

Software

Micah Altman
Harvard-MIT Data Center, Harvard University

- I. Introduction
 - II. Algorithms, Computability, and Computational Tractability
 - III. Implementation: Bugs, Verification, and Accuracy
 - IV. Software Development
 - V. What software to write and what software to use?
-

Glossary

Algorithm: a precise set of instructions, in an abstract programming language, describing how to solve a problem

Compilation: The translation of instructions in a programming language into a machine language.

Machine Language: The set of instructions directly executable by a particular type of computer hardware.

Program: a sequence of instructions, written in some programming language, that a computer can execute.

Programming Language: An artificial language used to write computer programs that can be translated into a machine language.

Pseudocode: A notation resembling a programming language, but intended for pedagogy, not translation into machine language.

Software Engineering: The discipline engaged in systematic study and practice of designing and implementing software.

Software rot: The tendency of software to fail as it becomes older. Usually this is because of changes in the operating environment or limitations in the design assumptions, and not because of changes to the software.

Software, generally defined, is a set of instructions designed to be executed by a computer. Practically every modern method of quantitative and statistical analysis relies upon software for its execution. Despite its ubiquity, software is usually regarded (when considered at all) as necessary, but preferably invisible, infrastructure for such research. In fact, an understanding of software is essential to correct and efficient application of many quantitative and statistical methods.

I. Introduction

A. How is Software Used?

The use of software in social science research is extensive and wide-ranging. Software is used at every stage of the research process – from collecting, organizing, and analyzing information, to writing and disseminating results. The mathematically demanding nature of modern statistical analysis makes the use of relatively sophisticated statistical software a prerequisite for almost all of quantitative research. Moreover, the combination of increasingly powerful computers, ubiquitous computer networks, and the widespread availability of the software necessary to take advantage of both, have made practical on a hitherto unprecedented scale the application of many complex methods such as maximum likelihood estimation, agent-based modeling, analytic cartography, experimental economics.

B. A Brief History

The idea of the algorithm, which is basic to all computer software, goes back as far as 825 CE, to the Persian Mathematician Abu `Abd Allah Muhammad ibn Musa Al-Khwarizimi. The invention of the computer program is much more recent: Ada Lovelace is usually attributed with creating the first theoretical computer programs in 1843, to be used with Charles Babbage's analytical engine (which was never physically constructed).

The modern form of 'software', a set of instructions that is separated from the physical computer itself, originated in 1945 when John Von Neumann first proposed the "stored program". (John von Neumann. 1945. First Draft of a Report on the EDVAC. Reprinted with corrections in (1993) *Annals of the History of Computing* 15: 25-75.) This stored program (now known as a "computer program"), would comprise a set of instructions for a general purpose computer that would be stored in the computer's memory, along with the data, rather than being physically 'wired' into the computer hardware itself. (The *Eniac*, the first general-purpose, programmable, electronic digital computer, had to be rewired in order to run different programs, subsequent digital computers have followed Von Neumann's design.)

In the earliest period of digital computing, from the construction of the Eniac in 1945 through the mid-1950's, software was developed either by the end-user or by the manufacturer. Typically, the vendor delivered, at most, low-level utility programs. And the end-user would have to write whatever software that they needed, or copy it from another user.

A market for software contracting first developed in the mid-1950's, but application software remained entirely custom-written: The vendors of the mainframes of the day would supply freely the operating systems software needed to run the system, since such software was coupled so closely to that particular type of hardware that no one else had the experience or the time to develop it. All other software was written by the end-user, or by contractors, who would write custom software tailored to the needs of that user and application.

It was only in the 1960's that generalized software packages products first emerged, and not until the late 1970's that software became a 'shrink-wrapped', standardized, standalone, mass-market commodity. Today much of the software used in social science research is of the shrink-wrap variety: A standalone package, capable of performing a wide variety of functions, and written in a programming language that permits portability across different types of computer hardware. Still, while software has become much more standardized, social scientists sometimes find it necessary in the course of their research to write their own programs. In addition, despite the increasing standardization of software, its intrinsic complexity is such that even standardized, widely-used commercial software may on occasion yield wildly incorrect or inaccurate results.

II. Algorithms, Computability, and Computational Tractability

A. What is an Algorithm?

The idea of the *algorithm* is fundamental to software. An algorithm is a sequential set of steps that can be used to solve a well-defined problem. More strictly, an algorithm

is a finite, deterministic, set of instructions, written in an abstract syntax, that, when executed, completes in a finite amount of time, and solves a specified problem. An algorithm is said to *solve* a problem (or to be *effective*) if and only if it can be applied to *every* instance of that problem and is guaranteed to produce an exact solution for each instance.

Consider the problem of computing the standard deviation of a population,

$$\sigma = \sqrt{\frac{\sum (x - \bar{x})^2}{n}},$$

and the following *pseudocode* describing an algorithm that computes

it:

```

function standard_deviation (X: vector ) {
  variables x_sum, x_mean, x_std, : real ; n , count: integer;
  x_sum = 0;
  n = length(X);
  for count = 1 to n {
    x_sum = x_sum + X[count];
  }
  x_mean = x_sum/n;
  x_dev = 0;
  for count = 1 to n {
    x_dev = x_dev + (x_mean - X[count])^2;
  }
  x_dev = sqrt(x_dev/n);
  return (x_dev);
}

```

Example 1: Pseudocode for computing the Standard Deviation

A cursory examination will show that this algorithm is effective for any vector of real numbers, and will complete in an amount of time roughly proportional to the length of the vector.

