To appear in *Encyclopedia of Applied Physics* (1997) BU-CCS-970502

Very Large-Scale Simulation of Physical Systems

Bruce M. Boghosian
Center for Computational Science,
Boston University 3 Cummington Street, Boston, Massachusetts 02215, U.S.A.
bruceb@bu.edu

June 23, 1997

Abstract

This article surveys the relationship between computation and physics. It describes the past, present and future of how numerical data for scientific computation is represented, manipulated and extracted from physical computing devices. Aspects of the theory of computation, computational complexity, computational physics and quantum computation are discussed, towards the end of understanding how computers are and will be used to gain insight into some of the most data-intensive and time-consuming problems in theoretical and applied physics.

Contents

Introduction

The purpose of this article is to survey the history and the state of the art of the relationship between computation and physics. Traditionally, this topic has been divided into three major areas of study: *Physical computation* is the study of the use of physical systems, from simple mechanical devices to modern very-large scale integrated circuitry, to perform computation. *The theory of computation* is the mathematical specification of exactly what we mean by "computation," and of what computers can do and what they cannot do. *Computational physics* is then the study of how to use computers to simulate physical systems.

Recent developments in this field make it clear that all of these areas of study are closely related, and that their relationship will only strengthen in the coming decades. The smallest electronic circuitry is already at the point where quantum mechanical effects play a significant role. The consequences of the fact that certain physical systems are capable of computation have had a subtle but profound impact on physics, from the study of dynamical systems to the Second Law of Thermodynamics. Finally, the recent realization that *quantum computers* may be able to do things that classical computers cannot (or at least do them much faster) reminds us that paradigms of computing are not unique mathematical abstractions, but are rather dependent upon an underlying physical model. The observation that *computation is physical* is very analogous to – and every bit as profound as – the nineteenth century realization that Euclidean geometry is not a unique mathematical abstraction and the early twentieth century addendum that the geometry of our physical world is in fact noneuclidean.

This is a vast field, and so this article is necessarily in the nature of a very cursory survey. The object has been to give the reader many different and varied perspectives on how physicists use and think about computers, rather than a complete account of any one of them. Nevertheless, it is hoped that it will give the interested reader some appreciation of how and why people use physical systems to compute, of the nature of the computational process, of the implications of the theory of computation to physics, of the use of computation to study physics, and of the future of this entire enterprise.

1 Computation and its Relationship to Physics

1.1 The History of Computation

Since the first shepherds tied knots in strings to keep track of their flocks, mankind has employed physical systems to tabulate numerical data and perform computations. The reason then was the same as it is now: Physical devices can be used quite effectively to reduce the tedium of storing, manipulating and retrieving large amounts of data. Other ancient calculational procedures may have involved the use of pebbles (indeed, the word "calculate" is derived from the Latin word for pebble), or marks on papyrus or clay to keep track of data. It is likely that early counters recognized that they could deal with data far more efficiently by employing representions for groups of objects, and then groups of groups of objects, etc. This observation gave rise to the *digital* representation of numbers [Schwartzman 1996], and to computing machinery that stored and manipulated numeric data in this format. The *abacus* was developed in the Middle East at least 2500 years ago and uses wooden beads on a wire to add and subtract numbers in a digital representation.

Mechanical aids for multiplication and division are a much more recent development. In early seventeenth century Scotland, John Napier carved logarithmically spaced divisions on ivory sticks called *Napier's Bones*. These could be used for multiplication and division, and soon gave rise to the *slide rule*, which had both linearly and logarithmically spaced divisions and could be used to add, subtract, multiply and divide. Slide rules remained in widespread use until the 1970's.

Meanwhile, the automation of tedious and difficult mechanical tasks in the Industrial Revolution gave rise to the idea that tedious and difficult calculations could likewise be performed by "computational engines." Thus, in Napoleonic France, Joseph Marie Jacquard developed a loom with an automatic card reader, so that information about desired patterns of colors on fabric could be easily encoded on punched cards. In Cambridge, England in the mid-1800's, the mathematician Charles Babbage worked on the construction of a *Difference Engine*, and later an *Analytical Engine*; these would accept input from Jacquard-style punched cards and, by means of a complicated system of gears and cogs, compute and automatically print tables of numerical data. Though Babbage never finished these projects, he can be credited with designing the first general-purpose computer with data storage and printed output. Moreover, his collaboration with the mathematician Ada Augusta Lovelace gave rise to the first *computer programs*, including the *loop* construct, *subprograms*, and *conditional jumping*. By 1890, Herman Hollerith had convinced the United States Census Bureau to store information on punched cards, so that they could be processed by an electrical device.

The twentieth century has seen spectacular advances – both quantitative and qualitative – in the field of computing. Between 1937 and 1944, Howard Aiken of Harvard University built a gigantic electrical computer called the Mark I, which could read and write punched cards and manipulate data by means of electrical switches and relays. It could add and subtract numbers in about 0.3 seconds, multiply them in less than 6 seconds, and divide them in less than 16 seconds. Meanwhile, at the University of Pennsylvania, Drs. John Mauchly and J. Presper Eckert developed the first electronic computer, called ENIAC, which used 18,800 vacuum tubes to perform computations. This represented a great advance in speed, as it could perform hundreds of multiplications and/or additions in one second; its memory, however, was limited to twenty ten-digit numbers. By 1949, the EDSAC (Electronic Delay Storage Automatic Calculator), built at Cambridge University in England, could perform both addition and multiplication in times on the order of a millisecond.

Thus was begun the exponential growth of computational speed and efficiency that, fueled by the microelectronics revolution of the late twentieth century, continues to the present day. Currently, several figures of merit of modern computers seem to maintain an exponential rate of growth: At the time of this writing, modern processors can add and/or multiply hundreds of millions of *floating-point numbers* per second. (This figure of merit is sometimes called "FLOPS.") Clock rates are on the order of a nanosecond, and processor speed seems to double every 1.5 years or so. Fig. 1 shows the amount of physical area required to store one *bit* of information as a function of time, beginning with relays as they were used in the Mark I and ending with modern integrated circuitry; the exponential rate of decrease of this quantity is apparent.

