A reprint from

American Scientist
the magazine of Sigma Xi, The Scientific Research Society

Calculemus!

Brian Hayes

THIS COLUMN MARKS an anniversary: It has been 25 years since I began writing these essays on the pleasures and possibilities of computation. My first columns appeared in *Scientific American*; later I wrote for *Computer Language* and then *The Sciences*; since 1993 the column has been happily at home here in *American Scientist*. (Some of my earlier essays are newly available online at bit-player.org/pubs.)

For my very first column, in October of 1983, I chose as an epigraph some words of Gottfried Wilhelm von Leibniz: "Let us calculate!" In Leibniz's Latin this exhortation was actually just one word: "*Calculemus!*" Leibniz was an optimist—he was the model of Voltaire's Dr. Pangloss—and he saw a bright future for what we would now call algorithmic thinking. Calculation would be the key to settling all human conflicts and disagreements, he believed. I can't quite match Leibniz's faith in attaining world peace through computation, but in my own way I'm an algorithmic optimist too. I see computing as an important tool for helping us understand the world we live in and enriching our experience of life.

When I wrote that first column, the idea of a personal computer was still a novelty, and there was some question what it might be good for. Now the computer is a fixture of daily life. We rely on it to read the news, to keep in touch with friends, to listen to music and watch movies, to pay bills, to play games, and occasionally to get a bit of work done. Oddly enough, though,

Celebrating 25 years of celebrating computation

one thing we seldom do with the computer is compute. Only a minority of computer users ever sit down to write a program as a step in solving a problem or answering a question. In this column I want to celebrate the rewards of programming and computing, and cheer on those who get their kicks out of this peculiar sport. I also have a few words to say about the evolution of tools for programming.

Inquisitive Computing

Let me be clear about what kind of programming I have in mind. I'm *not* talking about software development. In software development, the end product is the program itself. The developer builds a web browser, say, or a word processor—a program that others can then put to use. In *my* kind of programming, the product is not the program but the result of running the program. That result might be a number or a graph or an image; in general, it's an answer to a question. Let's call the whole process *inquisitive computing*.

The earliest computers were all used in an inquisitive mode. One of those innovative machines was the EDSAC, built at Cambridge University under the direction of Maurice V. Wilkes. On May 6, 1949, a punched paper tape was threaded into the EDSAC's input device, and a few seconds later a nearby teleprinter began typing out the numbers 0, 1, 4, 9, 16, 25, 36, continuing on to 9,801. Clearly, this was computing for answers, although I suppose the Cambridge dons assembled for the demonstration weren't really there to learn the squares of the integers from 0 through 99. A few days later another program generated a table of prime numbers up to 1,021.

If these accomplishments seem trifling, keep in mind that the first EDSAC programs had to take control of the machine at the level of bare metal.There were no operating systems or programming languages. Every step in an algorithm had to be specified in excruciating detail. Merely getting the teletype to print out a number took a dozen instructions.

Over the next decade, the EDSAC had a distinguished career in inquisitive computing. The statistician Ronald A. Fisher used the machine to solve a problem in genetics, quantifying the effect of selective advantage on gene frequency. The Danish astronomer Peter Naur came to Cambridge to calculate orbits of planetoids and comets. Naur was knocked out of his own orbit by this encounter with EDSAC; he left astronomy and became a distinguished theorist of computing. The biologist John Kendrew relied on the EDSAC to analyze x-ray diffraction patterns of the myoglobin molecule, thereby elucidating the three-dimensional structure of the protein.

In the 1970s, the hobbyists who built or bought the first microcomputers faced a predicament much like that of the EDSAC pioneers. The machines came with little or no software. If you wanted to do anything interesting with your new toy, your only option was to write a program. Thus another generation entertained themselves by printing out lists of squares and primes; the ambitious and persistent ones went on to plot the dizzy contours of the Mandelbrot set or to search for patterns in John Horton Conway's game of life.

