Analytical and Empirical Mathematics with Computers*

Stephen Wolfram
The Institute for Advanced Study, Princeton NJ 08540.

(June 1985)

Some of the practical, methodological and theoretical implications of computation for the mathematical sciences are discussed.

Computers are becoming an increasingly significant tool for research in the mathematical sciences. This paper discusses some of the fundamental ways in which computers have and can be used to do mathematics.

Computer-aided mathematical calculation [1]

The advent of electronic calculators made it easy to do arithmetic by computer, and made logarithm tables and the like obsolete. Now computer hardware is becoming powerful enough to be able to do not only arithmetic, but all kinds of conventional mathematics. But to realize this capability a computer language is needed. The language should be as close as possible to conventional mathematics, and should be able to incorporate as much mathematical knowledge as possible. Through its use, mathematical handbooks and tables of integrals should eventually become as obsolete as logarithm tables.

The language should be general, so that it can handle all kinds of mathematics. It should be interactive so that simple calculations are quick to do. It should be extensible, so that it can learn new mathematics. It should be efficient and heavy duty, so that it is able to do big calculations. And finally, it should be portable, and be able to run without significant change on many different kinds of computers.

There are numerical computer languages, such as FORTRAN, BASIC, APL and C. The intrinsic data types in these languages are numbers, or arrays of numbers. But mathematical calculations involve higher-level objects, such as algebraic formulae (say x^2-3xa^2). So to be able to handle many mathematical calculations one needs a language capable of symbolic manipulation of such objects.

Several "computer algebra" systems have been constructed. The prime examples are REDUCE and MACSYMA. The emphasis of these systems is on algebraic manipulation, involving calculations with polynomials and other algebraic objects. But to handle much of the mathematics that arises in

* Work supported in part by the U.S. Office of Naval Research under contract number N00014-85-K-0045.

practice, one needs a system with a more general basic structure. One needs a true general programming language for mathematical computation.

It was with this in mind that I designed the SMP language. SMP was first implemented in 1979-1981, in about 120000 lines of C code. It represents a general interactive system for mathematical computation. It incorporates a core of fundamental mathematical knowledge; new objects and operations can be defined in the SMP programming language. In addition, the SMP system includes such facilities as graphics and numerical C and FORTRAN code generation. There is also an expanding library of external files containing programs for specific applications.

```
#I[1]::  Ex[(x-1)↑3 (x+2)↑2]

                                2    3    4    5
#O[1]:   -4 + 8x - x  - 5 x  + x  + x

#I[2]::  Fac[%]

                          3    2
#O[2]:   (-1 + x)  (2 + x)

#I[3]::  Ar[4,f]

#O[3]:   {f[1],f[2],f[3],f[4]}

#I[4]::  Ar[5,$1!/Prime[$1]]

#O[4]:   {1/2,2/3,6/5,24/7,120/11}

#I[5]::  N[Log[%,2]]

#O[5]:   {-1,-0.584963,0.263034,1.77761,3.44746}

#I[6]::  Ar[{3,3},N[Psi[$1,$2]]]

#O[6]:   {{-0.577216,1.64493,-2.40411},{0.422784,0.644934,-0.404113},
                {0.922784,0.394934,-0.154113}}

#I[7]::  %-x Ar[{3,3}]

#O[7]:   {{-0.577216 - x,1.64493,-2.40411},{0.422784,0.644934 - x,-0.404113},
                {0.922784,0.394934,-0.154113 - x}}

#I[8]::  Det[%]

#O[8]:   0.323819 - 2.37808x
                 + (-0.695452 + (-0.577216 - x) (0.644934 - x)) (-0.154113 - x)

#I[9]::  Ex[%]

                                          2    3
#O[9]:   0.488368 - 1.29992x - 0.0863945 x  - x

#I[10]::  N[Sol[%=0,x]]

#O[10]:*  {x -> -0.212347 + 1.18258I,x -> -0.212347 - 1.18258I,x -> 0.3383}

#I[11]::  Ar[10,x↑$i-1]

                                2    3    4    5    6    7    8
#O[11]:   {-1 + x,-1 + x ,-1 + x ,-1 + x ,-1 + x ,-1 + x ,-1 + x ,
                       9    10
                 -1 + x ,-1 + x  }

#I[12]::  Map[Fac,%]
```

```
#O[12]:   {-1 + x,(-1 + x) (1 + x),(-1 + x) (1 + x + x^2),
                                                    2
          (-1 + x) (1 + x) (1 + x^2),(-1 + x) (1 + x + x^2 + x^3 + x^4),
                                                    2
          (-1 + x) (1 + x) (1 - x + x^2) (1 + x + x^2),
                                                2   3   4   5   6
          (-1 + x) (1 + x + x^2 + x^3 + x^4 + x^5 + x^6),
                                                    2     4
          (-1 + x) (1 + x) (1 + x^2) (1 + x^4),
                                                2     3   6
          (-1 + x) (1 + x + x^2) (1 + x^3 + x^6),
                                                    2   3   4
          (-1 + x) (1 + x) (1 - x + x^2 - x^3 + x^4)
                                                2   3   4
                  * (1 + x + x^2 + x^3 + x^4)}
#I[13]::  Map[Len,%]
#O[13]:   {2,2,2,3,2,4,2,4,3,4}
#I[14]::  Pos[4,%]
#O[14]:   {{6},{8},{10}}
#I[15]::  <XStat
#I[16]::  Mean[%13]
#O[16]:   14/5
#I[17]::  SD[%13]
#O[17]:   0.918937
```