This stunning increase in computing capability has already had a profound impact on physics. First, and most clearly, it has elevated computation to a status on a par with that of theory and experimentation as a means of acquiring scientific knowledge. For example, though analytic methods were often able to discern the overall behavior of the *linear* equations of classical mathematical physics, such as the Maxwell equations, computation has made it possible to solve such equations in a much wider variety of geometries, with inhomogeneities, and irregular boundary and/or initial conditions. Moreover, certain *nonlinear* equations of mathematical physics, such as the Navier-Stokes equations of viscous fluid dynamics, have been known for more than a century, but have only a very restricted set of known, nontrivial analytic solutions; the overall behavior of these equations is safely said to be unknown, as even the most elementary statistical properties of high-Reynolds number Navier-Stokes flow are analytically intractable. Computation has thus made it possible to examine the behavior of a much wider class of solutions to these equations, and compare these to experiment, thereby enhancing our ability to test the laws of physics. Much of this article will be devoted to outlining how computers are used for this purpose.

1.2 The Theory of Computation

Computers can simulate physical systems, but, at the same time, they are themselves physical systems. The computer simulation of physics can thus be viewed as the representation of one physical system by another. This observation has profound and surprising consequences. To begin to appreciate some of these, let us return to Fig. 1. In the physical world, no exponential rate of increase can go on forever, and the figure points to serious problems that will be faced by the computer industry in the coming decades. At the time of this writing, the physical area required to store one bit of information is seen to be on the order of the size of a biological cell. If the exponential rate of decrease were to continue unchecked, this size would be down to atomic scales before the year 2020. Such considerations have fueled new thinking about the relationship of computation to physics, and of possible physical limits to computation. The results of such thinking may ultimately be viewed as the second great contribution of the twentieth century to the science of computation.

Though the underpinnings of the theory of computation date back to the nineteenth century, it was the mathematicians Alan Turing and John von Neumann who, in the 1930's and 1940's, developed the modern theory of computation that guided the first fifty years of electronic computer evolution. Turing provided a mathematical model of the computational process, known as a *Turing machine*. Conceptually, this can be imagined as a linear strip of paper tape, infinite in both directions and imprinted with symbols chosen from some finite set, passing through a machine with some finite number of internal states. At each time step, the machine can read a symbol from the paper tape and, based on that symbol and the internal state of the machine at that moment, take one or more of four actions: (i) change the internal state, (ii) print something on the tape, (iii) advance the tape in one direction or the other, and (iv) halt the machine. Note that the Turing machine can be thought of as a dynamical system with Markovian evolution. In spite of its conceptual simplicity, Turing showed that his "machine" was capable of performing all of the arithmetic operations of number theory, such as addition, subtraction, multiplication, division and exponentiation, by initializing the paper tape with the proper (finite) "program." It is also possible to make the Turing machine perform more complex algorithmic procedures, such as looping and conditionally jumping.

More interestingly, it is even possible to make any Turing machine simulate any other Turing machine. That is, it is possible to write a program for a Turing machine A which accepts as input *both* the rules and input of another Turing machine B, and produces the same output that B would have produced. Because the Turing machine has this ability, we say that it is capable of *universal computation*. It therefore seemed eminently reasonable to regard the universal Turing machine as a complete mathematical paradigm of exactly what a computer is and does.

To be sure, the Turing machine is not the only possible computational paradigm that can be demonstrated capable of universal computation. One might imagine other schemes, with more tapes, more symbols, more internal state, etc. A particularly interesting computational paradigm, advanced by von Neumann just after World War II is called a *cellular automaton*. This consists of an infinite Cartesian grid with some (usually small) number of states at each site, which are updated according to a rule that depends on the states of neighboring sites. As we shall demonstrate below, this paradigm is also computationally universal, in that it is capable of simulating a Turing machine; thus, we say that it is *Turing equivalent*. Of course, this raises the question of whether some other digital computational paradigm might be found to be *super-Turing* – that is, capable of performing a computation that a Turing machine cannot perform [Analog]. Turing's name, along with that of the American logician Alonzo Church, is attached to the hypothesis that the answer to this question is "no." The *Church-Turing Hypothesis* states that any computer capable of universal computation may simulate any other computer capable of universal computation. That is, outside of quantitative considerations such as memory and speed, no universal computer is more powerful than another. There is no problem that one can do that another cannot also do, given sufficient time and memory. Thus, computer manufacturers may argue about whose product is faster and larger, but none can claim to perform any computation that their competitors cannot.

In addition to establishing the concept of universal computers and outlining what they could do, Turing addressed the even more interesting question of what they could *not* do. Strongly influenced by the contemporary work of Kurt Gödel who showed that there existed truths of number theory that could not be proven from its axioms, Turing recognized the close analogy between computation and the axiomatic systems of mathematics, and demonstrated that there were some functions that could never be calculated by a universal computer. To see this, imagine that we would like to construct a Turing machine to accept a number x as input, compute a function $f(x)$, write it as output, and then come to a halt; such a procedure will be called an *algorithm*. Since such programs must consist of a finite number of symbols chosen from a finite set, they can be enumerated; that is, they have the cardinality of the integers. It is therefore possible to create an (infinite) "alphabetized" list of *all* such algorithms. (We agree to discard from this list algorithms that are syntactically incorrect.) The question then reduces to whether or not every function that can be specified has an algorithm appearing on the list. Let us denote by $f_i(x)$ the result of applying the ith algorithm on the list to the numerical input value x; that is, f_i is the function corresponding to the ith algorithm on the list. From this, we can construct the function $g(x) \equiv f_x(x) + 1$. We can now prove that the function $g(x)$ cannot be on the list. If it were, there would be some j for which $f_j(x) = g(x)$ for all x; but this cannot be the case because when $x = j$ we would have $f_j(j) = f_j(j) + 1$, a contradiction. Thus, we have found a function that cannot be computed by any algorithm. This result is all the more odd because it *seems* straightforward to describe an algorithm for computing the function $g(x)$: Simply write a program which, given an input value x, causes the Turing machine to enumerate legal algorithms until it finds the xth such algorithm, apply this algorithm to the number x, output the result and halt. Turing noted that the problem with this putative procedure is that it assumes that there is an algorithm for determining whether or not a given string of symbols is indeed a legitimate algorithm – that is, one that eventually produces output and halts. The conclusion is that no such "algorithm-verifying algorithm" can possibly exist.