Inquisitive computing has a less prominent role today, if only because

Brian Hayes is senior writer for American Scientist. *A collection of his essays,* Group Theory in the Bedroom, and Other Mathematical Diversions, *was published in April by Hill and Wang. Additional material related to the "Computing Science" column appears in Hayes's Weblog at http://bit-player.org. Address: 211 Dacian Avenue, Durham, NC 27701. Internet: brian@bit-player.org*

so many other applications of computers have upstaged it. These days, if I suggest that you answer a question by consulting a computer, you would think I meant to go ask Google. Nevertheless, programming for answers is still a living art. In large-scale scientific computing, clusters with hundreds or thousands of processors are put to work modeling the planet's climate and simulating collisions of protons or collisions of galaxies. Studies of proteins have gone far beyond Kendrew's crystallography; computers now try to predict from first principles how the long strand of a protein molecule will fold up into a three-dimensional tangle.

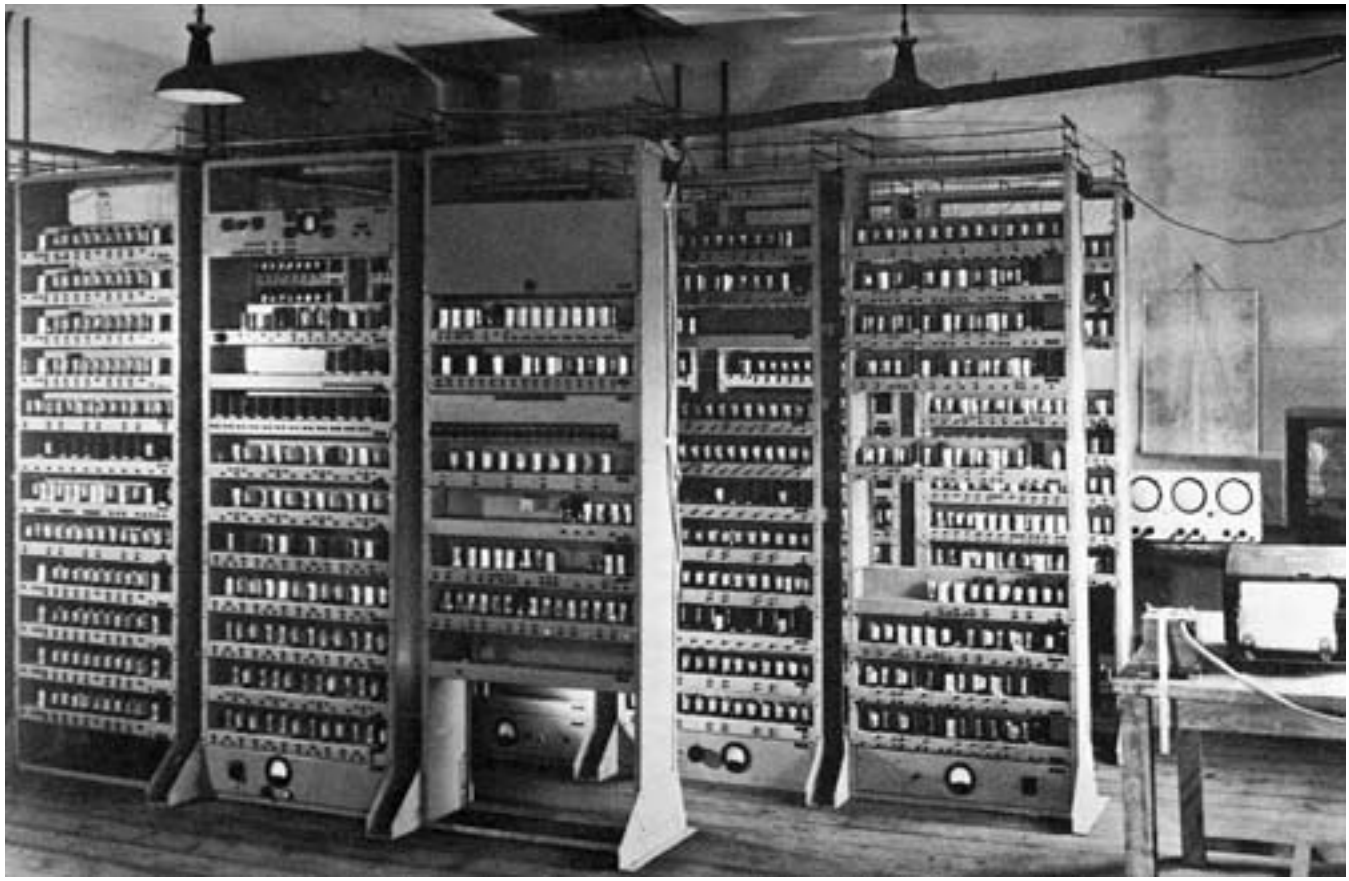
Climate models are a bit too ambitious for most of us. I want to focus on smaller and more casual programming challenges—computational problems or puzzles you might play with for a few minutes or a few days. Here are three examples that involve nothing more than simple arithmetic applied to integers.

