $\boldsymbol{B}_{kj} := \boldsymbol{B}_{kj} + t\boldsymbol{B}_{ki}$; endfor; endfor; endfor.

Iterate until an entry of $\mathbf{y}$ is within a reasonable tolerance of 0, or until precision is exhausted:

1. Select m such that $\gamma^i |\boldsymbol{H}_{ii}|$ is maximal when $i = m$.
2. Exchange the entries of $\mathbf{y}$ indexed m and $m + 1$, the corresponding rows of $\boldsymbol{A}$ and $\boldsymbol{H}$, and the corresponding columns of $\boldsymbol{B}$.
3. Remove the corner on $\boldsymbol{H}$ diagonal: If $m \leq n - 2$, set
$$t_0 := \sqrt{\boldsymbol{H}_{mm}^2 + \boldsymbol{H}_{m,m+1}^2}, \qquad t_1 := \boldsymbol{H}_{mm} / t_0$$
and $t_2 := \boldsymbol{H}_{m,m+1} / t_0$; for $i := m$ to n: set $t_3 := \boldsymbol{H}_{im}$, $t_4 := \boldsymbol{H}_{i,m+1}$, $\boldsymbol{H}_{im} := t_1 t_3 + t_2 t_4$ and $\boldsymbol{H}_{i,m+1} := -t_2 t_3 + t_1 t_4$; endfor; endif.
4. Reduce $\boldsymbol{H}$: For $i := m + 1$ to n: for $j :=$ min$(i - 1, m + 1)$ to 1 step -1: set $t :=$ nint $(\boldsymbol{H}_{ij} / \boldsymbol{H}_{jj})$ and $\mathbf{y}_j := \mathbf{y}_j + t\mathbf{y}_i$; for $k := 1$ to j: set $\boldsymbol{H}_{ik} := \boldsymbol{H}_{ik} - t\boldsymbol{H}_{jk}$; endfor; for $k := 1$ to n: set $\boldsymbol{A}_{ik} := \boldsymbol{A}_{ik} - t\boldsymbol{A}_{jk}$ and $\boldsymbol{B}_{kj} := \boldsymbol{B}_{kj} + t\boldsymbol{B}_{ki}$; endfor; endfor; endfor.
5. Norm bound: Compute $M := 1/\max_j |\boldsymbol{H}_{jj}|$. Then there can exist no relation vector whose Euclidean norm is less than M.

Upon completion, the desired relation is found in column $\boldsymbol{B}$ corresponding to the 0 entry of $\mathbf{y}$. (Some efficient "multilevel" implementations of PSLQ, as well as a variant of PSLQ that is well-suited for highly parallel computer systems, are described in a recent paper.[4])

Almost all applications of an integer relation algorithm such as PSLQ require high-precision arithmetic. The 64-bit IEEE floating-point arithmetic available on current computer systems can reliably recover only a very small class of relations. In general, if we want to recover a relation of length n with coefficients of maximum size d digits, then the input vector $\mathbf{x}$ must be specified to at least nd digits, and we must employ floating-point arithmetic accurate to at least nd digits. Maple and Mathematica include multiple precision arithmetic facilities. There are several freeware multiprecision software packages.[5–7]

Finding algebraic relations using PSLQ

One PSLQ application in the field of mathematical number theory is to determine whether or not a given constant α, whose value can be computed to high precision, is algebraic of some degree n or less. First compute the vector $\mathbf{x} = (1, \alpha, \alpha^2, \ldots, \alpha^n)$ to high precision, and then apply an integer relation algorithm. If you find a relation for $\mathbf{x}$, then this relation vector is precisely the set of integer coefficients of a polynomial α satisfies.

One of the first results of this sort was the identification of the constant $B_3 = 3.54409035955\ldots$. B_3 is the third bifurcation point of the logistic map $z_{k+1} = rz_k(1 - z_k)$, which exhibits period doubling shortly before the onset of chaos.[5] To be precise, B_3 is the smallest value of the parameter r such that successive iterates z_k exhibit eight-way periodicity instead of four-way periodicity. Computations using a predecessor algorithm to PSLQ found that B_3 is a root of the polynomial $0 = 4913 + 2108t^2 - 604t^3 - 977t^4 + 8t^5 + 44t^6 + 392t^7 - 193t^8 - 40t^9 + 48t^{10} - 12t^{11} + t^{12}$.