The similarity of this line of reasoning to Gödel's Theorem for axiomatic systems in mathematics should be evident. To see this, note that we can reduce any question about number theory to the computation of a mathematical function. For example, by knowing the value of

$$\min_{\substack{n>2 \\ i,j,k>0}} \quad (i^n + j^n - k^n)^2 \tag{1}$$

we can discern the truth or falsity of Fermat's Last Theorem (FLT). Why is it not always possible to specify an algorithm for the computation of such a function? After all, to pursue the above example, it seems quite straightforward to write a computer program that will perform an open-ended search for a counterexample of Fermat's Last Theorem. One could simply loop over all possible i, j, k and $n > 2$, evaluating the above expression, and stopping if it were ever found to be zero. Clearly we can restrict the search to $i \geq j$ without loss of generality, so j can be taken from the finite set $\{1, \ldots, i\}$; likewise it is easy to see that $k \in \{i+1, \ldots, 2i\}$. This leaves infinite loops over two integer variables, i and n; these can be done by conducting a single open-ended loop over their sum $m \equiv i + n \geq 4$, restricting i to the finite set $\{1, \ldots, m-3\}$, and taking $n = m - i$. (Of course, much better bounds are available, but these serve to illustrate the idea.) The problem with this grand scheme is that there is no *a priori* way of knowing if this program will ever halt (unless, of course, FLT is proven, as may be the case at the time of this writing). If it does not halt, then it is not a valid algorithm for computing the function in Eq. (1), since it does not provide a result in a finite number of steps. Thus, our result on uncomputable functions is related to the impossibility of solving what is often called the *Halting Problem* – this is the problem of making an *a priori* determination of whether or not a given computer program will ever halt.

The impossibility of solving the Halting Problem has profound implications for the simulation of dynamical systems on a computer, and may indeed have profound implications for physics itself. To see this, let us return to von Neumann's idea of computing by means of *cellular automata.* One very heavily studied cellular automaton, developed by the mathematician John Horton Conway, is called the *Game of Life.* It is a dynamical system on a two dimensional Cartesian grid, with only one bit of information (0 or 1) at every site, that is synchronously updated in discrete time steps according to rules that are easy to describe: Consider a site whose bit is b, and let the sum of the bits of that site's eight neighbors (at distances 1 and $\sqrt{2}$) be called S. The site then "updates" its bit to the value b' where

$$b' = \begin{cases} 1 & \text{if } (b, S) \in \{(1,2), (0,3), (1,3)\} \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

It is to be emphasized that this rule is to be applied *simultaneously* to all of the sites on the lattice.

If this rule is applied to a lattice with randomized initial conditions, a wide variety of interesting emergent behavior is observed. Fig. 2a shows configurations that are locally stable; that is, they are invariant under the action of the rule. Fig. 2b shows configurations that oscillate in time, returning to their initial configuration after a definite period of time. In the language of dynamical systems theory, we might say that these kinds of patterns are *fixed points* and *limit cycles*, respectively. It is also possible to find patterns that are reminiscent of solitons or nonlinear waves; Fig. 2c shows examples of patterns that return to their original shape – but translated in space – after some finite period. The first of these translating patterns is often called a "glider." Fig. 2d then shows a "glider gun" – that is, a pattern that oscillates (with period 24), emitting one glider at every oscillation.

Gliders are somewhat reminiscent of "bits" of information travelling along a wire in a computer. In this analogy, the glider gun plays the role of a "clock," emitting these bits at a set rate. It is also possible to show by construction that one can create the analog of *logic gates* for gliders – for example, a "NOT" gate which emits gliders at the clock frequency unless another glider is incident, and an "AND" gate which emits one glider if and only if gliders are present on both of two incoming channels. Since these logic gates are the basic building blocks of a digital computer, one could, in principle, use the patterns formed by the Game of Life to set up a universal computer on the lattice. This exercise may seem academic – indeed, almost frivolous – but it contains a very important lesson: It shows that the Game of Life is *Turing-equivalent* – that is, its dynamics are capable of universal computation. It is highly significant that a simple classical dynamical system, defined by nearest-neighbor rules on a lattice, can function as a universal computer. Among other things, it means that it is *provably impossible* to know the asymptotic behavior of the Game of Life for every initial condition, since this would be tantamount to solving the Halting Problem. Note that there are many physical models, such as the heat equation, the asymptotic behavior of whose solutions can be determined by elementary methods of analysis. We have just demonstrated that the Game of Life is one dynamical system that *cannot* possibly admit such a simple, general asymptotic treatment.

Now there are many other dynamical systems in physics – such as the driven pendulum – that have thus far resisted a general asymptotic treatment. Could this be because they somehow set up structures that are capable of computation? A partial answer to this question is provided by the work of Moore [Moore 1990, Moore 1991] who showed that bundles of trajectories in simple dynamical systems, as can be identified on a Poincaré surface of section ("puncture plot" in phase space), can shuffle about in successive time steps in a manner that is Turing-equivalent. Moore demonstrated this by construction for a system consisting of an ideal classical particle in three dimensions, with a finite number of planar and parabolic reflectors. The asymptotic state of such a system is therefore unknowable for general initial conditions. Note that this *uncomputability* is a much stronger result than the demonstration of "chaos" for such a system. Certain dynamical systems, such as the logistic map with maximum multiplicative parameter ($x_{n+1} = 4x_n(1 - x_n)$ where $x_n \in \{0, 1\}$), are certainly chaotic, in the sense that they have positive Liapunov exponent, but at the same time admit a simple asymptotic treatment. (The substitution $x_n = \sin(2\pi\theta_n)$ reduces this map to $\theta_{n+1} = (2\theta_n)_{\text{mod}1}$, which has the effect of chopping off the leading digit of the binary representation of the number at each time step. Thus, given the initial condition to infinite precision, one can immediately specify the result after iteration n.) Much less can be said about the time asymptotics of an *uncomputable* dynamical system.