Perfect Medians

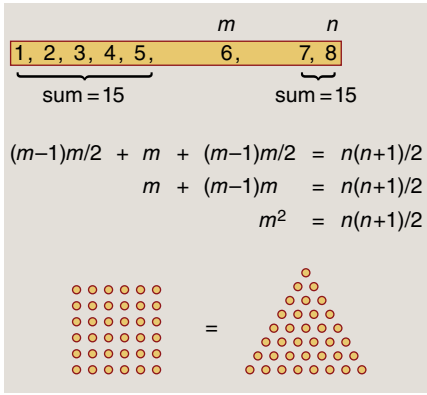
A puzzle attributed to the late David Gale observes that the sequence 1, 2, 3, 4, 5, 6, 7, 8 has a "perfect median," namely 6, because the sum of the terms preceding 6 is equal to the sum of the terms following it. Are there other subsequences of the counting numbers that have perfect medians? For what values of n does the sequence 1, 2, 3, ..., n have a perfect median?

You might be able to solve this problem without the aid of a computer, but I made no progress with pencil and paper. On the other hand, a program to search for perfect medians takes only a few lines of code. Instead of checking each sequence 1, 2, 3, ..., n to see if it contains a perfect median, it's easier to turn the problem inside out and check each integer m to see if it is the perfect median of some sequence. The first step is to add up all the numbers less than m; call the result T. (A bright 10-year-old might figure out a way to calculate T without actually doing all the additions.) Then loop through successive numbers starting with $m+1$, summing as you go. If this sum is ever equal to T, then m is a perfect median.

This approach immediately reveals that 35 is the perfect median of the sequence 1, 2, 3, ..., 49. In a few seconds you discover three more perfect medians: 204, then 1,189 and 6,930. Of course Gale wasn't really asking for a list of numbers; he wanted to know what pattern underlies the numbers. Getting a computer to answer questions of this kind is not so straightforward—unless you do the mathematical equivalent of consulting Google, that is, you look up the numbers in the Online Encyclopedia of Integer Sequences, maintained by Neil J. A. Sloane of AT&T Research. The search retrieves sequence number A001109, along with a wealth of related lore. It turns out that each perfect median m is the square root of a triangular number. In other words, m^2 dots can be arranged to form either a square or an equilateral triangle. This is a geometric connection I never would have discov-



Open racks of vacuum tubes were a stylistic statement in the first electronic computers. This machine is the EDSAC (Electronic Delay Storage Automatic Calculator) at Cambridge University. The paper-tape reader for input and teleprinter for output are at the far right. The first programs run on the EDSAC were simple demonstrations that calculated square numbers and primes. Later the machine was put to work in a problem-solving mode. Photograph courtesy of the Computer Laboratory, Cambridge University.

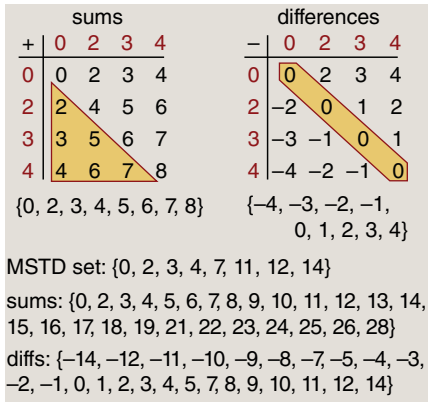


A sequence of counting numbers 1, ..., n has a perfect median m if the numbers less than m have the same sum as the numbers greater than m. For example, 6 is the perfect median of 1, ..., 8, since the sums on both sides are 15. Any sequence 1, ..., k has the sum $k(k+1)/2$, which is also the formula for a triangular number. In a sequence with perfect median m, the numbers to the left of m sum to $(m–1)m/2$, and so do those to the right of m. The sum of the entire series is therefore $(m–1)m/2 + m + (m–1)m/2$, which simplifies to m^2. The sum of 1, ..., n is also $n(n+1)/2$. Hence m is a perfect median if m^2 is a triangular number. For the example shown $m^2 = n(n+1)/2 = 36$, forming a square of side 6 and a triangle of side 8.

ered on my own. (For an explanation of the connection between squares, triangles and medians, see the illustration above.)

