Newton's contribution and the computer revolution*

Stephen Smale
University of California, Berkeley

My goal in this talk is to get some kind of understanding of the computer, maybe a little differently than is traditionally done. I want to look at the computer more from the point of view of a physicist than that of an engineer. My goal is not to design a better computer but to understand how the computer works in practice. I will talk about a search for the laws of computation – a mathematical picture for the process of computation.

For that reason it is useful to go back to Newton's *Principia* which is a conclusion of a scientific revolution. Learning how to look at the world from a physicist's point of view is a great lesson. Let me then look at some aspects of Newton's *Principia* of 300 years ago. The theme that I want to pursue is the relationship between the discrete and the continuous. In some sense the material world and the computer are both discrete objects. Yet the mathematics for the material world is continuous (the mathematics of physics is certainly continuous), and I hope eventually continuous mathematics will play a more substantial role for the computer than it has today.

In the *Principia* Newton took the world to be the world of Democritus – the atomistic world. That was way before quantum theory and before atoms were really understood. What it means is that if you take

---

* Public lecture given on 28 May 1988 and organised by the Department of Mathematics, National University of Singapore, Singapore Mathematical Society and Singapore National Academy of Science. Transcribed and edited by Y. K. Leong. This transcript has not been read by Professor Smale.

Professor Smale was Third World Academy of Science Lecturer at the International Conference on Numerical Mathematics, Singapore, held at the National University of Singapore from 31 May 1988 to 4 June 1988. He is a professor of mathematics in the University of California, Berkeley, since 1964. His ground-breaking work in differential and algebraic topology won him a Fields Medal (the mathematical equivalent of a Nobel Prize) in 1966. His recent research seeks to unify numerical analysis and theoretical computer science.

any bounded region in the universe, then there is only a finite number of particles in the region, each particle being indivisible. In Newton's time, this was called the *corpuscular* view of the universe. In recent decades, this view has been verified. Newton's mathematics came especially from Euclid. In fact, one of the main goals, some say the biggest success, of the *Principia* is to derive the elliptical orbits of the planets. These orbits were, you might say, conjectured by Kepler and given a strong mathematical foundation in the *Principia*. These are continuous objects – in complete contrast to the atomic picture. The set of real numbers $\mathbb{R}$ is a paramount example of the continuous, and Newton used it in his analysis of the differential equations of motion, which formed the nucleus of the *Principia* in deriving the Kepler orbits from the laws of motion. It was a big problem for Newton to reconcile the discrete picture of the world with the continuous picture of the mathematics. As Thomas Kuhn, a historian of science, wrote in his book *The Copernican Revolution*, this contrast between the discrete and the continuous prevented Newton from publishing the *Principia* for many years. What finally emerged is this picture. For example, Newton took the Earth to consist of a finite number of particles and computed the gravitational pull of the sun on these particles. Then he put in more particles and, in the limit, he got some kind of result saying that if you treat the Earth as continuous, the effect of the sun's gravitation on the Earth is the same as that on a single particle at the centre of mass of the Earth. Only after Newton had reconciled the contrast between the discrete and the continuous by taking the limit of a finite number of particles to get a continuous picture of matter did he publish the *Principia*.

Let me quote Thomas Kuhn:

> "In 1685 he [*Newton*] proved that, whatever the distance to the external corpuscle, all the earth corpuscles could be treated as though they were located at the earth's centre. That surprising discovery, which at last rooted gravity in the individual corpuscles, was the prelude and perhaps the prerequisite to the publication of the *Principia*. At last it could be shown that both Kepler's laws and the motion of a projectile could be explained as the result of an innate attraction between the fundamental corpuscles of which the world machine was constructed."

In subsequent parts of the *Principia* Newton went ahead to use these ideas to treat, not altogether successfully, what we call "continuum mechanics" – elasticity, fluid mechanics. In retrospect, we can say what he

had found was more or less a programme for doing these things. Eventually it was Euler, Bernoulli and Lagrange who actually successfully carried out the derivation of the partial differential equations of continuum mechanics. But it was based on and inspired by the programme of the *Principia*. I describe this picture to give you some perspective of the situation of the computer today. Let me pass then to this century and talk about in some sense the most significant theoretical basis due to Alan Turing for understanding the computer and theoretical computer science today .