1.3 Physical Computation

These mathematical limitations to computation are undeniably fascinating, but one might well wonder if they are simply academic exercises. After all, the proven existence of mathematical functions that cannot be described algorithmically certainly did not stop people from making good use of computers in the simulation of physical systems. At the same time, however, it should be kept in mind that real-world computers *are* themselves physical systems. Thus, as the power of real computers increased dramatically in the latter half of the twentieth century, it was natural for people to begin to wonder if there were any *physical* limitation to their size and speed, in addition to the mathematical limitations studied by Turing. As we shall see, this issue remains an important topic of study to this day, and in addressing it we shall find that many of the issues raised by Turing reappear in a slightly different form.

Turing machines are mathematical idealizations of computers. How fast and how small may we make a computer in the real world? In the drive to make computers more powerful, one of the fundamental engineering constraints is due to the heat released by its components. As these components are packed ever more tightly together in three dimensions, the dissipation of this heat becomes an important problem. This leads to the question: Must computation always release heat? As late as the 1960's, it was widely believed that the answer to this question must be "yes." The reasoning was predicated on an ingenious argument due to Landauer [Landauer 1961] that erasure of information in a computer – a process that goes on all the time in modern computers – must be accompanied by the dissipation of heat. This argument went as follows: Consider a microscopically reversible physical system that is capable of representing N macroscopically observable bits of information. Since each bit can take on one of two values, the system of N bits can be in any one of 2^N macroscopic states. Corresponding to each of these macroscopic states, let's say that there are Ω_0 microscopic states of the underlying physical structure, so that the full state-space volume can be written as $\Omega = 2^N\Omega_0$. Note that the volume associated with the *computational degrees of freedom* is 2^N, while that associated with the *noncomputational degrees of freedom* is Ω_0, and that these volumes are multiplicative since they correspond to independently specifiable degrees of freedom. Since the entropy of a system is Boltzmann's constant times the natural logarithm of its state-space volume, the entropy associated with our physical system is then $S = k_B \ln \Omega = k_B \ln(2^N\Omega_0) = k_B \ln \Omega_0 + N k_B \ln 2$. Since we have assumed that the underlying physical system is reversible, this entropy must be constant. Now suppose that our physical system "erases" a bit of information at the macroscopic level; that is, suppose that it forces one of its bits to take on a specific value, regardless of its previous value. This constraint has the effect of contracting the state-space volume of the computational degrees of freedom by a factor of 2, since it can now be in one of only 2^{N-1} states; equivalently, we see that this has the effect of decreasing the entropy of the computational degrees of freedom by $\Delta S = k_B \ln 2$. Since the entropy of the system as a whole must be constant, we see that this lost computational entropy must be dumped into the noncomputational degrees of freedom. If the underlying physical structure corresponding to the noncomputational degrees of freedom is maintained at a temperature T, the increase in its entropy by ΔS corresponds to the appearance of an amount $T\Delta S = k_B T \ln 2$ of dissipated heat energy. Moreover, the full force of the Second Law of Thermodynamics precludes the possibility of this heat energy ever being recovered as useful macroscopic information.

Landauer's argument indicates a close connection between the process of computation and thermodynamics. In particular, it indicates that any computer that erases bits of information must necessarily release heat. The trouble is that all of the familiar logic gates from which modern computers are built do this all the time. "AND" and "OR" gates, for example, are irreversible because their input can not be deduced from their output. This is illustrated in Fig. 3a. Since one bit of information is lost each time either of these gates is used, heat energy $T\Delta S = k_B T \ln 2$ must be dissipated. While this amount is many orders of magnitude smaller than the amount of heat released by the operation of a real computer gate, the result was considered interesting since it seemed to indicate that computation was fundamentally and necessarily irreversible.

In passing we note that this observation was used by Bennett [Bennett 1987] and others to finally and convincingly explain the "Maxwell's demon" paradox that had plagued kinetic theory for a century. The issue has to do with the apparent ability of a "demon," poised at a small opening in a partition between chambers of high- and low-pressure gas, to violate the Second Law of Thermodynamics by allowing molecules from the low-pressure side to enter the high-pressure side, but not vice versa. Such a "demon" effectively exploits the thermal energy of the molecules to create a pressure difference which could then be employed to do useful work – an egregious violation of the Second Law. A number of explanations of the demon have been put forth over the past century, but the issue was finally resolved only in the 1980's. Bennett observed that such a demon must perform what is essentially a computation in order to decide whether or not to allow a molecule through the partition. Indeed, Bennett was able to demonstrate this most clearly using a variant of the Maxwell demon argument due to Szilard. Requiring that the demon be able to perform computation is not, in and of itself, a fatal flaw, but the fact that the demon must *erase* his computational memory after each molecule is processed, in preparation for the next one, is the real explanation of why he is unable to violate the Second Law.

Returning to the lost information and concomitant heat generation inherent in computation, we note that one might try to retain the lost bits of information by considering two-input/two-output gates. Even then, however, one must restrict one's attention to those gates for which the input can be reconstructed from the output. It is not difficult to see that there are $2^8 = 256$ gates that accept two bits of input and yield two bits of output. Of these, $4! = 24$ are reversible, in that the two-bit output states are a permutation of the two-bit input states. Unfortunately, these gates are insufficient to construct a universal computer. It should be noted that, in addition to the usual gates necessary for this purpose (e.g., "AND" and "NOT"), it is necessary to worry about such things as the "FANOUT" function that is implicit every time one wire splits into two within a circuit, as in Fig. 3b; though "FANOUT" is reversible given a legal output state, not all possible output states are legal and the operation does seem to change the number of bits and hence the entropy of the computational degrees of freedom.