This is not, of course, the only algorithm that solves the problem. The mathematical expression could be expanded and rearranged to yield

$$\sigma = \sqrt{\frac{n \sum x^2 - (\sum x)^2}{n^2}},$$

which computes the standard deviation in a single pass, without

first computing the mean. Directly translating this expression into pseudocode yields a different algorithm.

```

function standard_deviation_2 (X: vector ) {
  variables x_dev, x_sum_sq, x_sum : real ; n, count: integer;
  x_dev = 0; x_sum=0; x_sum_sq=0;
  n = length(X);
  for count = 1 to n {
    x_sum = x_sum + X[count];
    x_sum_sq = x_sum_sq + X[count]^2;
  }
  x_dev=sqrt((n*x_sum_sq - x_sum^2)/n^2);
  return (x_dev);
}

```

Example 2: A Single-Pass Algorithm for the Standard Deviation

Algorithmically, the two methods of computing the standard deviation are similar in structure, and the execution time for either is a linear function of the length of x . As we shall see in the next section, however, straightforward implementation of each of these algorithms may differ greatly in real accuracy and speed.

As stated above, the term ‘algorithm’ when not otherwise qualified, denotes a deterministic, finite set of steps, guaranteed to produce results with well-defined properties. Other categories of algorithms exist: approximation algorithms, randomized algorithms, and heuristic algorithms. These qualified classes of algorithms, especially heuristics, are used most frequently to approach problems for which no tractable deterministic, finite, effective, algorithm is known.

Approximation algorithms produce results that are guaranteed to be within some formally defined distance (usually given as a relative measure) of the optimal solution to a problem. For example, approximation algorithms are sometimes used for the ‘traveling salesperson’ problem (TSP), which is stated as follows: Given a list of cities, and the cost of travel between each, find the cheapest route that visits each city once and returns to the starting point. No algorithm solves the TSP efficiently, and it can be proved that no approximations exist for the general TSP problem. If the cost of travel between cities satisfies the triangle inequality, approximation algorithms (such as the Minimum Spanning Tree algorithm) exist that are guaranteed to yield solutions that are no more than twice the optimal cost. For other problems, approximation algorithms may exist that yield solutions arbitrarily close to the optimal solution. For example, when one uses a converging infinite series, such as a Taylor series, to approximate a function, one can

reduce the approximation error of the algorithm as much as desired by adding more terms.

Randomized algorithms use non-deterministic steps and have a known probability of yielding a correct answers. Randomized algorithms that can sometimes return incorrect results are called ‘Monte Carlo’ algorithms, while ‘Las Vegas’ algorithms may return indication that a solution was not found, but never return incorrect results. In contrast, *Heuristic algorithms*, often known simply as *heuristics*, specify sets of steps, but yield solutions that do not have well-known properties. Heuristic algorithms, in other words, yield solutions that are not known to be correct (or even approximately correct), but are thought to be often useful in practice.

Turing Machines

The *Turing machine* is an abstract representation of a computer introduced by Turing in 1936 to give a precise definition to the concept of the algorithm. (Turing, A.M. 1936. *On Computable Numbers, with an Application to the Entscheidungsproblem*. Proceedings of the London Mathematical Society, Series 2, 42 (1936-37) 230-265.) It is still widely used in computer science, primarily in proofs of computability and computational tractability. Turing imagined a mechanical device that moved along an infinite length of recording tape, reading and modifying symbols on that tape in accordance with a fixed internal table of actions. As a Turing machine moves along a tape it use its table, in combination with the current input symbol, and the contents of its internal *state register*, to determine the next action. The table indicates to the machine whether to modify the current symbol and state, and whether to move forward or backward along the tape.

Mathematically, a Turing machine is a tuple: $M = (K, \Sigma, \delta, s)$, where K is a finite set of states, $s \in K$ is the initial state, Σ is a finite alphabet of symbols, and δ is a transition function that represents the ‘program’ for the machine:

$$\delta : K \times \Sigma \rightarrow (K \cup \{halt, accept, reject\}) \times \Sigma \times \{left, right, stay\}$$

A string of symbols σ from the alphabet Σ represents the iNPut to the Turing machine. To 'execute' the machine, one applies δ to the first symbol in σ : $\delta_o = \delta(s, S_0)$ and uses the output to update σ , and to provide the iNPut for the next iteration of δ .

The Church-Turing thesis, in its most common form, states that every physically possible form of computation can be carried out by a Turing machine. This thesis is now generally assumed to be true, and has some important and useful implications: First, all computer language that are *Turing complete*, which includes all common programming language, are equivalent in what they can compute – any computation possible in (e.g.) FORTRAN, is possible (if not necessarily equally convenient) in any other language. If one can construct a proof of the (non)computability of a particular problem, the effectiveness of an algorithm, that proof applies to all other physically possible forms of computation (including quantum methods).