Figure 1. A sample dialogue with SMP.

Figure 1 shows an example of a dialogue with SMP, in which SMP is used much like a symbolic mathematical calculator. The SMP language is designed to be as close to conventional mathematical notation and usage as possible. It is strongly based on pattern matching.

The command **a:x-1** assigns the symbolic expression **x-1** to be the value of the symbol or "variable" **a**. So wherever **a** appears, it is replaced by the expression **x-1**. Now the assignments **v[2]:x** and **v[3]:y** define values for components of the list **v**. With these assignments the object **v**, which can be considered as a vector or array, is defined to have value **{[3]:y,[2]:x}**. Making the assignment **v[p]:x+2** specifies a value for the element of **v** with symbolic index **p**. The value will be used whenever the literal expression **v[p]** appears. But no definition has been given for an expression like **v[q]**. To define a value for all "projections" of **v**, one makes the assignment **v[$x]:$x^2**. This specifies that the value of **v** when indexed with any expression represented by the "generic symbol" **$x** is the square of that expression. The assignment can thus be considered a definition for the "function" **v**. The notation makes it clear that a function can be viewed as an array with a continuous index. SMP always uses more specific definitions for a particular expression in preference to more general ones. Thus **v[p]** is replaced by **x+2**, but **v[q]** becomes **q^2**.

