IEEE SPECTRUM ONLINE

July 2005

intellectualproperty

Software Patents Don't Compute

No clear boundary between math and software exists

First of two articles on software patents

By Ben Klemens

In 1997 the U.S. Patent and Trademark Office granted Amazon.com a patent for "one-click shopping"—a system that lets customers make purchases without having to go through an online checkout. The patent started a fierce debate in both the business and the technical press. Critics felt the Amazon.com patent was the poster child for everything that was wrong with software patents, charging that such patents allowed obvious applications of existing technology to be wrapped up in intellectual property monopolies.

The Amazon.com patent is no fluke. Consider the following recently issued patents:

- Method and system for solving linear systems (U.S. Patent No. 6078938).
- Cosine algorithm for relatively small angles (No. 6434582).
- Method of efficient gradient computation (No. 5886908).
- Methods and systems for computing singular value decompositions of matrices and low rank approximations of matrices (No. 6807536).

The arcane details of these patents are not relevant. What is relevant is that these patents are for purely mathematical algorithms, and for centuries prior to the 1990s, mathematics was not patentable. So how did these patents come to be granted?

By U.S. law, scientific principles may not be patented. Electromagnetism, the theory of relativity, and a menagerie of quantum particles were all discovered after the inception of the U.S. Patent and Trademark Office, now based in Alexandria, Va. Yet none of these discoveries could have received patents, because until the early 1990s it was universally agreed that mathematical algorithms were in the category of scientific principles that could not be owned by an individual.

What has changed is that mathematics has become increasingly reliant on machines. Abstract algorithms that involve inverting large matrices or calculating hundreds of coefficients in a sequence are routine today and of only limited use without physical computers to execute them.

Conversely, devices such as video drivers, network interface cards, and robot arms depend on algorithms for their operation. Because of the machine-intensiveness of modern mathematics and the math-intensiveness of modern machines, the line between mathematical algorithms and machinery is increasingly blurred. This blurring is a problem, because without a clear line delimiting what is patentable and what is not, creative entrepreneurs will eventually be able to claim sole ownership of abstract mathematical discoveries. But how do we draw a line that would ensure that mathematical algorithms are not patentable while innovative machines are?

THE EASIEST LINE TO DRAW would be simply to say that if an invention is physical, then it should be patentable, and if it is abstract, then it should not be. But what do we do with inventions that involve both the physical and the abstract? For example, the case of Diamond v. Diehr involved a rubber-curing machine that relied on a significant amount of software to control the machine's timing. The U.S. Supreme Court ruled in 1981 that the patent was for industrial equipment, not an abstract algorithm, and thus the overall patent—software plus machine—was valid. But the court left a key question hanging: how much physical invention is necessary before the overall device is patentable? If all of the inventiveness is in the algorithm, which is then applied in a trivial manner to a simple machine, is the overall patent okay?

In a long series of rulings, culminating in 1994 with In re Alappat and In re Lowry, the U.S. Court of Appeals for the Federal Circuit ruled that an uninventive physical component added to an inventive abstract component makes the whole patentable. In other words, "a new algorithm to calculate Fourier transforms" is not patentable, but "a stock PC on which is programmed a new algorithm to calculate Fourier transforms" has enough of a physical component to be patentable.

Further, the court ruled that since a computer is so integral to a computational algorithm, patent examiners are obliged to assume that one exists. If an application is for "a pure computational algorithm," then the examiner must read it as if the words "a computer on which is programmed" had been prepended to the description of the algorithm.

This is the bottom of the slippery slope: there is no longer any meaningful barrier to the patenting of abstract algorithms. The use of any inventive mathematical algorithm that requires more calculation than can be reasonably done by hand is now patentable.

ANOTHER APPROACH MIGHT BE to distinguish between the pure mathematical algorithm, which should not be patentable, and its application to real-world problems, which should be.