Many of you, of course, know about Turing. There is a lot of attention about Turing in the last few years due apparently to a book *Enigma* by Andrew Hodge. When I was in New York last fall, there were two Broadway plays – maybe one Broadway and one opera – about the life of Turing. It's amazing to have going on at the same time two plays about a mathematician. I went to see one of them – called "Breaking the Code". It was a very successful Broadway play. It was full house when I went there. "Breaking the code" refers to two aspects of Turing's life. (Turing was a very interesting but tragic figure.) One is that Turing was the person primarily responsible for breaking the German code during World War Two. Also, there was the question of breaking the moral code in the 1950's. He was, in fact, convicted of homosexuality in 1952. To avoid going to jail, he had to take hormone treatment for a year. In 1954 he committed suicide by eating an apple which he had poisoned.

What I want to talk about today is his contribution to computer science in what we call "*Turing machines*". A Turing machine is the model of a computer universally used by people working in theoretical computer science. Roughly speaking, it is a tape on which are listed a finite number of 0's and 1's. You have a pointer which moves back and forth in some kind of a program. An abstract Turing machine is a machine with a list of 0's and 1's in a certain order proceeding back and forth, changing 0 to 1 and so on, and eventually one has an output with a different number of 0's and 1's. Maybe the machine never stops. What I want to emphasize is its discrete nature: only 0's and 1's are allowed on the tape. This is a very successful picture for analysing the main problems of theoretical computer science. A particular machine which Turing called the "*universal machine*" will duplicate the work of any other Turing machine and is related to the work of Gödel and is eventually used to prove things in mathematics like the undecidability of Hilbert's 10th Problem. Ideas of Gödel, Turing and others are used to show that there is no algorithm for solving Diophantine equations.

Personally, my mathematics has always been continuous mathematics – calculus mathematics. I have never been exactly happy with this picture of the computer. This is a model one can never do calculus on. But there are more basic criticisms to the Turing machine. Let me list them.

(a) It does not relate to scientific computing. Scientific computing is the type of computing which makes the biggest use of the computer. It starts with the partial differential equations of fluid mechanics, continuum mechanics. It is the kind of computation needed to understand turbulence for airplanes and weather predicting better. That body of usage involving differential equations cannot be viewed in terms of these machines or even recipes, methods or programs. The theoretical side of this is numerical analysis. Most numerical analysts see no use at all for the Turing machine nor, in fact, for the complexity theory of computer science. On the other hand, computer scientists often advise students to take calculus less and less. These two communities are diverging more and more. This is all so paradoxical since both are so involved with the computer. I think the big problem here is that the model of the Turing machine which computer scientists depend on is fairly useless for understanding the algorithms and programs of numerical analysts. For example, using the Turing machine, one can look at the problems of discrete mathematics and talk about all the algorithms of solving those problems. In numerical analysis, people talk about algorithms all the time but they do not define the word "algorithm". The reason is that there is no model of the computer that will allow them to talk about all possible algorithms.

(b) Another criticism is related to the notion of multiplication. In scientific computing multiplication does not depend on the size of the numbers. Even for small desk-size computers, if you take two numbers and multiply them – it does not matter how big they are – they are knocked off in the 8th or 10th decimals of precision. On the other hand, in the Turing model, multiplication depends on the size in terms of the logarithm of the number: the cost of multiplication for the Turing machine is equal to $\log |x|$. The idea is when you expand x in terms of bits (binary or decimal expansion of x), multiplication in theoretical computer science is going to depend on size in this sense. This is where scientific computing is different from theoretical computer science.

(c) Before computer science became highly developed, von Neumann (one of the big names in the foundations of computer science) had written on the subject. He writes, "The theory of automata of the digital type [*the automata being the computer*] is certainly a chapter in formal logic.

It therefore seems that it would have to share the unattractive property of formal logic. It will have to be from the mathematical point of view combinatorial rather than analytical." And he says analysis is technically the most successful and best elaborated part of mathematics. Formal logic by the nature of its approach is cut off from this part of mathematics. So the third criticism of the Turing machine is that it cuts off computer science from the most developed part of mathematics.

If one can have a unification of computer science with mainstream mathematics one can use the mathematics of analysis to understand computing. We can go back to the way Newton dealt with the universe. I would like to suggest that one could use a real model of the computer. I am going to talk about a "real" machine. I am taking a real number x and imagining it as an input of a machine. The computer scientist would want to find out how you are going to type a real number into the machine. Well, in some sense, one cannot really do that because a machine can only take in a finite number of decimals. In practice, a machine takes in only rational numbers. But my point of view is to idealise and to axiomatize. Just like the universe which consists only of discrete particles. It is best understood in terms of differential equations through Newton's idealisations. In the same way, I am suggesting that the machine be idealised to take in some object from the real numbers so that eventually one can use methods of analysis just as Newton idealised the universe by making it continuous. So this is going to be an idealisation but I would say this is not such a terrible idealisation because, for example, between 0 and 1 the machine will input 10^8 rational numbers so that any real number between 0 and 1 will have a high degree of approximation. In scientific computation this is a good approximation. Of course, one has to be careful about round-off errors. There is a whole science of that in numerical analysis which is able to deal with questions of round-off errors.