SMP includes a sophisticated pattern matcher, which makes it possible to implement complex transformation rules for patterns. So for example the definition **Acos[Sqrt[1-$x^2]] : Asin[$x]** specifies a simplification rule for inverse trigonometric functions. In general one can specify rules for mathematical functions and so on much as they are given in books of tables. Figure 2 shows an example.

```
SBer_: Ldist

        Ber[0] : 1

        Ber[$n_=Natp[($n-1)/2],0] : 0

SBer[1]:        Ber[$n] -> -Sum[Comb[$n+1,k] Ber[k], k, 0, $n-1] / ($n + 1)

SBer[2]:        Ber[$n,$x+$$y] --> Sum[Comb[$n,m] Ber[m,$x] $$y↑($n-m),m,0,$n]
                                                        /*R: MOS p. 25 */

SBer[3]:        Ber[$n_=Natp[$n],0] -> Ber[$n]

SBer[4]:        Ber[$n_=Natp[$n],1/2] -> -(1-2↑(1-$n))Ber[$n]

SBer[5]:        Ber[$n_=Natp[$n],1/4] -> (-1)↑$n Ber[$n,3/4]

SBer[6]:        Ber[$n_=Natp[$n],1/4] -> -2↑(-$n) (1-2(1-$n)) Ber[$n]\
                        $n 4↑(-$n) Eul[$n-1]

SBer[7]:        Ber[$n_=Natp[$n/2],5/6] -> Ber[$n,1/6]

SBer[8]:        Ber[$n_=Natp[$n/2],1/6] -> 1/2 (1-2↑(1-$n))(1-3↑(1-$n))Ber[$n]

SBer[9]:        Ber[$n_=Natp[$n],1-$x] -> (-1)↑$n Ber[$n,$x]

SBer[9]:        Ber[$n_=Natp[$n],$x] -> (-1)↑$n (Ber[$n,-$x]+$n (-$x)↑($n-1))

SBer[10]:       Sum[$m↑$n,$m,1,$N] -> 1/($n+1) (Ber[$n+1,$N+1] - Ber[$n+1])
                                                        /*R: MOS p. 26 */

SBer[11]:       Ber[$n_=Natp[$n],$x] -> Ber[$n,$x+1] - $n $x↑($n-1)
                                                        /*R: MOS p. 26 */

SBer[12]:       Sum[Comb[$n_=Natp[$n],$m_=Natp[$m-1]] Ber[$m,$x],$m,0,$n-1] ->\
                        $n $x↑($n-1)
                                                        /*R: MOS p. 26 */

SBer[13]:       Ber[$n_=Natp[$n],$x $m_=Natp[$m-1]] -> $m↑($n-1)\
                        Sum[Ber[$n]($x + 1/$m),1,0,$m-1]
                                                        /*R: MOS p. 26 */

                /**  Bernoulli numbers  **/

SBer[14]:       Ber[$n,0] -> Ber[$n]

SBer[15]:       Ber[$n,1] -> Ber[$n]

SBer[16]:       Ber[$n_=Natp[$n/2-1/2]] : 0
                                                        /*R: MOS p. 27 */

SBer[17]:       Ber[$n_=Natp[$n/2]] --> 2(-1)↑($n/2+1) $n!*\
                        Sum[(2 Pi m)↑(-$n),m,1,Inf]
                                                        /*R: MOS p. 27 */

SBer[18]:       Ber[$n_=Natp[$n/2]] -> 2(-1)↑($n/2+1)(2 Pi)↑(-$n) $n!*\
                        Zeta[$n]
                                                        /* MOS p. 28 */

SBer[19]:       Ber[$n_=Natp[$n/2-1]] -> -$n Zeta[1-$n]
                                                        /* MOS p. 28 */
        _XBer[Loaded]:1
```

Figure 2. An SMP external file containing some relations concerning Bernoulli polynomials.

The core of SMP incorporates a substantial body of mathematical knowledge and techniques. At the simplest level, it includes over 200 mathematical functions, covering for example all the standard special functions of mathematical physics (hypergeometric functions, elliptic functions, and so on). It incorporates many standard mathematical operations, including polynomial manipulation (expansion, factorization, etc.), equation solving (linear, polynomial, etc.), calculus (differentiation, integration, etc.) and series expansion (power series, rational approximations, continued fractions). A central part of many of these mathematical operations is simplification of expressions to canonical form. So for example both **y+x+x+a** and **x+y+a+x** are transformed to the canonical form **a+2x+y**. The derivative **D[f[x],x]** is transformed to the canonical form **D[f[#1],{#1,1,x}]** which displays explicitly the single differentiation with respect to a dummy variable **#1** followed by evaluation at **#1:x**. This canonical form makes possible immediate definition of symbolic derivatives. So for example **D[f[$x],{$x,1,$y}] : g[$y]** defines the derivative of **f** to be **g**. With this definition, the standard pattern matching process transforms **D[f[x^2]+f[3x],x]** to **2x g[x] + 3g[x]**.

To allow for general mathematical operations, SMP is able to perform not only standard mathematical operations, but also purely structural operations on symbolic expressions. Expressions can be treated either as symbolic tree structures, or can be manipulated by essentially graphical means. SMP includes many list manipulation primitives, covering for example the capabilities of APL. For example, **Ar**[*list specification,element specification*] creates lists of particular structure and with particular elements. Thus **Ar[4,f]** yields the four-element list **{f[1],f[2],f[3],f[4]}**, while **Ar[{2,2},g]** yields the 2×2 matrix **{{g[1,1],g[1,2]},{g[2,1],g[2,2]}}**. SMP includes a wide range of matrix manipulation functions. **Ar** can also be used to create higher rank tensors. So for example a general definition for the *n*-dimensional Levi-Civita (totally antisymmetric) tensor can be given simply as **Levi[$n_=Natp[$n]] : Ar[Ar[$n,$n],Sig]**, where **$n_=Natp[$n]** represents only expressions that satisfy the natural number predicate, and **Sig** signifies the signature function.