Recently, British physicist David Braodhurst identified $B_4 = 3.564407268705\ldots$, the fourth bifurcation point of the logistic map, using PSLQ.[4] It had been conjectured that B_4 might satisfy a 240-degree polynomial, and further analysis sug-

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k}\left[\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6}\right]$$

Figure 1. The formula found by PSLQ, which can be used to compute hexadecimal digits of π.

$$\pi^2 = \frac{2}{27}\sum_{k=0}^{\infty} \frac{1}{729^k}\left[\frac{243}{(12k+1)^2} - \frac{405}{(12k+2)^2} - \frac{81}{(12k+4)^2} - \frac{27}{(12k+5)^2} - \frac{72}{(12k+6)^2} - \frac{9}{(12k+7)^2} - \frac{9}{(12k+8)^2} - \frac{5}{(12k+10)^2} + \frac{1}{(12k+11)^2}\right]$$

Figure 2. A base-3 formula for π^2.

$$\sum_{k=1}^{\infty}\left(1+\frac{1}{2}+\mathrm{L}+\frac{1}{k}\right)^2 (k+1)^{-4} = \frac{37}{22{,}680}\pi^6 - \zeta^2(3)$$

$$\sum_{k=1}^{\infty}\left(1+\frac{1}{2}+\mathrm{L}+\frac{1}{k}\right)^3 (k+1)^{-6} = \zeta^3(3) + \frac{197}{24}\zeta(9) + \frac{1}{2}\pi^2\zeta(7) - \frac{11}{120}\pi^4\zeta(5) - \frac{37}{7560}\pi^6\zeta(3)$$

$$\sum_{k=1}^{\infty}\left(1-\frac{1}{2}+\mathrm{L}+(-1)^{k+1}\frac{1}{k}\right)^2 (k+1)^{-3} = 4\mathrm{Li}_5\left(\frac{1}{2}\right) - \frac{1}{30}\ln^5(2) - \frac{17}{32}\zeta(5) - \frac{11}{720}\pi^4\ln(2) + \frac{7}{4}\zeta(3)\ln^2(2) + \frac{1}{18}\pi^2\ln^3(2) - \frac{1}{8}\pi^2\zeta(3)$$

Figure 3. Some multiple-sum identities found by PSLQ. $\zeta(t) = \sum_{j=1}^{\infty} j^{-t}$ is the Riemann zeta function, and $\mathrm{Li}_n(x) = \sum_{j=1}^{\infty} x^j j^{-n}$ denotes the polylogarithm function.

gested that the constant $\alpha = -B_4(B_4 - 2)$ might satisfy a 120-degree polynomial. To test this hypothesis, Broadhurst applied a PSLQ program to the 121-long vector $(1, \alpha, \alpha^2, \ldots, \alpha^{120})$. Indeed, a relation was found, but the computation required 10,000-digit arithmetic. The recovered integer coefficients descend monotonically from $257^{30} \approx 1.986 \times 10^{72}$ to 1.

A new formula for π

Throughout the centuries, mathematicians have assumed that there is no shortcut to computing just the nth digit of π. Thus, it came as no small surprise when such an algorithm was recently discovered.[8] In particular, this simple scheme lets us compute the nth hexadecimal (or binary) digit of π without computing any of the first $n - 1$ digits, without using multiple-precision arithmetic software, and at the expense of very little computer memory. The one-millionth hex digit of π can be computed in this manner on a current-generation personal computer in only about 60 seconds runtime.

This scheme for computing the nth digit of π is based on a formula that was discovered in 1996 using PSLQ (see Figure 1). Similar base-2 formulas exist for other mathematical constants,[8,9] and Broadhurst recently has obtained some base-3 formulas,[10] including an identity for π^2 (see Figure 2), using PSLQ.

Identifying multiple-sum constants

In the course of researching multiple sums, scientists recently found many results using PSLQ, such as those shown in Figure 3. After computing the numerical values of these constants, a PSLQ program determined if a given constant satisfied an identity of a conjectured form. These efforts produced numerous empirical evaluations and suggested general results[11]—for which scientists eventually found elegant proofs.[12,13]

In another application to mathematical number theory, several researchers have used PSLQ to investigate sums of the form

$$S(k) = \sum_{n>0} \frac{1}{n^k \binom{2n}{n}}.$$

For small k, these constants satisfy simple identities, such as $S(4) = 17\pi^4/3240$. Thus, researchers have sought generalizations of these formulas for $k > 4$. As a result of PSLQ computations, several researchers have evaluated the constants $\{S(k) \mid k = 5 \ldots 20\}$ in terms of multiple zeta values,[14] which are defined by

$$\zeta(s_1, s_2, \ldots, s_r) = \sum_{k_1>k_2>\cdots>k_r>0} \frac{1}{k_1^{s_1} k_2^{s_2} \cdots k_r^{s_r}}$$

and multiple Clausen values[15] of the form

$$M(a,b) = \sum_{n_1>n_2>\cdots>n_t>0} \frac{\sin(n_1\pi/3)}{n_1^a} \prod_{j=1}^{b} \frac{1}{n_j}.$$

Figure 4 gives a sample evaluation.