C. Computability and Computational Complexity

A problem is said to be *computable* (or decidable) if and only if there exists an algorithm that solves the problem. Turing, in his 1936 paper, first proved the *halting problem* to be undecidable. Informally, the halting problem can be stated as:

Given a description of an arbitrary algorithm and its input, whether the algorithm halts (yielding an answer) or runs infinitely.

Turing demonstrated that a direct consequence of the halting problem being undecidable is that there cannot be an algorithm that, given any statement about the natural numbers, determines that statement's truth. Subsequently, many other undecidable problems have been described – and the typical method of proof has been to show that a new problem reduces to the halting problem. Remarkably, two decades later, Rice showed that given any non-trivial property of a computer program (or mathematically, a partial function), the problem of determining whether that property applies to an arbitrary computer program is generally undecidable. (Rice, H. G., 1953, "Classes of Recursively Enumerable Sets and Their Decision Problems," *Transactions of the American Mathematical Society*: 74, 358-366.)

It is important to note that Turing's and Rice's proofs apply to the set of algorithms as a whole – not to all individual instances. It is certainly possible, for example to prove the correctness of some computer programs, although it is impossible for an algorithm to be able to determine the correctness of any program given to it. For individual instances of algorithms, the central roles of algorithmic analysis are to determine the correctness and efficiency of individual algorithms, and to characterize the difficulty of different classes of problems.

Computer scientists use *computational complexity classes* to characterize the difficulty of computable problems. A complexity class comprises a model/mode of computation (e.g., the deterministic Turing machine described above), a resource we wish to bound (e.g., execution time or storage space), and a bounding function. A problem is said to be a member of the class, if some algorithm exists that, using the specified mode of computation, can solve any instance of that problem using amounts of resources limited by the given bound. Bounding functions for execution time are conventionally denoted using “Big O notation”, $O(f(n))$, where n is the size of the problem. Additive and multiplicative constants are omitted, as these vary with the computing model used. For example, $O(2^n)$ denotes that the time it takes to execute an algorithm grows exponentially as input grows (for worst-case instances).

There are an infinite number of possible complexity classes. Two classes, **P** and **NP**, are of particular interest, because they are widely used as measures of computational tractability. Both **P** and **NP** are measures of time complexity, which is proportional (by construction) to the number of the instructions that algorithm must execute to reach a solution. The bounding function is expressed in terms of the *size* of the problem-instance, which itself is defined as the number of parameters or items (of fixed size) in the instance. (Technically, **NP** applies to decision problems that yield true or false as an answer. However, other types of problems can easily be converted to decision problems to determine the complexity class.)

The class **P** is the set of problems for which algorithms exist that can solve *any* instance in polynomial time. Formally, the class **NP** is defined as the set of problems

solvable in polynomial time by a *non-deterministic* Turing machine, which would automatically choose the right answer from among a finite set of possible logic branches. (This is a useful mathematical construct, but not physically possible.) This class of problems is widely thought to require exponential time to compute by any real computer, for at least one instance of the problem: $O(c^n)$, $c > 1$. **NP-complete** problems are thought to be the most difficult problems in **NP**. A problem is **NP-complete** if it is in **NP** and if every other problem in **NP** is reducible to it.

A problem is said to be *computationally tractable* if it is in **P**. A problem is said to be *computationally intractable* (also “computationally complex” or “computationally hard”) if a problem is at least as hard as a problem in **NP**. No polynomial-time algorithm is known to exist, using conventional computers, for any problem in **NP**.

The most common way to show that a particular type of problem is **NP-hard** is to show that another problem already known to be **NP-hard** can be reduced to it. There are many types of reduction techniques, and one particularly straightforward type is called the *Karp Reduction*, or the polynomial-time many-one reduction. To use this technique, one finds a known **NP-hard**, and a polynomial-time algorithm that converts any instance of that problem to an instance of the new problem. Since any algorithm that solves the new problem would also solve the time-complexity of the intractable problem, the new problem must be at least as hard as the intractable problem. To show that the problem is **NP-complete**, one proves, that a known **NP-complete** problem can be reduced to the new problem *and vice-versa*..

The use of membership in **P** and **NP** to characterize a problem as tractable or intractable (respectively) has two advantages: First, it is independent of any particular computer hardware design. Intractable problems cannot be made tractable through improvements in conventional hardware technology. Second, it is independent of any particular algorithm, since it is the problem itself, not a specific algorithm, that drives the requirement of exponential time. Intractable problems cannot be made tractable through advances in software or algorithmic design.

There are, however, some limitations to this characterization of tractability. First, the distinction between tractable and intractable problems is most important for instances of large size - where the exponential factors in the time requirements of these problems become dominant. If problem “A” is solvable in $O(1.1^n)$ time, problem “B” is solvable in $O(n^{2000})$ steps, “B” is formally more tractable than “A”, but is more difficult to solve in practice. Second, **NP**-completeness is a worst-case measure of complexity: Some problems in **NP** may have instances that can be solved in polynomial time, and it may even be the case that the average instance is solvable in polynomial time. Third, **NP** applies strictly to deterministic exact algorithms. An ‘intractable’ problem, may still be ‘solved’ by a randomized algorithms that gives a solution with high probability, or an approximation algorithm that is close to the desired solution. Third, a small number of problems in **NP** (but not **NP**-complete) are thought to be solvable efficiently with quantum computers, should such computers ever be constructed on a large enough scale. It is now believed, however, that no physically possible quantum computer can compute **NP**-complete problems efficiently. (Bennett, C.H., Bernstein, E., Brassard, G., Vazirani, U., “Strengths and Weaknesses of Quantum Computing”, *SIAM Journal on Computing* **26**, 1510-23)

Despite these theoretical limitations, relatively few **NP**-hard problems have been found to be easier than expected in practice, few have yielded to approximations, randomization, or quantum algorithm techniques, and few have been found easy in the average case. **NP**-completeness remains a powerful and widely used gauge of computational tractability.