The definition of **Levi** just given is an example of a program in SMP. SMP has a wide range of programming constructs: its symbolic nature makes it substantially richer than numerical languages. SMP programs are usually built up interactively, with individual functions tested before being combined with others. This is made possible by the fact that any intermediate data corresponds to a legal SMP expression, which can be input or output like any other. And the powerful primitive functions of SMP are arranged to have expressions input and output in very standard forms, making them easy to string together into programs. Typical SMP programs make extensive use of pattern matching. Rather than having extensive conditional statements, they consist of a sequence of definitions to be used when they apply - a form much closer to that found in mathematical handbooks.

The input and output syntax of SMP can be defined by the user so as to be as close as possible to conventional notation. Two- and three-dimensional graphical output can also be obtained. The graphs are stored internally as symbolic expressions, and so can be manipulated for example by the same structural operations as are used for standard mathematical expressions.

Another capability of SMP, not directly linked to symbolic manipulation, but extremely significant in practice, is the ability to convert functions defined in SMP into C or FORTRAN programs which can then in turn be loaded into an SMP job. This capability makes it possible to write symbolic programs which can create programs needed for efficient numerical computation.

The kernel of SMP contains a core of basic mathematical knowledge. But the ultimate power of SMP lies in the ability to create programs for a wide range of mathematical problems. There is an expanding library of external files which contain definitions for particular applications. These files are indexed with a database system which uses keywords based on standard English language phrases. There are already several hundred external files, which serve for example to define new mathematical functions, specify relations and simplification rules for functions, catalogue mathematical, physical and other data, and to implement a wide variety of mathematical definitions and algorithms. With time, one can expect that much of the knowledge now found in mathematical handbooks can be put in this form, and accessed through SMP.

Experimental mathematics [2]

An increasingly important application of computers is in doing experiments on mathematical systems. Using a computer, one can find out how a mathematical system behaves, even though with conventional mathematical techniques one cannot carry out a complete analysis. (Nevertheless, having found evidence that a certain mathematical fact is true, it may be possible to prove it using conventional mathematical techniques.)

Probably the most significant consequence of the experimental mathematics method is that it is making possible the investigation of many complex systems hitherto inaccessible to theoretical study. It is central to such fields as dynamical systems theory.

As one example of experimental mathematics, I discuss a class of systems called cellular automata that I have studied extensively. These are mathematical systems whose microscopic construction is very simple, yet whose overall behaviour can be highly complex. They probably capture the essential mathematical mechanisms by which complexity is generated in many natural systems.

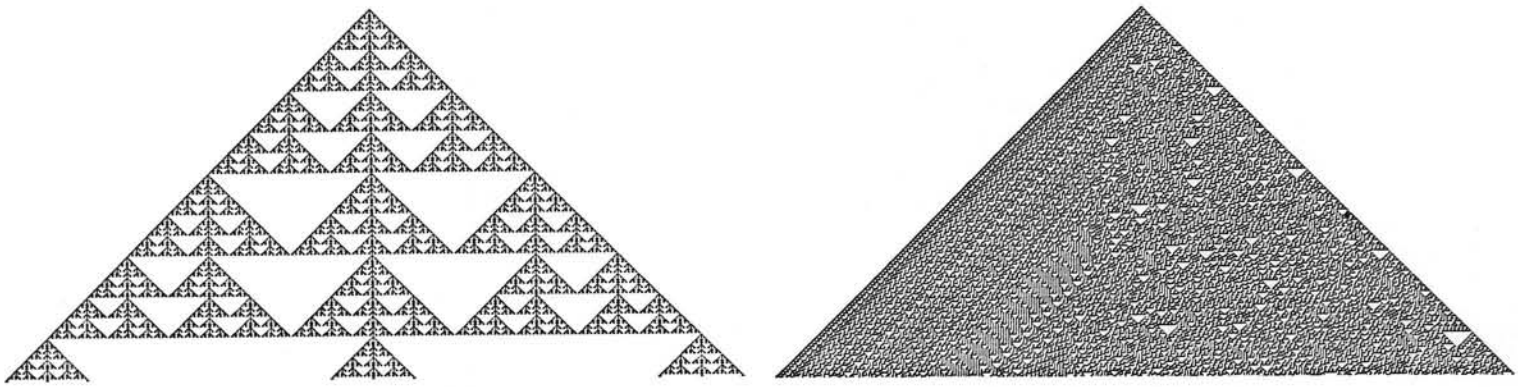

Figure 3. Patterns generated by some simple cellular automata (with two possible values for each site, and nearest-neighbour rules).