The evaluation of $S(20)$ is an integer relation problem with $n = 118$, requiring 5,000-digit arithmetic. (The full solution is given in a recent paper.[4])

$$S(9) = \pi\left[2M(7,1) + \frac{8}{3}M(5,3) + \frac{8}{9}\zeta(2)M(5,1)\right] - \frac{13,921}{216}\zeta(9) + \frac{6,211}{486}\zeta(7)\zeta(2) + \frac{8,101}{648}\zeta(6)\zeta(3) + \frac{331}{18}\zeta(5)\zeta(4) - \frac{8}{9}\zeta^3(3)$$

Figure 4. A sample evaluation of an of an Apery sum constant using PSLQ.

Connections to quantum field theory

In a surprising recent development, Broadhurst found an intimate connection between these multiple sums and constants resulting from evaluating Feynman diagrams in quantum field theory.[16,17] In particular, the renormalization procedure (which removes infinities from the perturbation expansion) involves multiple zeta values. Broadhurst used PSLQ to find formulas and identities involving these constants. As before, a fruitful theory emerged, including a large number of both specific and general results.[14,18]

More generally, we can define Euler sums by[14]

$$\zeta\begin{pmatrix} s_1, s_2, \ldots, s_r \\ \sigma_1, \sigma_2, \ldots, \sigma_r \end{pmatrix} = \sum_{k_1>k_2>\cdots>k_r>0} \frac{\sigma_1^{k_1}}{k_1^{s_1}} \frac{\sigma_2^{k_2}}{k_2^{s_2}} \cdots \frac{\sigma_r^{k_r}}{k_r^{s_r}}$$

where $\sigma_j = \pm 1$ are signs and $s_j > 0$ are integers. When all the signs are positive, we have a multiple zeta value. Constants with alternating signs appear in problems such as computing an electron's magnetic moment.

Broadhurst conjectured that the dimension of the Euler sum spaces with weight $w = \Sigma s_j$ is the Fibonacci number $F_{w+1} = F_w + F_{w-1}$, with $F_1 = F_2 = 1$. I've obtained complete reductions of all Euler sums to a basis of size F_{w+1} with PSLQ at weights $w \le 9$. At weights $w = 10$ and $w = 11$, Broadhurst stringently tested the conjecture by applying PSLQ in more than 600 cases. At weight $w = 11$, such tests involve solving integer relations of size $n = F_{12} + 1 = 145$.[4]

Some recent quantum field theory results are even more remarkable. Broadhurst has now shown, using PSLQ, that in each of 10 cases with unit or 0 mass, the finite part of the scalar, three-loop, tetrahedral vacuum Feynman diagram reduces to four-letter "words." These words represent iterated integrals in an alphabet of seven "letters" comprising the one-forms $\Omega := dx/x$ and $w_k := dx/(\lambda^{-k} - x)$, where $\lambda := (1 + \sqrt{-3})/2$ is the primitive sixth root of unity, and k runs from 0 to 5.[10]

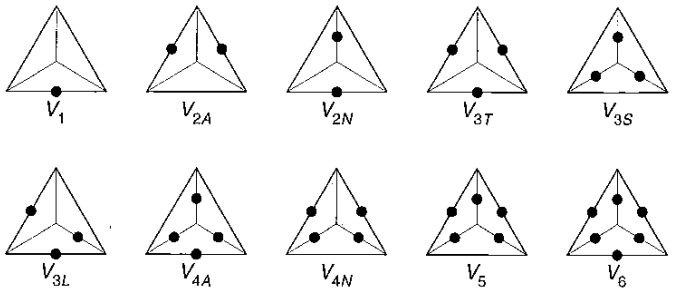


Figure 5. The 10 tetrahedral cases.