The observation that two-bit reversible gates are insufficient to construct a universal computer, and the realization of the problematic nature of operations such as "FANOUT" long taken for granted by computer designers tended to support the argument that universal computation is necessarily irreversible. In the 1970's, however, it was realized [Bennett 1973, Fredkin 1982] that this argument is specious. Fig. 3c shows a three-input/three-output gate, known as the *Fredkin gate*, that is fully reversible and quite sufficient to construct a universal computer. It has three inputs (b_1, b_2 and b_3) and three outputs (b'_1, b'_2 and b'_3). If the first input is zero, then the other two outputs are exchanged; otherwise the outputs are the same as the inputs. The truth table for this gate is thus

b_1	b_2	b_3	b'_1	b'_2	b'_3
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	0
1	1	1	1	1	1

Note that if $b_3 = 0$ then $b'_2 = b_1 \wedge b_2$. (Here we identify 0 with *false* and 1 with *true*, as is customary.) This means that the gate captures all the functionality of the "AND" gate. If we fix $b_2 = 1$ and $b_3 = 0$, then $b'_3 = \sim b_1$ giving us the functionality of the "NOT" gate, and $b'_2 = b_1$ giving us "FANOUT." These functions are sufficient to create a universal computer.

One can therefore imagine a universal computer that accepts input bits on the left, passes them through circuitry consisting entirely of Fredkin gates, and yields output on the right. There is, however, one remaining problem: In addition to the desired output, numerous "garbage" bits will have accumulated in the course of the calculation. For example, as we have seen, it is possible to arrange for the Fredkin gate to output the "AND" of two of its input bits, but it also produces other output bits when it does so. That other output, inessential for our calculation, cannot be simply thrown away because that would generate heat and violate reversibility. It seems possible that we have just postponed the problem of heat generation to the stage at which we read the computer's output and reset it for a new computation. As soon as we read the portion of the output that we want and uncerimoniously erase the rest of it, heat will be generated in proportion to the number of garbage bits destroyed. It is therefore necessary to bound the number of "garbage" bits created.

Thanks to an ingenious argument of Bennett [Bennett 1982], this is easy to do, and necessitates only a factor of two or so more circuitry. Suppose that we set up a reversible circuit of Fredkin gates as above, with input on the left, and both output and garbage bits on the right. Since the Fredkin gate is its own inverse, it is possible to recover the input from the output using a mirror image of the very same reversible circuit. Adding this on the right, we recover the original input bits which can then be channelled back to the left-hand side of the circuit and reused. At this point the circuit is fully reversible, and all of the "garbage" bits have been recycled. Recall, however, that the desired output resides somewhere in the middle of all this circuitry. By extracting it with "FANOUT" operations, we arrive at a perfectly reversible circuit that produces no garbage bits.

This discovery was quickly followed by that of a reversible cellular automaton that is capable of universal computation [Margolus 1984], and by Fredkin's observation [Benioff 1982] that a system of ideal, billiard balls and reflectors could perform reversible logic. Fig. 4a shows the paths taken by two billiard balls with perpendicular trajectories in the event that both balls are present, and that either ball is absent. If the presence of a ball is regarded as a "bit" of information, examination of the final trajectories allows us to perform the "AND" operation on these bits. Given a steady supply of balls, emerging at the clock rate, the "NOT" operation is also easy to arrange: Simply collide the input balls with the stream balls in such a way that the former knock the latter (and themselves) out of the stream. The stream balls that do get through will then be the "NOT" of the incoming signal. Once again, "AND" and "NOT" are sufficient to attain universal computation.

One critique of billiard-ball computational paradigms is that they are inordinately error prone. The slightest deviation of any of the balls from their intended trajectories will tend to propagate and amplify in time according to the Liapunov exponent of the dynamics. It therefore seems that reversible computation could not be achieved experimentally. It turns out to be possible, however, to implement a simple and effective *error correction* scheme. Indeed, note that the "NOT" gate described above does precisely this. The output ball of the "NOT" gate is likely to be much more accurately aimed than the input, since the output balls have just emerged from their source. Thus, by passing a signal through two "NOT" gates, we can remove any errors that may have accumulated in their trajectories.

Thus, bits of information do not have to be destroyed in order to perform universal computation, and heat dissipation is not an unavoidable side effect of computation. Moreover, reversible computation can be achieved reliably, with error correction. The notion that computation can be reversible and robust is central to the idea of quantum computation, which shall be discussed later on in this article.

2 Languages for Scientific Computation

We now descend abruptly from the lofty heights of ideal computational models to the ignominious realities of performing useful computation on contemporary computers. As noted above, the realization that computers could not compute everything did not stop people from trying to compute as many things as possible. In order for computers to become a practical tool for scientific computation, however, it was necessary that they become easy to program. At the very lowest level, computers accept instructions in *machine language*, which is really nothing more than a sequence of elemental instructions in the binary format expected by the processor. In fact, the very first computer programs input this binary data by setting hardware switches on the computer's front panel. This approach is obviously inappropriate for all but the smallest sets of instructions.

At the next level, *assembly language* uses mnemonic abbreviations for certain of these elemental operations and groups thereof. In the late 1950's, researchers at IBM developed the first *high-level language*, designed to simplify the programming of a computer for solving scientific problems. A high-level language typically employs a program called a *compiler* (or perhaps an *interpreter*) to translate instructions written according to a carefully defined language syntax into assembly language. This first high-level language was called *Fortran* (an abbreviation of "FORmula TRANslation"), and it quickly came to be the predominant high-level language for scientific computation – a status which it retains to this day.

Since its invention, Fortran has had three principal stages of evolution: In the early 1960's many divergent varieties of the language existed. To combat this tendency, computer users and vendors cooperated on the development and publication of a language standard. The result, called Fortran 66, was the first standardized computer language (in that programs written in this language could be compiled and executed on different computer platforms) and this stability contributed to its popularity among scientific programmers. Though powerful, this language was very undisciplined. It allowed for control to jump about from one part of the code to another in arbitrary ways, by means of a `GOTO` statement, leading some programmers to faceitiously refer to it as "spaghetti code." Unless great care was taken in its use, it could be very difficult to read and understand an existing Fortran 66 code. For this reason, programs of more than 10,000 lines of instructions were very difficult to write, debug, maintain and modify.