D. An Application of Complexity Theory: How Hard is it to Manipulate an Election?

One key problem that confronts voting systems is the potential for voters to manipulate the results through strategic voting. A voter is said to act strategically when she casts a vote that does not reflect her true ranking over the choices, but is calculated

instead to achieve a favorable outcome: For example, a voter in a presidential election might prefer the Libertarian candidate to the Republican and Democratic candidates, but casts a vote for the Republican candidate because she believes the Libertarian has a negligible chance of winning. A voting system is said to be non-manipulable if it is not possible for any voter to gain from strategic voting.

Strategic voting has been studied extensively in political science and economics. A powerful negative result, discovered by Gibbard and Satterthwaite (independently) in the early 1970's, is that any non-dictatorial voting scheme is manipulable (for elections with at least three candidates). Like Arrow's theorem, on which Satterthwaite's proof drew, this impossibility result engendered some pessimism regarding the design of electoral systems.

Fifteen years later, Bartholdi, Tovey, and Trick used computational complexity theory to show that under some voting systems, effective strategic voting is **NP**-hard. (Bartholdi, J. J. III, C. A. Tovey, and M.A. Trick. 1989. "The Computational Difficulty of manipulating an election" *Social Choice and Welfare* 6, 227--241, 1989.) Thus, these elections systems are, if not manipulation-proof, at least manipulation-resistant. In using complexity theory to inform social choice theory, they initiated the study of the computational properties of electoral systems.

Bartholdi, Tovey and Trick's proof is too long to present here, but another proof in the same vein is instructive. This proof, which comes from previous work by the author, shows that the problem of constructing 'optimal' election districts is **NP**-hard. The portion below shows that constructing a district plan having the minimum possible population deviation among them from discrete non-uniform census blocs, as the law requires, is **NP**-hard:

1. Note that each of the n census blocs, c_i , must be assigned to one and only one of k election districts, d_j , and that the population of the district is simply the sum of all census blocs that comprise it:

$$Population = P(d_i) = \sum_{j \in d_i} c_j$$

2. Define the measure of population deviation for a redistricting plan to be the difference in population between the most and least populous district.

$$\text{Population Deviation Score} = \mathbf{PD}(\mathbf{c}) = \max_{i \in k} \left(\sum_{j \in d_i} c_j \right) - \min_{i \in k} \left(\sum_{j \in d_i} c_j \right)$$

3. The optimal districting plan is thus a division (strictly a partition) of the n census blocs into k districts such that \mathbf{PD} is minimized.
4. Finally, we show that we convert any instance of **3-Partition**, a known **NP**-complete problem, into the redistricting problem above:

3-Partition

INPUT: Set A of $3m$ elements, a bound $B \in \mathbb{Z}^+$ and a size $s(a) \in \mathbb{Z}^+$ for each $a \in A$ such that $B/4 < s(a) < B/2$ and such that $\sum_{a \in A} s(a) = mB$.

Solution: Determine whether A can be partitioned into m disjoint subsets such that for $1 \leq i \leq m$, $\sum_{a \in A} s(a) = B$.

To convert an instance of 3-Partition into an instance ‘optimal redistricting’:

- a) For each element $a_i \in A$, create an artificial census bloc with a population equivalent in size, $c_i = s(a_i)$.
- b) Take the solution for the optimal redistricting problem, \mathbf{PD}^* , using the artificial census blocs created in (a). If $\mathbf{PD}^* = 0$, the answer to the corresponding 3-partition is “true”, otherwise, the answer is false.

Thus any algorithm which solves the optimal redistricting problem can also be used to solve 3-partition. Since the 3-partition problem is computational intractable optimal redistricting is computationally intractable, as well.

III. Implementation: Bugs, Verification, and Accuracy

Although algorithms are at the conceptual core of all software, computers execute not algorithms, but *programs* – implementations of an algorithm written in some real

programming language and executing within a particular computing environment. The same algorithm may be expressed using various computer languages, may use varying encoding schemes for variables and parameters, may rely on arithmetic operators with varying levels accuracy and precision in calculations, and may run on computers with varying performance characteristics. Three problems can grow in the gap that arises between the formal algorithm and its actual implementation: bugs, inaccuracies, and performance bottlenecks.

A. Bugs and Verification

Any computer program of reasonable size is sure to have some programming errors, or “bugs”, and there is always some possibility that these errors will affect research results. In practice, software used in research will be tested, but not proven correct. As Dahl, Dijkstra, and Hoare (1972) famously wrote, program testing can be used to show the presence of bugs, but never to show their absence. (Dahl, O.J., Dijkstra, E. W., and Hoare, C. A. R. (1972). *Structured Programming*. San Diego: Academic Press.) It has also been observed generally that even software which worked well when first written tends to encounter problems as changes occur to the computing environment within which it runs, such as the operating system, system libraries and computing hardware. (This phenomenon is known colloquially as *software rot*.)