Figure 5 shows these 10 cases. In the diagrams, dots indicate particles with nonzero rest mass. Table 1 gives the formulas that Broadhurst found, using PSLQ, for the corresponding constants. In the table, $U = \Sigma_{j>k>0}(-1)^{j+k}/(j^3k)$; $V = \Sigma_{j>k>0}(-1)^j \cos(2\pi k/3)/(j^3k)$; and the constant $C = \Sigma_{k>0} \sin(\pi k/3)/k^2$.

Table 1. The formulas found by PSLQ for the 10 cases in Figure 5.

Tetrahedral cases	PSLQ formulas
V_1	$6\zeta(3) + 3\zeta(4)$
V_{2A}	$6\zeta(3) - 5\zeta(4)$
V_{2N}	$6\zeta(3) - 13/2\ \zeta(4) - 8\ U$
V_{3T}	$6\zeta(3) - 9\ \zeta(4)$
V_{3S}	$6\zeta(3) - 11/2\ \zeta(4) - 4\ C^2$
V_{3L}	$6\zeta(3) - 15/4\ \zeta(4) - 6\ C^2$
V_{4A}	$6\zeta(3) - 77/12\ \zeta(4) - 6\ C^2$
V_{4N}	$6\zeta(3) - 14\ \zeta(4) - 16\ U$
V_5	$6\zeta(3) - 469/27\ \zeta(4) + 8/3\ C^2 - 16V$
V_6	$6\zeta(3) - 13\zeta(4) - 8U - 4C^2$

Using integer relation algorithms, researchers have discovered numerous new facts of mathematics and physics, and these discoveries have in turn led to valuable new insights. This process, often called "experimental mathematics"—namely, the utilization of modern computer technology in the discovery of new mathematical principles—is expected to play a much wider role in both pure and applied mathematics during the next century. In particular, when fast integer relation detection facilities are integrated into widely used mathematical computing environments, such as Mathematica and Maple, we can expect numerous more discoveries of the sort described in this article, and, possibly, new applications that we cannot yet foresee.

Acknowledgments

This work was supported by the Director of the Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the US Department of Energy, under contract DE-AC03-76SF00098.

References

1. H.R.P. Ferguson and R.W. Forcade, "Generalization of the Euclidean Algorithm for Real Numbers to All Dimensions Higher Than Two," *Bulletin Am. Mathematical Soc.*, Vol. 1, No. 6, Nov. 1979, pp. 912–914.
2. J. Hastad et al., "Polynomial Time Algorithms for Finding Integer Relations Among Real Numbers," *SIAM J. Computing*, Vol. 18, No. 5, Oct. 1989, pp. 859–881.
3. H.R.P. Ferguson, D.H. Bailey, and S. Arno, "Analysis of PSLQ, An Integer Relation Finding Algorithm," *Mathematics of Computation*, Vol. 68, No. 225, Jan. 1999, pp. 351–369.
4. D.H. Bailey and D. Broadhurst, "Parallel Integer Relation Detection: Techniques and Applications," submitted for publication, 1999; also available at www.nersc.gov/~dhbailey, online paper 34 (current Nov. 1999).
5. D.H. Bailey, "Multiprecision Translation and Execution of Fortran Programs," *ACM Trans. Mathematical Software*, Vol. 19, No. 3, 1993, pp. 288–319.
6. D.H. Bailey, "A Fortran-90 Based Multiprecision System," *ACM Trans. Mathematical Software*, Vol. 21, No. 4, 1995, pp. 379–387.
7. S. Chatterjee and H. Harjono, "MPFUN++: A Multiple Precision Floating Point Computation Package in C++," University of North Carolina, Sept. 1998; www.cs.unc.edu/Research/HARPOON/mpfun++ (current Nov. 1999).
8. D.H. Bailey, P.B. Borwein and S. Plouffe, "On The Rapid Computation of Various Polylogarithmic Constants," *Mathematics of Computation*, Vol. 66, No. 218, 1997, pp. 903–913.
9. D.J. Broadhurst, "Polylogarithmic Ladders, Hypergeometric Series and the Ten Millionth Digits of $\zeta(3)$ and $\zeta(5)$," preprint, Mar. 1998; xxx.lanl.gov/abs/math/9803067 (current Nov. 1999).
10. D.J. Broadhurst, "Massive 3-loop Feynman Diagrams Reducible to SC* Primitives of Algebras of the Sixth Root of Unity," preprint, March 1998, to appear in *European Physical J. Computing*; xxx.lanl.gov/abs/hep-th/9803091 (current Nov. 1999).
11. D.H. Bailey, J.M. Borwein, and R. Girgensohn, "Experimental Evaluation of Euler Sums," *Experimental Mathematics*, Vol. 4, No. 1, 1994, pp. 17–30.
12. D. Borwein and J.M. Borwein, "On an Intriguing Integral and Some Series Related to $\zeta(4)$," *Proc. American Mathematical Soc.*, Vol. 123, 1995, pp. 111–118.
13. D. Borwein, J.M. Borwein, and R. Girgensohn, "Explicit Evaluation of Euler Sums," *Proc. Edinburgh Mathematical Soc.*, Vol. 38, 1995, pp. 277–294.
14. J.M. Borwein, D.M. Bradley, and D.J. Broadhurst, "Evaluations of k-fold Euler/Zagier Sums: A Compendium of Results for Arbitrary k," *Electronic J. Combinatorics*, Vol. 4, No. 2, #R5, 1997.
15. J.M. Borwein, D.J. Broadhurst, and J. Kamnitzer, "Central Binomial Sums and Multiple Clausen Values (with Connections to Zeta Values)," No. 137, 1999; www.cecm.sfu.ca/preprints/1999pp.html (current Nov. 1999).
16. D.J. Broadhurst, J.A. Gracey, and D. Kreimer, "Beyond the Triangle and Uniqueness Relations: Non-Zeta Counterterms at Large N from Positive Knots," *Zeitschrift für Physik*, Vol. C75, 1997, pp. 559–574.
17. D.J. Broadhurst and D. Kreimer, "Association of Multiple Zeta Values with Positive Knots via Feynman Diagrams up to 9 Loops," *Physics Letters*, Vol. B383, 1997, pp. 403–412.
18. J.M. Borwein et al., "Combinatorial Aspects of Multiple Zeta Values," *Electronic J. Combinatorics*, Vol. 5, No. 1, #R38, 1998.