In the simplest case, a cellular automaton consists of a line of sites, each with say two possible values. The values are updated in a sequence of discrete time steps according to a fixed rule that depends on neighbouring values. Figure 3 shows some examples of patterns produced in this way. Considerable complexity is evident. But a number of definite features, such as self similarity, are seen. Almost all of these features were discovered using experimental mathematics, as a result of explicit computer simulations.

Cellular automata provide models for many physical, biological and other systems. They can be considered for example as approximations to partial differential equations. Not only is a discrete lattice taken in space and time, as in standard numerical analysis, but in addition discrete values are assumed for the variables at each site. Just one or two bits of information are included at each site, rather than the 32 or 64 bit numbers commonly used in the standard numerical analysis approach on digital computers. Local averages must be performed to find continuum quantities such as fluid density. The cellular automaton approach to a problem like fluid flow can be viewed as intermediate between molecular dynamics, in which very many molecules must always be averaged over to obtain a result, and the continuum partial differential equation approach, which gives directly specific results for continuous parameters.

An important advantage of cellular automaton models is that they can be implemented very efficiently by the forthcoming generation of massively parallel computers. The architecture of such computers is close to the architecture of cellular automata, and apparently also to the "architecture" of many physical systems.

Computation theory perspectives [2]

Computers not only provide practical tools for mathematical science, but also suggest new conceptual approaches that can be taken. Cellular automata and other models for physical systems can themselves be considered as computational systems, which process information given as initial conditions to yield final state output. One can use ideas from the mathematical theory of computation to develop principles for the overall behaviour of complex physical systems.

One question which can be addressed by computation theory methods is the fundamental basis for experimental mathematics. The outcome of the evolution of a physical system can always be deduced by experimental observation, or by explicit simulation, as in the experimental mathematics method. But there remains the question of whether it is, in principle, always possible to find a short-cut predictive procedure that obviates the need for explicit simulation.

All predictions must be made by some form of computer. But the system itself can be considered as a computer. Effective prediction is only possible if the predicting device can outrun the system itself, essentially by performing a more sophisticated computation. However, there is increasing evidence that many physical systems can act as universal computers, as powerful in their computational capabilities as any computer built from physical components can be. In such cases, the behaviour of the system can indeed in general be found essentially only by explicit simulation: the evolution of the system is "computationally irreducible".