The fact is that the actual machine is not exactly a real machine. Moreover, we understand from modern physics that idealisations are not unique. For the universe we have Newton's idealisation and we also have the quantum picture and the relativity picture. All these different pictures give different insights of the universe. I should say that there is no unique ideal way of looking at the computer. The Turing machine is fine in lots of context, but in the context of scientific computing one requires something more like a real machine.

Let me describe a little bit some recent work on the theory of a real machine. This is joint work with L. Blum and M. Shub. The idea is to

have a theory of computation over the real numbers. For one thing we surely do not want to lose everything that computer science has developed. We want to keep the structure and beautiful theorems that are already developed in computer science. So the way of doing that is as follows. We require a machine over a ring R. For example, the ring R could be the integers $\mathbb{Z}$ or it could be the rational numbers $\mathbb{Q}$ or the real numbers $\mathbb{R}$. Our idea is that these are not Turing machines. Since they are defined over a ring, everything is done more algebraically. Turing machines come from logic, so the whole foundation of Turing machines is built with logical notations.

For a machine M we define an input-output function ϕ_M, which does the following. The machine reads in some input, the machine runs and you may get some output. So the input-output function is a map which takes in an input and gives an output. Now if $R = \mathbb{Z}$ then the input-output functions are precisely the ones given by Turing machines; they are the classical computable functions. Logicians call them "partial recursive functions".

Our object is to take R to be the real numbers $\mathbb{R}$ and we can then study the algorithms of numerical analysis. One can talk about computable functions over $\mathbb{R}$. In this way we can describe a model for all the algorithms for solving problems in numerical analysis. Let me mention three results.

(a) *There exists a universal machine over any ordered ring R. The universal machine is independent of the ring. It does not use Gödel coding.*

The Gödel coding reduces everything to the integers and destroys the algebra. If there were an algebraic structure, the universal machine of Turing will destroy it with the Gödel coding. Our universal machine retains the algebraic structure and is, in some sense, bigger.

(b) *Partial recursive functions over R.*

This is an intrinsic description of the input-output functions of our machines over a ring R.

(c) *NP-completeness over R.*

This is at the heart of complexity theory of computer science. There is one theorem which is the core of the subject and is due to Cook – a beautiful theorem proved about 18 years ago. It is the starting point of the whole subject of P versus NP. The idea is to define *polynomial time algorithms*

or *polynomial time machines* which will compute things in "polynomial time". Tractable problems, or problems which do not require "exponential time" to solve, are placed in the class P. There is another class NP, introduced by Cook, called "*nondeterministic polynomial*" which is easy to confirm for an algorithm but it is not clear whether the algorithm is fast or not. The main theorem of computer science is "$P = NP$", or as is more likely, "$P \neq NP$". This is the theorem that is compared by people in computer science to the Riemann Hypothesis in importance. The idea is that NP is easy to check; P is very fast. If $P = NP$ it means that a lot of problems in discrete mathematics would be tractable. What we did was to look at that problem over the real numbers.

Cook's Theorem says that there exists one problem in NP which every problem can be reduced to. Without being formal, we have: any problem in NP can be transformed to a special problem, and this is a good reduction to the special problem. The special problem of Cook turns up in discrete mathematics as a satisfiability problem involving lots of 0's and 1's. What we showed is that NP-completeness over the real numbers is the same thing, but now the special problem is a problem that one can understand in classical mathematical terms. We showed the existence of NP-completeness and the special problem over the real numbers is the following one.

Given a polynomial $f : \mathbb{R}^n \longrightarrow \mathbb{R}$ in n variables of degree 4. It is represented by its non-zero coefficients (what we call a "sparse representation"). They form the input to the problem, and the size of the problem as a measure of time is the number of non-zero coefficients of the polynomial. The question is:

$$\text{Does there exist } x \in \mathbb{R}^n \text{ such that } f(x) = 0?$$

It is a yes-or-no problem. This is very much like problems in classical mathematics. There are algorithms for solving this due to Tarski and Seidelberg: elimination theory. Those algorithms are not fast algorithms. They are slow algorithms. They are not polynomial time algorithms. What is not known is whether this problem can be solved in polynomial time. This seems to be a hard problem. What we showed is that any problem in NP over the real numbers can be transformed to that problem. This is very much inspired by Cook in classical computer science theory. The problem now is to study systematically what the machine can do to solve problems over the real numbers.