Sums and Differences

Take a set of integers, say {0, 2, 3, 4}, and calculate the sums of all possible pairs of numbers drawn from the set. A set of four numbers yields 16 pairs, but not all the sums are necessarily distinct. In this case there are just eight different sums: {0, 2, 3, 4, 5, 6, 7, 8}. Now build the analogous set of pairwise differences; it turns out there are nine of them: {–4, –3, –2, –1, 0, 1, 2, 3, 4}.



Sets of integers generally have more pairwise differences than sums. Because addition is commutative, the upper and lower triangles of the sum matrix are mirror images, eliminating almost half the entries as duplicates. In subtraction, entries across the diagonal differ in sign. (The diagonal itself is all zeros.) Only a few anomalous sets have more sums than differences (MSTD).

If you try the same experiment with a few more small sets of numbers, you may be ready to guess that the sums *never* outnumber the differences. And there's a plausible rationale to back up this conjecture: Addition is commutative but subtraction isn't. The sums 5+8 and 8+5 both yield 13, whereas 5–8 and 8–5 produce two distinct differences, –3 and +3. Nevertheless, the conjecture is false. A counterexample is the eight-member set {0, 2, 3, 4, 7, 11, 12, 14}, which has 26 distinct pairwise sums but only 25 differences. It is called an MSTD set (for "more sums than differences").

I first learned about MSTD sets in publications by Melvyn B. Nathanson of Lehman College in the Bronx, Kevin O'Bryant of the College of Staten Island and Imre J. Ruzsa of the Mathematical Institute of the Hungarian Academy of Sciences. (I have written about their work earlier at bit-player.org.)

A program to search for MSTD sets can take a direct approach to the problem. For each set of numbers, the program forms all pairwise sums and then eliminates duplicates; it does the same for the differences, and then compares. The trickiest part of the program turns out to be the routine for generating the sets of integers to be tested. The sets are characterized by two parameters: the number of elements n and the size of the largest element m (which cannot be less than $n–1$). For any given values of n and m, the sets can be ordered from smallest to largest and enumerated in a way that's something like ordinary counting, but you have to be careful that a set never has duplicate elements.

This process of enumerating sets and checking all sums and differences sounds arduous, but it goes faster than you might expect. For example, in less than a second of running time you can establish that the example mentioned above, {0, 2, 3, 4, 7, 11, 12, 14}, is the smallest MSTD set with eight elements. (Peter V. Hegarty of Chalmers University of Technology has since shown that there are no MSTD sets with fewer than eight elements, so this is the smallest example overall.) Checking all sets with $n = 11$ and $m \le 20$ takes less than a minute; there are 184,756 of these sets, and 160 of them are MSTDs, including a dozen where the sums exceed the differences by 2.

The search for MSTD sets is a peculiar kind of quest that seems to be possible only in mathematics. The sets are very rare, and yet there are infinitely many of them.

ABCs

My third example is another pursuit of shy, elusive mathematical objects. It concerns the simple equation $a+b=c$, where a, b and c are positive integers that have no divisors in common (other than 1); for example, the equation $4+5=9$ qualifies under this condition.

Now for some number theory. Multiply the three numbers a, b and c, then find all the prime factors of the product. From the list of factors, cast out any duplicates, so that each prime appears just once. The product of the unique primes is called the radical of abc, or $rad(abc)$. For the triple {4, 5, 9}, the product is $4\times5\times9=180$, and the factor list is 2, 2, 3, 3, 5. Removing the duplicated 2s and 3s leaves the unique factor list 2, 3, 5, so that $rad(180) = 30$.