David H. Bailey is the chief technologist of the National Energy Research Scientific Computing Center at the Lawrence Berkeley National Laboratory. His research has included studies in highly parallel computing, numerical algorithms, fast Fourier transforms, multiprecision computation, supercomputer performance, and computational number theory. He received his BS in mathematics from Brigham Young University and his PhD in mathematics from Stanford University. In 1993, he received the Sidney Fernbach Award, which is bestowed by the IEEE Computer Society for contributions to the field of high-performance computing. He has also received the Chauvenet Prize and the Merten Hasse Prize from the Mathematical Association of America and the H. Julian Allen Award from NASA. He just served as the technical papers chair for Supercomputing '99 and as the chair of the 1999 Gordon Bell Prize committee. Contact him at the Lawrence Berkeley Laboratory, MS 50B-2239, Berkeley, CA 94720; dhbailey@lbl.gov.

THE FAST MULTIPOLE ALGORITHM

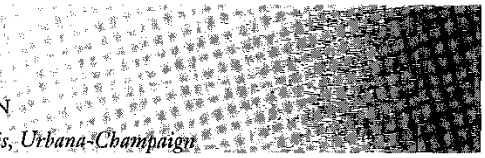
Accurate computation of the mutual interactions of N *particles through electrostatic or gravitational forces has impeded progress in many areas of simulation science. The fast multipole algorithm provides an efficient scheme for reducing computational complexity.*

Problems that involve computing the mutual interactions within large sets of particles pervade many branches of science and engineering. When the particles interact through electrostatic (Coulomb) or gravitational (Newton) potentials, the long range of the resulting forces creates a computational headache. This is because forces arise for all pairs of charges (or masses) in such systems. Because the number of pairs increases quadratically with the number of particles N included in a simulation, the computational complexity of simulations carried out without well-designed computational strategies is said to be of order $O(N^2)$. This quickly renders computational studies of the so-called N-body problem practically impossible as systems increase in size to levels relevant for realistic problems. Thus, from a computational point of view, the problems a biophysicist encounters simulating ion conduction through cellular membranes are essentially the same as those an astrophysicist encounters simulating accretion of planetary systems.

Examples from biomedicine illustrate the dilemma $O(N^2)$ algorithms pose for computing the Coulomb forces that arise in atomic simulation. All biomolecules carry partial charges centered around their atoms, resulting in Coulomb interactions. The overall biomolecular charge is usually small—local neutrality limits the effect of such forces to some degree, although accurate simulations cannot neglect them. When unbalanced charges or dipole moments arise in molecular systems, long-range Coulomb forces can dominate biomolecular-system arrangement and dynamics, and faithful descriptions of these forces are essential.