Errors in mathematically-oriented programs are often particularly difficult to detect, since the software may be incorrect yet return plausible results rather than failing entirely. Well-replicated studies of experienced spreadsheet programmers performing standardized tasks have demonstrated thoroughly that undetected bugs are the rule, not the exception. While extensive testing will substantially reduce the rate of errors which remain in the final version of a piece of software, much caution is still warranted when creating one’s own software.

In limited circumstances it is possible to prove software correct, but it is exceedingly unlikely that any particular software package used by social scientists will have been subject to these formal methods of verification. Until recently, in fact, such formal methods were widely viewed as completely impractical by practitioners, and

despite increasing use in secure and safety critical environments, usage remains costly and restrictive.

B. Computer Arithmetic, Numerical Accuracy and Stability

Mathematicians, social scientists, and other humans, perform arithmetic symbolically – computers do not. The difference between symbolic and computer arithmetic can lead to inaccuracies, and to avoid these inaccuracies, we need to understand how computers do math. All computer hardware, and nearly all software, performs arithmetic by representing every number as a fixed-length sequence of 1's and 0's, or *bits*, b . Integers are often represented as a single sequence of bits, each representing a different power of two, with a single bit indicating the sign. Under this representation, arithmetic on integers operates according to the “normal” (symbolic) rules of arithmetic, as long as the integer operands and results not too large ($> 2^{b-1} - 1$), which can cause a (possibly undetected) *overflow* error. For example, usually $b=32$, so the number “2147483648” ($1 + 2^{32-1} - 1$) would overflow and may actually roll-over to -1

Real numbers are represented using *floating point arithmetic*. Floating point numbers are represented by two sequences of bits, with one sequence representing a mantissa (m), and the other representing an exponent (e): $\pm m * 10^e$. An additional bit indicates the sign. The specific details of floating point arithmetic operations vary somewhat across different computing platforms. (The algorithms for performing floating point arithmetic encompass some subtle technical details, which are beyond the scope of this article.)

All floating point representations are subject to overflow, *underflow* – when the true number is smaller than the smallest value capable of being represented, and rounding errors. Rounding and other numerical problems can lead to inaccurate results, even when every step of an algorithm is correctly followed. (The accuracy of the solution is, roughly, the distance between the results that are actually produced, and the correct answers, when computed using infinite precision.) One source of rounding error arises directly from storing data in this representation -- some numbers cannot be exactly represented using this scheme. An example is the number 0.1, which has an infinitely

repeating binary representation using this technique. The infinitely repeating floating point transformation of “0.1” must be rounded to m bits, resulting in a slight loss of accuracy when performing subsequent calculations. A second source of rounding error occurs when a number is added to (or subtracted from) a very much smaller number. This type of rounding error can occur even when both operands are exactly represented. In the extreme case the result simply rounds to the very large number.

Underflow, overflow and rounding have many implications for accurate computing. One of the implications of the pattern of rounding errors just described is that summations are more accurate when performed on a list of elements that is sorted in order of increasing magnitude. So, the algorithm in Example 1 would produce a more accurate result if we modified it as below:

```
function standard_deviation_3 (X: vector ) {  
  variables X_sort: vector;  
  
  X_sort= sort_least_to_greatest(X);  
  return(standard_deviation(X_sort));  
}
```

Example 3: A more accurate standard deviation

Standard proofs of the correctness of particular algorithms usually ignore the underlying arithmetic implementation, and the effects of rounding.

The limits of computer arithmetic, and the variations in it across different platforms, have three implications for replicability and accuracy. First, a computer program can produce different answers when run on different computers, run on the same computer using a different operating system, or when recompiled with different options. Second, numeric errors can accumulate within an algorithm, or can (albeit rarely) cancel each other. Third, inaccuracies in floating point arithmetic can interfere with the formal mathematical properties of elementary and non-elementary functions: For example, the associative law of arithmetic does not hold universally for computer arithmetic: Where ‘ \oplus ’ is the floating-point addition, $(a \oplus b) \oplus c \neq a \oplus (b \oplus c)$. Practically, floating point inaccuracies can cause a range of problems – from small inaccuracies in the results, to results that are returned without error but are completely inaccurate, or even failure of an

otherwise correct algorithm to halt. Careful analysis of the accuracy of each numerical implementation, in its entirety, is necessary in order to ascertain the level of accuracy that can be associated with a particular solution.

Surprisingly, accuracy alone is not enough to ensure that implementations of algorithms produce usable results. Because of the rounding of data when stored initially, and because of the possibility of measurement error in much of social science, a reliable implementation must be *numerically stable* as well as accurate. An algorithm for computing a function is said to be “numerically stable” if small errors in input cause only small errors in output, i.e.: $\hat{y} + \Delta y = f(x + \Delta x)$, where x is the true input, \hat{y} is the true output, and $\hat{y} + \Delta y$ is the computed value. Δx is error that enters into computations through, for example, converting a decimal number into a binary number with a finite degree of precision. Less formally, a stable algorithm gives “almost the right answer to almost the right problem.”