The desire to impose some structure on the language led to the next major leap in computer programming, called *structured programming*, in the late 1960's and early 1970's. The idea was that control in a computer code should have a definite and more-or-less unidirectional flow to it; the `GOTO` statement should be avoided. Frequently reused groups of instructions should be broken up into succinct and easily invoked *subroutines* or *function subprograms*. These subprograms should have the ability to declare and use *local variables*, and they should be capable of calling themselves in a recursive fashion. A number of new languages, including Pascal, Algol, PL1 and C, were invented to support this new paradigm. Eventually, however, most of these ideas found their way into the next major Fortran language standard, called *Fortran 77*. This was adopted by the International Standards Organization in 1980, and remains the dominant dialect of the language in use to this day. Using Fortran 77 or C, programmers routinely write and maintain computer codes with tens of thousands of lines. As larger and more complex software was developed, however, it soon became clear that codes larger than about a hundred thousand lines would be very difficult to develop and maintain.

At the time of this writing, the newest and most fashionable structuring paradigm for computer software is called *object-oriented programming*. It is defined by the following principal characteristics:

- **Encapsulation, persistency of state, and data hiding:** The notion of subprograms should be strengthened to allow the creation of "black box" software with local data that is completely protected from outside interference, and may be persistent in that it does not disappear when the subprogram completes execution. This allows for the development and maintainance of software libraries in complete isolation from the application programs that utilize them. This should be contrasted with more conventional software libraries which often require internal state to be passed back and forth to the calling routine.

- **Data abstraction:** In structured programming languages, users could specify if a given variable represented, e.g., a logical value, an integer, a real number, or a complex number. In object-oriented programming, the user should have the ability to create his/her own data structures and representations out of more fundamental ones, and to declare variables in them. For example, one might imagine creating a data structure for quaternions, representing them by four real numbers; a user could then declare variables of quaternionic type.

- **Polymorphism:** Since object-oriented computer languages allow for many new and varied data types, one might well worry about the management of the plethora of new operations necessary to manipulate them. Of course, mathematicians use the same symbol "+" to denote addition of integers, reals, complex numbers, and quaternions. Likewise, object-oriented languages allow for *operator overloading* so that the same operation may be applied to variables of different type. The compiler is able to determine the type of the arguments and arrange for the appropriate operations to be performed when the code is executed.

- **Inheritance and extensibility:** Object-oriented programs should be modular, and allow for data to acquire or inherit certain properties of other data. For example, if a program has data structures for cats, mammals and animals, it ought to be possible for objects of type cat to inherit all properties of objects of type mammal, and for these to in turn inherit all properties of objects of type animal. This greatly facilitates hierarchical code development.

At the time of this writing, the most popular object-oriented computer languages are C++, an extension of the C language, and Java. Nevertheless, Fortran has incorporated many of these ideas in its latest standard, called *Fortran 90*. Other languages developed for scientific programming, such as Ada, allow for some, but not all, object-oriented features. Most of the software examples used in the remainder of this article will be presented in Fortran 90, using an object-oriented style.

3 Data Structures and Algorithms

3.1 Algorithms and Computational Complexity

The concept of a computer *algorithm* was described earlier, in reference to Turing machines. While modern algorithmic development has become a science unto itself, the basic idea remains much the same: An algorithm is a sequence of basic instructions, or "recipe," for solving a given problem on a computer. Often the same problem will allow for several different algorithms, and it becomes important to be able to compare them to see which is better. One way that this comparison can be achieved is to compare the algorithms' *computational complexity*. This is simply a count of the number of elemental operations required to solve the problem; it can be more mathematically defined as the number of steps required for a standardized Turing machine to implement the algorithm.

Consider the procedure known as *Gaussian elimination* for the solution for x of

$$Ax = b, \tag{3}$$

where A is a dense N-by-N matrix, and x and b are N-vectors. Assuming that nothing is singular, we begin by dividing each element of the first row by the first element in that row (a total of N divisions, excluding that of the first matrix element since we know that will yield one, but including that of the first element of b). We then multiply this row by the first element of each of the remaining rows, and subtract it from those rows, so that only the first row has a nonzero entry in the first column, and that entry is one (a total of $(N-1)(N-1)$ multiplications and subtractions for the nonzero matrix elements). To get to this point, we have performed a total of $2N^2 - 3N + 2$ operations (subtractions, multiplications or divisions). We can now repeat this process on the lower right-hand n-by-n submatrices, where n ranges from $N-1$ down to 1. At that point, we will have performed a total of

$$\sum_{n=1}^{N} (2n^2 - 3n + 2) = \frac{N}{6}(4N^2 - 3N + 5) \tag{4}$$

operations, and the problem will be solved. We say that the computational complexity of this algorithm is $\mathcal{O}(N^3)$; that is, if the number of unknowns is increased by a factor of z, the number of operations required to solve the problem will increase by z^3 in the limit of large problem size.

Oftentimes, clever algorithmic design can drastically reduce the computational complexity required to solve a given problem. For example, the *optimal wedding problem* is well known in combinatorics: Given N men and N women who state quantitative preferences for each other, say as a number from one to ten, how can they be paired so as to maximize the sum of the preferences? At first glance, it would seem that all possible pairings of men and women must be considered in turn, leading to an algorithm with complexity $\mathcal{O}(N!)$. In fact, an algorithm with $\mathcal{O}(N^3)$ complexity is known for solving this problem [Khuller 1991].

Perhaps more pertinent to scientific computation is the example of the discrete Fourier transform of N data elements, f_j where $j \in \{0, \ldots, N-1\}$. Since the definition of this transform,

$$\tilde{F}_k \equiv \sum_{j=0}^{N-1} e^{2\pi ijk/N} f_j, \tag{5}$$

involves a sum over N elements to obtain each of N results, it would appear that the transform requires a minimum of $O(N^2)$ elemental operations to compute. In 1965, however, Cooley and Tukey [Cooley 1965] showed that it is possible to develop an algorithm, called the *Fast Fourier Transform*, with computational complexity of order $\mathcal{O}(N \log N)$.

3.2 Parallel Computers

Since computers perform elemental operations at some rate fixed by an internal "clock," it follows that a single processor will perform a calculation in a time proportional to that calculation's computational complexity. A reduction of computational complexity therefore translates directly into faster execution time. Another way to decrease execution time is to apply more than one processor to a problem; this is called *parallel computing*.