An example of this is DNA in biological cells. DNA contains two negatively charged phosphates for each pair of bases that establish the genetic code. Positive ions (such as Na+) that exist in physiological fluids neutralize these negative charges, but positive ions spread diffusively so that Coulomb forces remain strong in spite of them. Lipid bilayers that form membranes and

1521-9615/00/$10.00 © 2000 IEEE



JOHN BOARD
Duke University
KLAUS SCHULTEN
University of Illinois, Urbana-Champaign

are the staging ground for many biomolecular processes are ubiquitous in biological cells. The lipids contain head groups that may be charged but invariably feature a strong electric dipole. Because lipids are more or less aligned in parallel in membranes, the dipole moments sum rather than cancel each other. Water molecules that also carry strong dipole moments are always present near lipid bilayers and tend to counterbalance the lipid dipoles, but do so only to a limited degree (strong Coulomb forces remain). These forces imply the presence of strong electric fields across biological membranes that need to be accurately described if simulations of membrane processes are to be meaningful.

The computational complexity of naive Coulomb solvers severely constrains progress: systems with at most a few thousand atoms can be studied with $O(N^2)$ solvers on very fast computers. However, at least 200 lipids must be included in a lipid bilayer simulation to have an acceptable volume-to-surface ratio that can help model bulk properties. Additionally, at least one water molecule layer must be added to the simulation on each side of the bilayer. Thus, the smallest simulated volume is about 100Å × 100Å, which is filled with over 30,000 atoms of lipids and water. If proteins are embedded in such a system, the simulated system's size can quickly reach 200,000 atoms. The DNA-coded information is controlled through proteins that can recognize DNA sequences. Simulations that seek to understand this control must include a segment of DNA, proteins, and the ubiquitous water along with physiological ions. The smallest system of this type contains well over 30,000 atoms. Modern biology poses many exciting challenges that require simulations of systems with hundreds of thousands to millions of atoms—viral infection, the conversion of light energy into chemical energy at photosynthetic membranes of bacteria or plants, the description of DNA and proteins in chromosomes, or the transcription of genetic information into proteins at ribosomes. These challenges motivated computational scientists to seek practical solutions to the N-body problems inherent in Coulomb and gravitational interactions. The first such algorithm that reduced the computational effort to $O(N)$ was Vladimir Rokhlin and Leslie Greengard's fast multipole algorithm (FMA).[1]

Historical context

Rokhlin and Greengard's work arguably provided the first numerically defensible method for reducing the N-body problem's computational complexity, but they weren't the first to work on the problem. Some form of the N-body problem is at the core of many computational problems, but astrophysical simulations of gravitating bodies and the evaluation of electrostatic interactions between charged particles (such as the atoms in a biomolecular simulation) have motivated much of the reported work.

Before the development of the FMA and related algorithms, those running N-body simulations had two choices: either attack the $O(N^2)$ complexity of the problem with brute force or truncate the potential's infinite range to a more manageable but less accurate value. Astrophysicists tended to gravitate to the former solution, the Grape (*gra*vity *pipe*) project in Japan being the most notable effort. The project built a series of massively parallel machines to rapidly evaluate gravitational interactions, culminating in Grape-4;[2] over 1,000 processors handled the computational complexity. Even so, $O(N^2)$ eventually overwhelms any number of processors, so the maximum problem size is limited.

Rokhlin and Greengard arguably provided the first numerically defensible method for reducing the N-body problem's computational complexity.

Classical molecular simulation has many more complications than the astrophysical case in that many other forces act between atoms in addition to the $1/r$ interaction (forces constraining bond lengths and angles, van der Waals forces, and more). Nonetheless, the Coulomb interaction's infinite range makes it the computationally dominant factor in such simulations. Researchers in the molecular simulation community also took the brute-force approach with Coulomb's law, with custom machines[3,4] dedicated to molecular dynamics simulation. As in the gravitational case, however, complexity trumps hardware—time limited these machines as to what size of biomolecular system they could study. The Illinois 60-processor transputer-based machine ran for over two years to obtain a successful simulation of a lipid bilayer of 200 lipids with 32,000 atoms that included water. Although this was a groundbreaking simulation at the time (it demonstrated a high degree of accuracy in comparison with experimental observation), the effort required to execute it was not easily repeatable, and the approach could not be scaled to the systems 10 to 1,000 times larger that computational biologists wanted to study.