C. Performance Tuning and Bottlenecks

Algorithms determine performance at the grand scale. For sufficiently large values of n , an $O(n)$ sort algorithm will finish before an $O(n^2)$ algorithm. Nevertheless, implementation matters -- it is not uncommon that a well-tuned implementation of a particular algorithm will run an order of magnitude faster and use an order of magnitude less resources than a naïve implementation of the same algorithm. This arises most commonly because of performance bottlenecks. In all computing architectures, executing a program involves accessing implicitly a variety of heterogeneous resources such as: floating-point units for arithmetic calculations, memory chips and storage devices for access to data, and networks and busses for communication. These resources have different performances characteristics, and it and a bottleneck may occur when the calculations of the program are interrupted to wait for some slower (or temporarily unavailable) computer resource. It is rare that formal algorithmic analysis delves down to this level of detail.

Programmers and compilers can use profiling tools to analyze the empirical behavior of a particular computer program, by monitoring its performance and the resources it uses as it runs. By rearranging the order of operations, changing the pattern of access to data, and substituting equivalent (but more efficient) sets of computing instructions, the program can be made to run faster. Low-level performance tuning, however, is often tied closely to a particular hardware configuration, programming language, and operating environment, and is at odds with portability, clarity, and maintenance. Moreover, it is usually counter-productive to tune a program without an empirically generated profile of its performance. Thus, performance tuning is best done after the software is designed, written, and tested.

E. Application: Benchmarking a Statistical Package

Consider, the two algorithms for computing the standard deviation discussed above. If the vector X is very large, it may be expensive to read values from it. In this case, despite both algorithms having roughly equivalent time-complexity, the program implementing the single-pass algorithm in **Example 2** could be much faster than a program implementing the algorithm in **Example 1**, which requires that every element of X be read twice. However, the algorithm in **Example 2** is, in practice, much more susceptible to rounding errors, when $n\sum x^2$ and $(\sum x)^2$ are both large. (In this case, the tradeoff between performance and accuracy is avoidable – more accurate one-pass algorithms exist.)

In practice, programs are often so complex that numerical *benchmarks*, rather than formal analysis, are used to gauge their accuracy. Numerous benchmarks are now available for testing simple statistical functions and models, such as univariate descriptive statistics, cumulative distribution functions, linear regression, analysis of variance, and nonlinear regression. (See Altman, et. al 2004, for a review). One particularly popular and useful set, the Statistical Reference Datasets (StRD) is maintained by the National Institute of Standards and Technology (NIST).

Each StRD problem is composed either of data taken from published research or specially generated to stress computational capabilities. For each problem, the data are

accompanied by values certified by NIST to be correct. These values are obtained analytically where possible, or by using supercomputers to compute approximate results using two independent algorithms and exceptionally high-precision floating point arithmetic.

As an example of a benchmark consider a simple univariate descriptive statistics problem: compute the mean, standard deviation, and one-observation lag autocorrelation for the first n digits of π . The StRD provides both the input data and the result values. These have been calculated on a super-computer using very high precision arithmetic, and the results rounded to fifteen significant digits. The resulting values, are said to be “certified” by the benchmark.

To gauge the reliability of a particular statistical package using the StRD, one loads the data into the program, runs the analysis, and compares the results generated by the software to the certified values provided by NIST. While no benchmarking method can prove that a piece of software is accurate, performing well on the StRD benchmark provides some evidence that a software package is accurate for that domain of problems.

The following transcript shows the results of running the NIST ‘Pi-digits’ benchmark in a popular statistics package:

```

> options(digits=22) # set number of digits to display
> x<-read.csv(file="PiDigits.txt"
  header=TRUE) # read test data
> x[1:10,1] # check data, digits of pi
[1] 3 1 4 1 5 9 2 6 5 3
> xm<-mean(x[[1]]) # calculate mean of data
> xm
[1] 4.534799999999999720046
> xsd<-sd(x[[1]]) # calculate standard deviation
> xsd
[1] 2.867543699108215271565
> xacf<-acf(x[[1]], lag.max=1, type=c("correlation"),plot=FALSE)
> xac<-xacf[[1]][2] # calculate autocorrelation
> xac
[1] -0.003683261271785385603666
> -log(abs((xm-4.5348)/4.5348)) # calculate LRE for mean,sd,ac
[1] Inf
> -log(abs( (xsd-2.86733906028871)/2.86733906028871))
[1] 35.0175962904394
> -log(abs( (xac+0.00355099287237972)/-0.00355099287237972))
[1] 34.8496905126905

```

Example 4: Testing A Statistics Package for Accuracy, by Using the Digits of Pi

In this example, we calculate the number of correct digits in the results produced by the statistical software using the *log relative error*, or *LRE*. More formally the LRE is:

$$-\log_{10}\left(\left|\frac{\text{result} - \text{certified}}{\text{certified}}\right|\right), \text{certified} \neq 0.$$

The statistics package above agreed with the certified benchmark results to at least 34 digits. Since the accuracy of the benchmark is only certified to 15 digits we can infer that the statistical package was accurate to at least 15 digits for these calculations.