Since the 1980's parallel computers, containing anywhere from a few to thousands of processors, have been extensively applied to problems of computational science. There are several reasons for this trend. First, as has been noted, *very large-scale integration* (VLSI) of electronic components is approaching some fundamental physical limitations of size and speed (see Fig. 1). By the mid 1980's, the fastest processors had clock rates measured in nanoseconds. It was therefore natural that people would begin to investigate increasing computational power through the cooperative use of many such processors as an alternative to making them faster. Second, as processors and memory components became faster, and as memories became much larger, people began to worry that the classic computer architecture, in which a single processor reads a few words of data from memory and then writes a result back to memory, was becoming a bottleneck to speed. Since only those words of memory actually accessed by the processor were in use at any one moment, most of memory remains idle. Such an architecture was first advocated by von Neumann (not to be confused with his advocacy of cellular automata as a computational paradigm), and is therefore called a *von Neumann architecture*. This problem is therefore sometimes called the *von Neumann bottleneck*. Third, as processors became cheaper in the 1980's, so that processor and memory prices (per unit area of silicon) became comparable, it made sense to try to combat the von Neumann bottleneck by spreading more processing power out to the memory, so to speak.

In spite of these rather compelling arguments, parallel computers took some time to find applications to computational science because of an initial fear that they would be difficult to program. When it turned out that the regular structure of many scientific programs was quite amenable to parallelization, and that most major parallel algorithmic paradigms could be easily captured in high-level languages, parallel computing exploded in popularity. At the time of this writing, virtually all major high-performance computer vendors offer one or more parallel computer platforms.

Early scenarios imagined *massively parallel* or *data parallel* computers, in which the number of processors available was comparable to the amount of data in a problem [Hillis 1993]. While computers with tens of thousands of processors have been built [Hillis 1985], the current market seems to favor a coarser, more modest, "granularity" in which anywhere from a few to a few hundred processors, with sophistication comparable to that of modern workstations, can communicate as if they are connected to a very fast *local-area network*. Nevertheless, if we imagine spreading an algorithm with N data elements over M processors, it is interesting to try to determine the asymptotic behavior of the time required to execute the algorithm as a function of N and M; of course, as $M \to 1$ this result must approach the computational complexity of the algorithm. To demonstrate this idea, we can consider a few simple examples.

Suppose that we have an *array* of N numerical data elements, D_i where $i = 1, \ldots, N$, and we wish to find its sum, $\sum_{i=1}^{N} D_i$. Clearly, this has computational complexity of order N, since $N-1$ additions are required. We could, however, distribute N/M data elements in each of M processors. These processors would then require a time of order N/M to sum their elements, but they could all do this simultaneously. Once they each had their subtotal, the M data elements would need to be combined into one total, but this could be carried out in the "tree" structure shown in Fig. 5, in which all the operations at each level of the tree are performed in parallel. For a binary tree, the combination requires only $\log_2 M$ parallel steps to complete, and the time required to execute the algorithm scales like

$$T(M, N) \sim c_1 \frac{N}{M} + c_2 \log_2 M + c_3, \tag{6}$$

where c_1, c_2 and c_3 are architecture-dependent constants. Specifically, c_1 is the time required to add two numbers in the same processor, c_2 is the time required to communicate a number from one processor to another, and c_3 is whatever constant "overhead" or *latency* time is required to orchestrate the calculation. In Fig. 6 we plot the *speedup* due to parallelism, defined as

$$S(M, N) \equiv \frac{T(1, N)}{T(M, N)}, \tag{7}$$

as a function of the number of processors M, for $N = 100$ and $c_2/c_1 = c_3/c_1 = 5$. Note that this initially increases with M, due to the parallelism, but quickly reaches a point of diminishing returns, as the latency and interprocessor communication begin to dominate the performance. It is not difficult to see that a maximum speedup of

$$S_{\max}(N) = \frac{c_1 N + c_3}{c_2 \log_2 \left(\frac{2 c_1 N}{c_2}\right) + c_3} \tag{8}$$

is obtained for M equal to

$$M_{\max} \equiv \frac{c_1 N}{c_2} \tag{9}$$

which is 20 for our example. For $M > M_{\max}$ the speedup actually begins to *decrease* with M! This rather pessimistic assessment of the benefits of parallelism is a special case of what is known as *Amdahl's Law*. This is the observation that, as the parallelism reduces the time required for the parallel portion of the computation towards zero, the latency (and interprocessor communication) will always end up dominating the speedup which therefore must have an upper bound.

Though Amdahl's Law seems compelling, in recent years researchers have very successfully applied parallel computers with thousands of processors to scientific problems. This seems like a contradiction, and has been the source of much confusion, but the reason is simple: Amdahl's Law considers the application of parallel computers to problems of a fixed size N. As problems become ever larger, however, note that both the maximum speedup, given by Eq. (8), and the number of processors for which it is obtained, given by Eq. (9), can become arbitrarily large. Thus, parallel computers are naturally most useful for the largest and most data-intensive problems of computational science. As we shall see, there is no shortage of such problems, so parallel computers are here to stay.

Before leaving the topic of parallelism and the scaling of execution time with the number of processors, we note that different parallel data structures can result in quite different time scalings. To illustrate this, we consider the problem of multiplying a pair of dense N-by-N matrices, A and B, to obtain their product C. Since

$$C_{i,j} = \sum_{k=1}^{N} A_{i,k} B_{k,j} \tag{10}$$

are N^2 quantities, each of which requires N multiplications and additions to compute, it is clear that the problem has $\mathcal{O}(N^3)$ computational complexity, and thus requires a time proportional to N^3 to complete on a single processor. By contrast, on a parallel computer of N^2 processors, this problem can be done in $\mathcal{O}(N)$