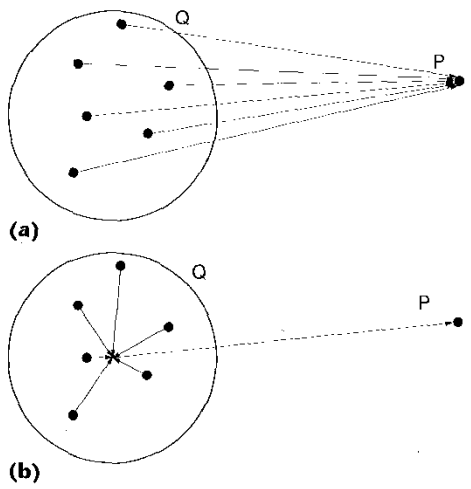


Figure 1. Rather than interact with each of the distant particles individually (a), the particle at P can interact with an approximate aggregation of the distant group, such as its centroid-located net charge (b).

Most researchers in the molecular simulation community took an easier way out. By truncating the Coulomb interaction's infinite range to 8Å–20Å or so, they reduced the $O(N^2)$ complexity to a linear problem over a fixed, finite neighborhood easily addressed with neighbor lists and other techniques. Of course, this ignored all interactions beyond the cut-off radius, thus reducing the simulation's fidelity. One argument is that because the many other forces involved in a molecular simulation are modeled at best to a few significant figures of accuracy by largely empirically determined linear or quadratic expressions, being completely faithful to the electrostatics isn't as important. Although some problems can be successfully studied this way, truncation clearly has limitations that make it unsuitable for many simulations, especially the membrane and DNA simulations discussed earlier.

The method

Matthew Pincus and Harold Scheraga suggested the first basic idea behind the FMA in the biophysical literature in 1977.[5] They described, and others later implemented, an approximation where the effect of a group of distant, charged particles on a particle of interest is described by replacing the entire distant group with a single pseudoparticle that embodies the group's properties. The key properties of the distant group of particles are its net charge, its dipole moment, and its quadrupole and higher multipole moments. Mathematically, these properties are conveniently represented by the multipole expansion of the distant group. The infinite but rapidly converging multipole series expansion is truncated at a convenient number of terms, in practice usually three to eight, with more terms giving higher accuracy in the approximation. The particle of interest can now interact with the entire distant group by instead interacting with the single multipole expansion that represents the group, instead of with all the distant group's individual members (see Figure 1).

The second key idea of the FMA and related methods is to use a hierarchical decomposition of space to rationally separate the simulation region into areas that are suitably distant from each other to invoke the multipole expansion approximation. In Figure 2's oct-tree decomposition, ever-larger regions of space that represent increasing numbers of particles can interact through individual multipole expansions at increasing distances. The first practical algorithms[6,7] combined the two ideas for use in astrophysical simulations. Both methods have a computational complexity of $O(N \log N)$ in the number of particles N, an improvement over $O(N^2)$. Additionally, the monopole moment is large in the Newtonian case, because all mass is positive. Thus, the monopole term alone, and certainly the first two terms of the multipole series (monopole plus dipole terms), computed the gravitational interactions quite accurately.

The electrostatic problem is complicated because charge distributions' monopole moments are usually small; positive and negative charges roughly cancel each other out. By adding more terms to the multipole series, we can adapt the Barnes-Hut algorithm to the electrostatic case, but the resulting method is difficult to rigorously analyze for its numerical robustness. Enter Greengard and Rokhlin's fast multipole algorithm in 1987. Their introduction of a *local expansion* further reduced the procedure's complexity from $O(N \log N)$ to $O(N)$, at least in certain important cases. Additionally, their method's machinery was amenable to a rigorous numerical analysis that bounded the method's error, removing the somewhat ad hoc feel of the earlier methods. We can now confidently determine how many terms are required in a multipole expansion to achieve a certain guaranteed level of accuracy.

The local expansion idea is critical to their improved scheme. Now, distant groups of particles interact with entire groups of target particles at once: both the distant group and the target group are represented by multipole expansions. An interaction between these groups essentially

involves a convolution of the coefficient arrays describing their respective multipole expansions.

An additional and crucial benefit of Greengard and Rokhlin's approach is that it is not restricted to the $1/r$ potential. Multipole-like formulations can be constructed for any power law potential and for other functional forms; we can apply the same complexity-reducing FMA mechanics to these cases with similar results. The $1/r$ case enjoys some special properties that simplify its analysis, but extension of these methods to other classes of potential functions is an active and fruitful area of current work.