In this example, c is less than $rad(abc)$. Can it ever happen than c is greater than $rad(abc)$? Yes: The triple {5, 27, 32} has the product $5\times27\times32=4{,}320$, for which the unique primes are again 2, 3 and 5. Thus $c = 32$ is greater than $rad(4{,}320) = 30$. Triples where c exceeds $rad(abc)$ are called abc-hits. As with MSTD sets, there are infinitely many of them, and yet they are rare. Among all abc triples with $c \le 10{,}000$, there are just 120 abc-hits.

If c can be greater than $rad(abc)$, how much greater? It's been shown that c can exceed $rad(abc)$ plus any constant or $rad(abc)$ multiplied by any constant. How about $rad(abc)$ raised to some power greater than 1? A conjecture formulated by Joseph Oesterlé of the University of Paris and David W. Masser of the University of Basel claims there are only finitely many exceptional cases where $c > rad(abc)^{1+\varepsilon}$, for any ε no matter how small. The conjecture has made the search for abc-hits more than an idle recreation. If the conjecture could be proved, there would consequences in number theory, such as a much simpler proof of Fermat's Last Theorem.

In a program to search for abc-hits the one sticky point is factoring the product abc. Factoring integers is a no-

torious unclassified problem in computer science, with no efficient algorithm known but also no proof that the task is hard. If you want to get serious about the search, you need to give some thought to factoring algorithms—or else latch on to code written by someone else who has done that thinking. On the other hand, for merely getting a sense of what *abc*-hits look like and where they're found, the simplest factoring method—trial division—works quite well.

Searchers for *abc*-hits can also join ABC@home (www.abcathome.com), a distributed computing project.

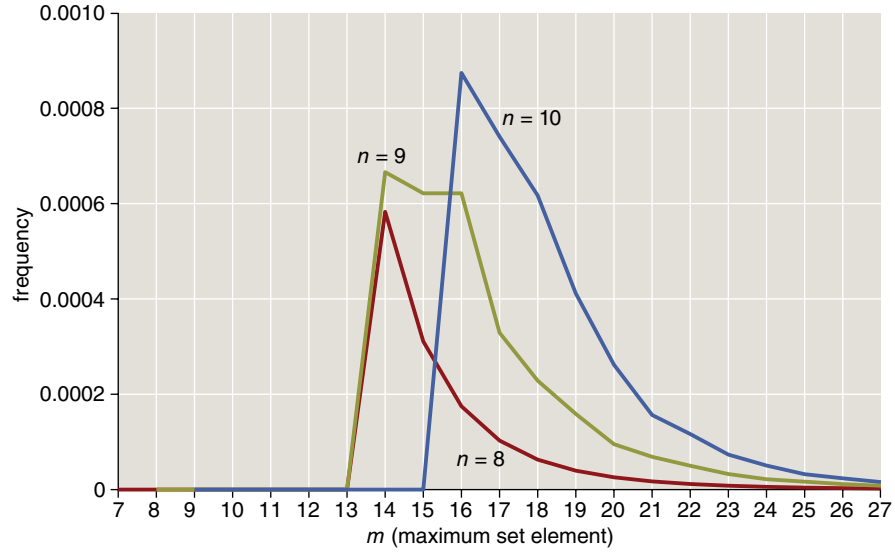
The BASIC Story

The programs I've been describing are simple, small and straightforward; writing them requires no arcane wizardry. On the other hand, writing them is not quite as easy as it ought to be, because the tools available for this kind of programming have not kept up with progress in software technology.

For inquisitive programming, the great age of innovation came and went in the 1960s. The best-known artifact of the era was the BASIC programming language, created in 1964 by John Kemeny and Thomas Kurtz of Dartmouth University. Kemeny and Kurtz set out to broaden the spectrum of computing enthusiasts; they were especially eager to draw in students in the liberal arts and the social sciences. *(Calculemus!)* Their new programming language was meant to lower the barriers to entry.

But the language was just the start. BASIC was designed in conjunction with the Dartmouth Time Sharing System, an early experiment in interactive computing. Elsewhere, batch processing was still the rule: Deliver your shoebox of punchcards in the afternoon, pick up a ream of printouts in the morning. DTSS and BASIC offered a direct connection to the computer via a teletypewriter or, later, a video terminal. Programming became more like a conversation with the computer. Compared with most other computing environments of the time, it was well suited to an exploratory style of problem solving.