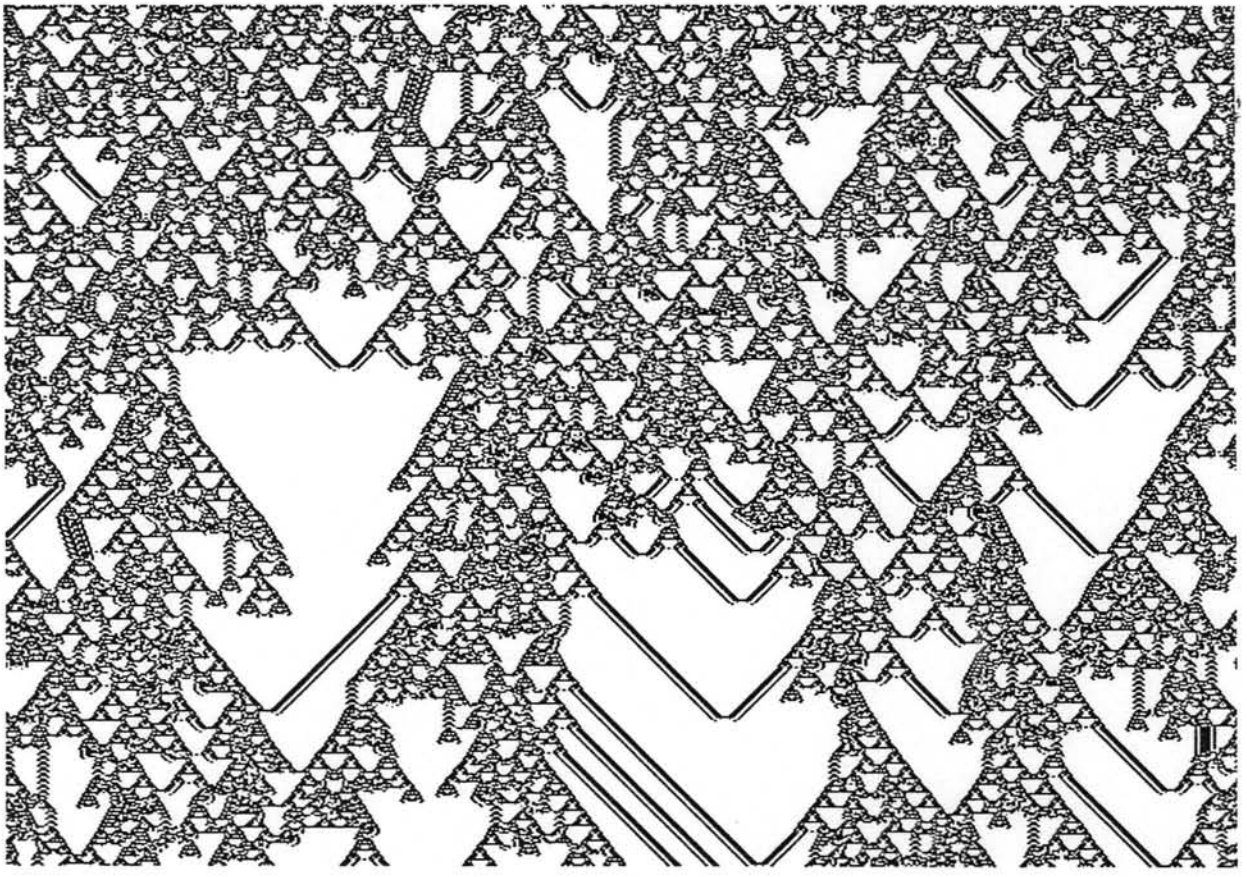

Figure 4. Evolution of a cellular automaton that is thought to be computationally irreducible, so that its outcome can essentially be found only by explicit simulation.

There are in fact many systems, even with rather simple construction, which seem to be computationally irreducible. Cellular automata provide many examples, one of which is shown in figure 4. Conventional theoretical physics has tended to concentrate on computationally reducible systems (such as

the two-body problem) for which exact mathematical formulae can be derived. But one suspects that the vast majority of complex systems are computationally irreducible. And for these systems explicit simulation following the experimental mathematics method is not only convenient, but actually necessary as a matter of principle.

As one further example of the application of computation theory to physics, I mention a problem with which I have recently been concerned: What are the basic mathematical mechanisms for the apparent randomness in such phenomena as turbulent fluid flow? One theory is that random behaviour results from amplification of external noise introduced through boundary conditions or thermal fluctuations. Another related possibility, extensively investigated in dynamical systems theory, is that the flow is sensitively dependent on in its initial conditions. Most arbitrarily-chosen initial conditions, it is argued, involve real numbers whose digit sequences are random, and are progressively "excavated" by the evolution of the system. However, as examples of cellular automata (such as those of figure 3) clearly show, very complicated and seemingly random behaviour can be generated even with simple initial conditions. This phenomenon is analogous to the process of pseudorandom number generation, in which a formula applied to a simple seed produces a sequence of apparently random digits. It is indeed common for a comparatively simple mathematical process to yield a sequence that seems random: the digits of π or $\sqrt{2}$ are examples.

In general one suspects that the evolution of many complex physical systems corresponds to a sufficiently sophisticated computation that it "encrypts" the initial conditions to produce output indistinguishable from random, at least in practical physics experiments. The phenomenon of turbulence could then be understood in computation theory terms.

Notes

1. See S. Wolfram, "Symbolic mathematical computation", Commun. ACM 28 (1985) 390. For further information on SMP, contact Computer Mathematics Group, Inference Corporation, 5300 W. Century Blvd., Los Angeles 90045 (213-417-7997).

2. See S. Wolfram, "Computer software in science and mathematics", Sci. Amer. 251 (September 1984) 188. For information on cellular automata, see S. Wolfram, "Cellular automata as models of complexity", Nature 311 (1984) 419. For more technical discussion of computation theory applied to physics, see S. Wolfram, "Undecidability and intractability in theoretical physics", Phys. Rev. Lett. 54 (1985) 735; S. Wolfram, "Origins of randomness in physical systems", Phys. Rev. Lett., to be published.