IV. Software Development

Developing software remains a complex and difficult activity. Many practitioners suggest that this complexity is unavoidable. The complexity of the problem domains to which software is applied, the inherent malleability of software, and the problems that characterize the behavior of discrete systems with large state-spaces, all contribute to the overall difficulty of developing software. Regardless of its source, the difficulty of writing software has a number of important consequences, and much of the work in the field of software engineering is devoted to managing the complexity of software and the risks that result from such complexity.

A. The Implications of Complexity for Programming

One well-known consequence of software's complexity is that there is wide variation in the quantity and quality of work produced by individual programmers: Early findings by Sackman *et al.* found differences of more than 20 to 1 in the time required by experienced programmers to solve the same problem. (Sackman, H., Erikson, W.I., Grant, E.E., 1968. "Exploratory Experimental Studies Comparing Online and Offline Programming Performances," *Communications of the ACM* **11**: 3-11.) This result has been widely replicated, with findings of large differences in productivity and defect rate across programmers and programming teams.

A second consequence of the software's complexity is that as the desired functionality of a program increases, the program itself eventually becomes too complex to understand and manage all at once. As a consequence, most programming techniques seek to manage the complexity of programs by decomposing them into smaller, simpler, components. All decompositions attempt to form *abstractions* to represent each of the components, so that some details about the implementation can be *encapsulated*, or hidden from, the other components. And for each component, a set of *interfaces* are created that define the way each component is used, so that components can be integrated together to form the program. If the decomposition is successful, the implementation of each component can be changed over time and later be adapted to different computing needs. And, as long as the interface behavior is preserved, the program will continue to function correctly as a whole.

The first-generation of programming languages, such as FORTRAN I, were designed to support decomposition of programs into mathematical expressions. It was quickly realized that this method of decomposition was too limiting. Modern programming languages and methods typically support one of five strategies for decomposing problems: Procedure-oriented methods aim to decompose programs directly into sets of algorithms. Object-oriented methods aim to decompose programs into classes, objects, and behaviors, the last of which encapsulates the specific problem-solving algorithms used. Logic-oriented, rule-oriented, and constraint-oriented methods aim to decompose programs into set of goals, if-then rules, and constraints (respectively). Each

of these latter three methods then uses generalized algorithms to solve for or optimize against the specific sets of goals, rules, or constraints.

For example, consider the **sort_least_to_greatest()** routine in the **Example 3**. This function is an example of a procedure-oriented decomposition of a problem. The decomposition of our program separates the sorting algorithm from the algorithm used to compute the standard deviation. The implementation of the sort procedure is unspecified, but could be any one of a wide variety of sort algorithm, depending on the programmer's desire to save space, time, or effort. Regardless, whatever algorithm is used, the program as a whole remains correct as long as the same procedural interface is provided with each algorithm.

There is no method of decomposition that is universally better for all applications. For example, logic-oriented programming techniques and languages are considered to be particularly well-suited for some applications in 'artificial intelligence.' For general applications, however, the procedure and object-oriented methods are most commonly used, and modern best-practices and programming languages emphasize the object-oriented approach. Moreover, research is active in the area of extending and augmenting the object-oriented paradigm: Techniques such as component-based programming and design patterns provide methods for grouping objects into larger abstractions, and new paradigms such as aspect-oriented programming aim to augment object-oriented approaches with alternative, concurrent decomposition strategies.

B. Software Development Life-Cycle

Special caution is warranted when a software program is large enough to involve more than one author. In 1975, Brooks, discussed the widespread difficulties of software development and formulated his well-known, and well-studied, 'law': "Adding manpower to a late software project makes it later." – resulting from the fact that adding additional programmers often increases the communication and coordination costs involved in a project faster than it decreases the remaining work. (Brooks, F. P., jr. 1975, *The Mythical Man Month*, Addison-Wesley: Reading, Mass.)

Surveys of practice in industry continue to demonstrate the difficulties of software development: A significant minority of projects are never completed, and the vast majority of the remainder finish significantly over budget, long past the original deadlines, and often with reduced or impaired functionality. Although the exact percentages are debated, it is widely recognized that the majority of software projects of moderate size fall short in some serious way.

Projects falter for many reasons, but the two causes most commonly diagnosed are poor schedule estimation, which is nearly universal, and problems deriving from excessive, changing, unclear, or incomplete requirements. Models for development have been proposed to address these common problems. The first model for developing software, now known as the “waterfall” model, was articulated (although not advocated in its pure form) by Royce in 1970, and quickly gained dominance. (Royce, W. W., 1970. "Managing the Development of Large Software Systems," Proceedings, IEEE Wescon.) It advocated that software development proceed in five phases: *Requirements specification*, in which the functionality of the software is described in detail. *Design*, in which the overall structure of the software is designed, and broken into sub-components with well-defined interfaces. *Implementation*, in which the code for each component is written and tested individually. *Integration*, in which individual components are integrated into a complete system which is then tested together. And, *operation and maintenance*, in which the software is delivered to the customer, modified to meet changing requirements, and repaired as bugs are discovered.