BASIC spread from Dartmouth to other universities in the 1960s, then it gained a mass audience a decade later when Bill Gates and Paul Allen wrote a BASIC interpreter for microcomputers. This was the first product of the company that became Microsoft. The initial model of the IBM PC had a BASIC interpreter permanently inscribed in read-only memory; indeed, this was the only software supplied with the machine (even the operating system was an extra-cost option).



MSTD sets (with more sums than differences) are rare and unevenly distributed. A set can be characterized by the number of elements *n* and the maximum element *m*. Within a narrow range of values for these parameters, the abundance of MSTD sets reaches a sharp peak; the frequency is close to zero elsewhere. In this context the frequency signifies the fraction of all sets with a given *n* and *m* that are MSTD sets.

Whatever happened to BASIC? Its main attraction was also its undoing. As a language for beginners, it had the taint of training wheels. And it attracted the scorn of those who wanted to make programming a professional engineering discipline. Edsger Dijkstra, the curmudgeon-in-chief of computer science, groused: "It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration."

BASIC didn't disappear, but in response to such acid critcism, it was transformed beyond recognition. In the 80s we got "structured BASIC." Later, when the next fad swept the world of software, BASIC became an "object-oriented" language, with features for creating windows and menus and other gadgets that make up a graphic user interface. The surviving versions of the language are doubtless superior in many ways, but they have become tools for software development rather than for inquisitive programming.

Several other languages of the 1960s also offered an environment suited to inquiry rather than development. Logo has suffered a fate similar to BASIC's: It was designed as a programming language for children, and so adults were reluctant to take it seriously. In fact Logo is a very expressive language, an offshoot of Lisp.

Lisp itself (invented in the late 1950s) is my own favorite language for inquisitive programming. Most Lisp systems allow an incremental and interactive style of work: You write and test individual procedures rather than building monolithic programs.

There's also APL, a terse mathematical notation invented by Kenneth Iverson in 1962. Again, APL was intended mainly for problem solving rather than software development.

All of these languages still exist; indeed, each of them has its devoted following. But they are not where the energy is in computing today. As niche products, they have a hard time keeping up with changes in technology and attracting investment. Meanwhile, facilities for other kinds of programming grow steadily more luxurious. If you develop software in Java or C++, you get leather upholstery, walnut paneling and a dozen cupholders. If your vocation is inquiry-based programming, you sit in a folding chair.

A Wish List

The dream of Kemeny and Kurtz was "programming for everyone," based on the conviction that getting answers out of a computer should be seen as

an essential skill in a technological society. But the standard apparatus of software development is far from ideal for teaching or learning this kind of programming. If computing is to be a broadly practiced art form, we need an improved infrastructure for inquisitive programming. Here's my wish-list for a programming environment:

It should be self-contained, easily installed (and uninstalled), well-documented, internally consistent, bug-free and foolproof. Perhaps it's a scandal that GUI-coddled computer users have forgotten how to install an executable in usr/local/bin and add it to their $PATH, but that's the way the world is.

I don't want just to write programs; I want to have a dialogue with the computer. I need to know the answer to one question before I can decide what to ask next. Thus the typical unit of interaction should be a single statement or expression. This kind of incremental computing can't work if a language requires a lot of scaffolding—header files, a "main" procedure—just to run a fragment of code.

I should be able to think in terms of the problem at hand, not the innards of the machine. I don't want to worry about the encoding of characters or whether binary numbers come big end or little end first. I want to be insulated from annoyances such as allocating and reclaiming memory. Data-type declarations should be optional.

I want a system that's mathematically well-behaved. I should not have to live in a world where the integers end at 4,294,967,295 or where $\frac{1}{3} \times 3$ is equal to something other than 1. Give me unlimited precision and exact arithmetic, at least for rational numbers.

I want a rich selection of ready-made functions and data structures. I shouldn't have to build lists or trees or queues out of lower-level constructs such as pointers.

Pamper me. I want the source-code editor with the leather seats, the walnut and the cupholders.

I want a thriving community of enthusiasts, who contribute to the system, share code and knowledge, and keep us all feeling young. For this to happen, the software needs to be free, or close to it.

Along with the wish list there's an unwish list of things I'm willing to give up. Computational efficiency is one of them. I'll typically spend longer writing (and testing and debugging) a program than I will running it. Another variable I'm willing to leave unspecified is the programming language itself. I fervently believe that some languages are better than others, but the programming environment is more important.