For example, the case of Gottschalk v. Benson concerned the patentability of a program to convert between binary-coded decimal and plain old binary. Evidently, this was too close to unapplied pure math; the Supreme Court struck down the patent in 1972, because "the patent would wholly pre-empt the mathematical formula and in practical effect would be a patent on the algorithm itself."

In contrast, State Street Bank and Trust Co. v. Signature Financial Group Inc. was a suit over alleged infringement of State Street's patented system for doing the bookkeeping for a suite of mutual funds. The system did not push around physical objects, but the Court of Appeals for the Federal Circuit ruled that the share prices and other numbers it derived still have a real, tangible effect and may therefore be considered to be a valid subject for a patent. So this attempt to distinguish the patentable from the unpatentable is too unreliable.

TO FIND OUT WHY all these distinctions fail, we turn to one of the founders of computer science, Alan Turing. In 1936, Turing described a theoretical computer that is effectively equivalent to every computer in existence today. His design included an infinitely long tape and read/write head, which did different things depending on the data on the tape and the machine's state. Because different states cause the machine to do different things, his contraption is often called a state machine.

A modern PC is equivalent to Turing's tape and head—a physical device that can store data and execute various operations. The programs that instruct the computer's operation generate the states that dictate how it will operate: they make up the impermanent information that guides the computer, but they do not change its fundamental design or composition.

Because almost all modern programming languages encompass the ability to turn a computer into a state machine, they are all, at a deep level, equivalent—a program written in one such language can be directly translated to any other language, such as from Perl to C++ or from Microsoft Visual Basic to Lisp. So all software is essentially made of the same stuff.

In 1936, Alonzo Church proved that that stuff is mathematics. Church created lambda calculus, a formal means of writing mathematical expressions and also a tool that can be used to program a state machine. That is, any program written in a language such as C is a trivial translation of a set of purely mathematical lambda-calculus expressions.

So where is the line drawn between software and mathematical expression? Based on Church's and Turing's work, there is none. Any legal attempt to force a wedge between pure math and software will fail because the two are one and the same. A patent on a program is a patent on a mathematical expression, regardless of whether it is expressed in C, Lisp, or lambda calculus.

BUT WHILE DEMOLISHING the distinction between software and math, Turing and Church's work offers a natural division between patentable machinery and unpatentable mathematics—exactly what we have been looking for. Let the devices that implement state machines—physical objects such as computers—be patentable, and the states to which they are set—information such as programs and data—remain unpatentable. The distinction meets the goal of ensuring that pure mathematics is not patentable while letting those who design faster and better computing devices patent their inventions.

The distinction is clear, and it offers no slippery slope down which the courts could slide. An innovative field-programmable gate array (FPGA) is a state machine and so would fall on the patentable side of this fence, while code loaded onto the FPGA would be an unpatentable state to which the state machine has been set.

A Java machine constructed on an application-specific integrated circuit (ASIC) would be a state machine, but a Java machine existing only in software running on a general-purpose central processing unit would be a state. A robot arm would be a state machine, but its device driver would be a state.

The courts failed to review the mathematics literature and as a result made several vain attempts to reinvent the wheel. Software and lambda calculus are in the same equivalence class, which means any law that allows software to be patentable allows the patenting of the evaluation of certain mathematical expressions.

But, fundamentally, if we are to disallow the patenting of pure scientific and mathematical discoveries to foster basic research and innovation, the only way to do so is to disallow the patenting of the states to which state machines may be set—that is, to abolish software patents.

Editor's Note: This is the first of two articles on software patents. This article focuses on how the U.S. patent system attempts to draw a dividing line between patentable machines and unpatentable mathematics—and why the system is failing. Next month's article will discuss the economic and legal impact of software patents and a proposed solution.

ABOUT THE AUTHOR
Ben Klemens has a Ph.D. in social sciences from California State Polytechnic University, in San Luis Obispo. He is currently a guest scholar at The Brookings Institution, Washington, D.C. His book *Math You Can't Use: Patents, Copyright, and Software* is to be published by the Brookings Institution Press.

ILLUSTRATION: DAVID RODRIGUEZ