Researchers are studying very large astrophysical simulations with hybrids of the FMA and the earlier Barnes-Hut scheme. In the biophysical-simulation world, the Ewald summation method is an additional competitor. Since the development of the FMA, scientists have created various fast versions of the nearly 80-year-old Ewald method that are faster than multipole codes in some cases, although their error behavior is harder to quantify. The Ewald codes also handle periodic boundary conditions automatically; FMA-derived codes can be extended to this case with extra effort. Nonetheless, FMA and its offspring remain important, and the newest formulations promise to again challenge Ewald codes for the title of fastest electrostatic solver.

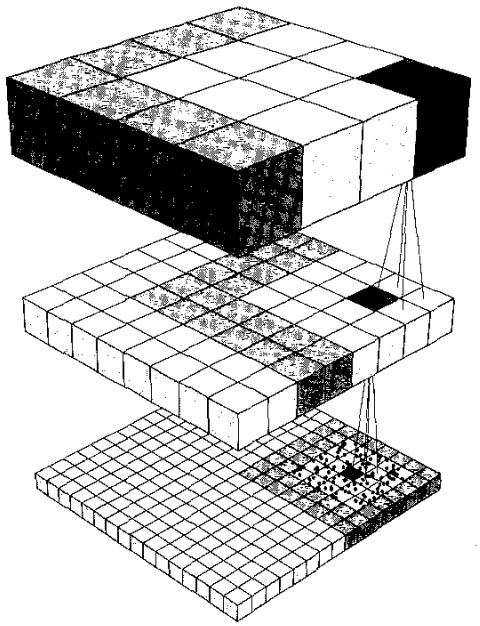


Figure 2. Multipole algorithms use hierarchical spatial decomposition to separate the simulation space into regions sufficiently far apart from each other to interact according to the approximate method in Figure 1. At increasing distances, ever-larger regions of space can be lumped into single approximations.

John Board is the Bass Associate Professor and associate chair in the Electrical and Computer Engineering Department at Duke University. He is also director of the Center for Computational Science and Engineering at the university. His research interests include the application of high-performance computing techniques to problems in the biological and physical sciences. He received his MS in electrical engineering from Duke, and his D.Phil in theoretical physics from Oxford University. Contact him at the Dept. of Electrical and Computer Eng., Duke Univ., Box 90291, Durham, NC 27708; jab@ee.duke.edu.

Klaus Schulten is Swanlund Professor of physics and director of the Theoretical Biophysics Group at the Beckman Institute at the University of Illinois, Urbana-Champaign. His research interests focus on the architecture, functions, and mechanisms of molecular aggregates in biological cells. In his research he employs large-scale molecular dynamics simulations as well as quantum mechanical and statistical mechanical descriptions. His research group develops molecular graphics, simulation, and collaboratory software. He received his Diplom in physics at the University of Münster, Germany, his PhD in chemical physics at Harvard, and his habilitation degree at the University of Göttingen, Germany. Contact him at the Beckman Inst., Univ. of Illinois, Urbana, IL 61801; kschulte@ks.uiuc.edu.

References

1. L. Greengard and V. Rokhlin, "A Fast Algorithm for Particle Simulation," *J. Computational Physics*, Vol. 73, No. 2, Dec. 1987, pp. 325–348.
2. J. Makino and T. Makoto, *Scientific Simulations with Special-Purpose Computers: The GRAPE Systems*, John Wiley & Sons, New York, 1998.
3. D.J. Auerbach, W. Paul, and A.F. Bakkers, "A Special-Purpose Computer for Molecular Dynamics: Motivation, Design, and Application," *J. Physical Chemistry*, Vol. 91, No. 19, 10 Sept. 1987, pp. 4881–4890.
4. K. Boehncke et al., "Molecular Dynamics Simulation on a Systolic Ring of Transputers," *Transputer Research and Applications 3*, A.S. Wagner, ed., IOS Press, Amsterdam, 1990.
5. M.R. Pincus and H.A. Scheraga, "An Approximate Treatment of Long-Range Interactions in Proteins," *J. Physical Chemistry*, Vol. 81, No. 16, 11 Aug. 1977, pp. 1579–1583.
6. A.W. Appel, "An Efficient Program for Many-Body Simulation," *SIAM J. Scientific and Statistical Computing*, Vol. 6, No. 6, Jan. 1985, pp. 85–103.
7. J.E. Barnes and P. Hut, "A Hierarchical O(*N*log(*N*)) Force-Calculation Algorithm," *Nature*, Vol. 324, No. 6096, 4 Dec. 1986, pp. 446–449.