Making It Happen

Will I ever get my wishes? Nothing I've asked for is unattainable or even particularly novel. I wouldn't know enough to propose these ideas if I hadn't seen them at work in the past. Indeed, over the years I have used a number of programming systems that met most of the criteria on my list. Some of those systems were pure delight. What makes me grumpy is that they are now obsolete and will not run on modern hardware. And I don't see replacements on the horizon.

My sentimental attachment to Lisp makes me long for a revival of interest in that language, but I don't see it happening.

A smarter bet is Python. It has a large and zealous community of boosters, especially in the world of science. The language itself is well adapted to inquisitive programming, and it serves perfectly well for introductory programming courses. Python's weak point, in my view, is the surrounding infrastructure of editors, command-line tools, code libraries and modules. There are several interactive programming environments for Python, but none of them give me the feeling of a place where I am at home and in control of my own environment.

Going off in another direction, programs such as Mathematica and MATLAB are truly wonders of the age. But software that costs more than the computer it runs on is not going to win the public over to casual programming.

A project called Sage claims as its mission, "Creating a viable free open source alternative to Magma, Maple, Mathematica and Matlab." Sage was initiated by William Stein of the University of Washington, and it now has hundreds of contributors. Sage is certainly not the self-contained, cohesive package that I argue for; on the contrary, it's a loose confederation of dozens of more specialized programs, such as GAP for group theory and R for statistics. All of the pieces are stitched together with Python code, which is also supposed to provide a consistent user interface. It works better than I would have guessed, but it's hardly seamless. The principal developers are putting most of their energy into creating a research tool for advanced mathematics, which can leave the skittish beginner without a safe point of entry.

One aspect of Sage I find particularly intriguing is the "notebook" interface, which runs in a web browser. Factoring a polynomial feels just like shopping at Amazon. Moreover, the interface is exactly the same whether the browser is connected to a locally installed copy of Sage or to a remote server. (There's a public Sage server at www.sagenb.org.) This is surely the way of the future. Perhaps the next big step in inquisitive programming will be offered as a service, not as a product. And maybe they'll call it *Calculemus!*

Bibliography

Berlekamp, Elwyn, and Joe P. Buhler. 2005. Puzzles column. (Problem 1, attributed to David Gale.) *Mathematical Sciences Research Institute Emissary*, Fall 2005, p. 3.

Campbell-Kelly, Martin. 1992. The Airy tape: An early chapter in the history of debugging. *IEEE Annals of the History of Computing* 14(4):16–25.

Dijkstra, Edsger W. 1975. How do we tell truths that might hurt? www.cs.utexas.edu/~EWD/transcriptions/EWD04xx/EWD498.html

Granville, Andrew, and Thomas J. Tucker. 2002. It's as easy as abc. *Notices of the American Mathematical Society* 49(10):1224–1231.

Hayes, Brian. 1983. Computer recreations: Introducing a department concerned with the pleasures of computation. *Scientific American*, October 1983, pages 22–36.

Hayes, Brian. 2006. Counting sums and differences. bit-player.org/2006/counting-sums-and-differences.

Hayes, Brian. 2007. Easy as abc. bit-player.org/2007/easy-as-abc

Kemeny, John, and Thomas B. Kurtz. 1964. BASIC. Hanover, N.H.: Dartmouth College Computation Center. www.bitsavers.org/pdf/dartmouth/BASIC_Oct64.pdf

Oliphant, Travis E. 2007. Python for scientific computing. (Special issue on Python.) *Computing in Science and Engineering* 9(3):10–20.

Renwick, W. 1950. The E.D.S.A.C. demonstration. In *Report of a Conference on High Speed Automatic Calculating-Machines*, Cambridge University Mathematical Laboratory.

Sloane, Neil J. A. The Online Encyclopedia of Integer Sequences. Sequence A001109. www.research.att.com/~njas/sequences/A001109

Stein, William, and David Joyner. Sage Programming Guide. www.sagemath.org/doc/html/prog/prog.html

Wheeler, Joyce M. 1992. Applications of the EDSAC. *IEEE Annals of the History of Computing* 14(4):27–33.