Although the waterfall model can still be found in modern use, it is now regarded as somewhat naïve. A decade later, the model was widely criticized, as it became clear with experience that it did not adequately address either design risks or requirements risks: Risks that requirements were not properly anticipated, were misunderstood, or needed to change over the course of the project, and risks that the design of the project was flawed, incomplete, or misunderstood in implementation. Recognition of these risks has led many to abandon the waterfall model in favor of incremental and iterative models of software development, in which multiple versions of a project are developed and delivered to users over the lifecycle of the project. Rapid development, testing, and

delivery of smaller (and possibly incomplete) increments, prototypes, and/or versions of a software product allows for the ongoing incorporation of feedback from users into the requirements and design phases of future increments. While there are continuing and vigorous debates about which processes are best, it is widely recognized that some form of incremental approach is necessary to successfully complete a project.

C. Software Distribution

Once software is created, it is most often distributed in one of two forms. Software can be distributed as source code, which provides the high-level, human-readable set of instructions comprising the software, or the source code can be *compiled* (converted) into low-level instructions, commonly known as “object code”, and which can be distributed alone and executed directly by the computer.

Object code is difficult, although not impossible, for humans to inspect or modify, and inherently obscures the design and algorithms used in the software. Distribution of software as object code is the norm for commercial software products, because it helps to protect the intellectual property embodied in the software. In addition, this intellectual property is protected by copyright law, and often, in addition, by some combination of patent, trademark, and trade-secret law. Under current law, large civil and/or criminal penalties can be levied against those who make illicit use of source code, or even simply reverse engineer the object code.

On the other hand, source code distribution, when done properly, is conducive to software reuse and extension. Source code distribution is the method of choice in academic research and non-commercial projects, because distribution of the source code enables others to learn from and potentially improve the software itself. Moreover, an innovative family of licenses, known as “Open Source” licenses, have been designed to provide an incentive for others to learn from and improve software by guaranteeing that the software may be freely used for any purpose (including commercial purposes) and that any future modifications or extensions of the software, by anyone, will also be freely available in source form. The open source license parallels the academic norm of

openness, which requires that publicly recognized (published) research be replicable, in order that others may verify and extend them.

v. What software to write and what software to use?

In most research, especially that involving standard methodologies, it is usually more appropriate to use a standard software package than to write one's own. Since all software contains bugs and can produce inaccurate results in some circumstances, one should choose a package that is as open to inspection as possible. The software should document the algorithms used, especially those relevant to data processing and analysis. The documentation should include also information regarding the expected range of inputs for each algorithm and the accuracy of the results within this range, and should explain the warnings or errors that the software will produce when it encounters problems. Software packages that provide source have an advantage in this area, as users can inspect the code directly. Still, the availability of source code is not a substitute for thorough documentation.

Choose code that is, in addition, well-tested, particularly for the tasks to which you intend to put it. The developers of the software should have a clearly explained methodology for testing their software, and provide a complete record of all changes to the software, including previously reported bugs and subsequent fixes. Mathematical programs should document the accuracy of any functions and routines available to the users, and provide test results using standard benchmarks.

Sometimes, however, one may not be able to find well-documented and thoroughly-tested software that uses algorithms appropriate for one's problem. When this occurs, one must weigh carefully the potential for inaccuracies or inefficiencies to arise from applying an algorithm to a problem for which it is not well suited appropriate against the considerable effort required to develop software, and the prevalence of bugs and inaccuracies in freshly written software.

See Also

Computer Simulation
Computer-based Mapping
Computer-Based Testing
Computerized Adaptive Testing
Correspondence Analysis
Data Distribution and Cataloging
Factor Analysis
Fixed Effect Models
Innovative Computerized Test Items
Linear Models, Problems
Mathematical Demography
Maximum Likelihood Estimation
Multidimensional Scaling Models
Network Analysis
Nonlinear models
Ordinary Least Squares
Path Analysis
Spatial Econometrics
Spatial Pattern Analysis
Structural Equation Models
Time series
Web-Based Surveys

Bibliography

Altman, M, J. Gill, and M. McDonald, 2003, *Numerical Issues in Statistical Computing for the Social Scientist*, John Wiley and Sons: New York.

Bentley, J. 1982, *Writing Efficient Programs* Prentice Hall Computer Books, New York.

Booch, G. ,1994. *Object-Oriented Analysis and Design With Applications*, Addison-Wesley: Reading, MA

Cormen, T. H., C. E. Leiserson, R. L. Rivest, 1990; *Introduction to Algorithms*, MIT Press: Cambridge.

Gamma, E., R. Helm, R. Johnson, J. Vlissides, 1995. *Design Patterns*, Addison-Wesley: Reading, Mass.

Hennesy, J.L, Patterson, D.A., Goldberg, D. 2002. *Computer Architecture: A Quantitative Approach* (3rd Edition), Morgan Kaufman, New York.

Higham, N. J. 2002, *The Accuracy and Stability of Numerical Algorithms* (2nd edition), SIAM Press: Philadelphia.

Knuth, D. E., 1998; *The Art of Computer Programming* (2nd Edition) V. 1-3, Addison-Wesley: Reading, Mass.

McConnell, S. C., 1996; *Rapid Development: Taming Wild Software Schedules*, Microsoft Press: Redmond

Papadimitriou, C. H., 1994, *Computational Complexity*,
Addison-Wesley: Reading, Mass

Royce, W. 1998. *Software Project Management: A Unified Framework*
Addison-Wesley: Reading, Mass

Skienna, S. S., 1997, *The Algorithm Design Manual*,
Springer-Verlag: New York