time. The algorithm used is the systolic or Cannon algorithm [Cannon 1969], and it is naturally parallel. Both operands, call them A and B, are stored as parallel variables. We begin by cyclically shifting the ith row of A by $i-1$ entries for all $i \in \{1, \ldots, N\}$, and the jth column of B by $j-1$ entries for all $j \in \{1, \ldots, N\}$. This puts $A_{i,j+i-1}$ and $B_{j+i-1,j}$ in array element (i,j) (where all subscripts are taken modulo N). We then multiply these and accumulate into the product parallel variable $C_{i,j}$. These N^2 multiplications and accumulations can be done in fixed time, since we have assumed that our parallel computer has $\mathcal{O}(N^2)$ processors. Continuing, we cyclically shift all the rows of A and all the columns of B by one, and then multiply and accumulate into C. We repeat this step a total of N times. After the kth such step, array element (i,j) contains $A_{i,k+j+i-1}$ and $B_{k+j+i-1,j}$, so the N multiplications and accumulations into array element (i,j) are equivalent to a sum over the dummy index. Thus, after N steps, the result for $C_{i,j}$ is correct and in place. The algorithm takes $\mathcal{O}(N)$ time because N steps are required. The Fortran 90 implementation of the Cannon algorithm is

```
SUBROUTINE CANNON(A, B, C)

  INTEGER :: i, n
  REAL, DIMENSION(:, :) :: A, B, C

  n = SIZE(A, 1)

  ! Initial skewing of operands:
  A = CSHIFT(A, DIM = 2, SHIFT = (/(i, i = 0, n-1)/))
  B = CSHIFT(B, DIM = 1, SHIFT = (/(i, i = 0, n-1)/))

  ! Multiplication of matrices:
  C = 0.0
  DO i = 1, n
     C = C + A*B
     A = CSHIFT(A, DIM = 2, SHIFT = 1)
     B = CSHIFT(B, DIM = 1, SHIFT = 1)
  END DO

  ! Restore operands:
  A = CSHIFT(A, DIM = 2, SHIFT = -(/(i, i = 0, n-1)/))
  B = CSHIFT(B, DIM = 1, SHIFT = -(/(i, i = 0, n-1)/))

END SUBROUTINE CANNON
```

Note that the `CSHIFT` command performs the cyclic shifts of either rows or columns; its first argument is the array to be shifted, and the other two keyword arguments are the dimension in which to shift it, and the amount of the shift. Also, we have added a step to restore the operands to their input values.

To take this one step further, suppose that we have a parallel computer with $\mathcal{O}(N^3)$ processors. We can then allocate arrays `D`, `E` and `F` of size $N \times N \times N$, and *spread* array `A` over the third dimension of array `D` so that $A_{i,k}$ is stored in $D_{i,k,j}$ for all j, and *spread* array `B` over the first dimension of array `E` so that $B_{k,j}$ is stored in $E_{i,k,j}$ for all i. We can then perform all N^3 multiplications of $D_{i,k,j} = A_{i,k}$ by $E_{i,k,j} = B_{k,j}$ in fixed time, since we have assumed a parallel computer with $\mathcal{O}(N^3)$ processors. The answer for array `C` is then obtained by summing array $F_{i,k,j} = D_{i,k,j}E_{i,k,j}$ over its second (dummy) dimension. The Fortran 90 implementation of this algorithm is as follows:

```
SUBROUTINE FAST_MATMUL(A, B, C)

  INTEGER :: n
  REAL, DIMENSION(:, :) :: A, B, C
  REAL, DIMENSION(:, :, :), ALLOCATABLE :: D, E
```

```
  n = SIZE(A, 1)
  ALLOCATE(D(n, n, n), E(n, n, n))

  D = SPREAD(SOURCE = A, DIM = 3, NCOPIES = n)
  E = SPREAD(SOURCE = B, DIM = 1, NCOPIES = n)
  C = SUM(D*E, DIM = 2)

  DEALLOCATE(D, E)

END SUBROUTINE FAST_MATMUL
```

Note that there was no need to actually declare the array `F`. Note also that the most expensive operations were the *spreads* of array elements over a new array dimension, and the summation or *reduction* of array elements over an existing array dimension. Both of these operations, however, can be carried out in time of $\mathcal{O}(\log_2 N)$, by the same arguments that we used to parallelize summation, illustrated in Fig. 5. Thus, on $\mathcal{O}(N^3)$ processors, the multiplication of two $N \times N$ matrices takes $\mathcal{O}(\log_2 N)$ time.

3.3 The Computer Representation of Real-Valued Data

Most physical models are formulated in terms of quantities that can take on a continuous range of values. Many physical models are described in terms of *fields* (either classical or quantum), for which the number of degrees of freedom itself is infinite. By contrast, digital computers can manipulate only a finite amount of finite-precision data. Thus, one of the principal challenges of computational physics is to reformulate physical laws in terms of finite and discrete mathematical models.

Real-valued quantities can be represented as finite-precision *floating-point numbers* in most all high-level languages available today. There is a standard for the storage of such numbers, adopted by the Institute of Electrical and Electronics Engineers (IEEE), whereby *single-precision* floating-point data is represented as

$$\sigma(1+m)2^e, \tag{11}$$

where the *mantissa* $m \in [0,1)$ is represented by 23 bits of data, the exponent $e \in \{-127,\ldots,128\}$ is represented by 8 bits of data, and the sign of the number $\sigma \in \{-1,+1\}$ is represented by one bit, for a total of 32 bits. *Double-precision* floating-point numbers are also available in which the mantissa is represented by 52 bits, the exponent by 11 bits, and the sign by one bit, for a total of 64 bits. Thus, both single- and double-precision floating-point data take their values in a finite range, and the computer user must be ever vigilant not to *overflow* or *underflow* this range. There are many subtleties involved in this truncation; for some nice examples, see [Kahan 1996].

Should we expect the truncation of real-valued data into finite-precision floating-point numbers to cause problems in physical simulations? Obviously, this finiteness circumvents the *computability* problem noticed by Turing and discussed above. Recall that Turing machines are idealized models of computers with infinite storage capacity. Real computers have finite storage capacity, and all finite strings of bits are computable, so that any number represented on a real computer is finite-precision and computable. Thus, the issue of computability would seem to be an esoteric problem, wrapped up in infinities, that plagues idealized models of computing but not real computers. This side-stepping of the issue is satisfactory so long as the real world can be described in terms of computable numbers alone. Interestingly, Pour-El and Richards [PourEl 1982] have proven that even simple linear physical models, such as the wave equation, when given computable initial conditions and evolved for a computable length of time, can give rise to uncomputable values of the dependent variable. Should this be a cause for concern? The real answer is that almost nobody worries about it. Nevertheless, we shall soon discuss some situations in which truncation of continuum data plays a very detrimental role in physical simulations, so this issue is important to at least keep in mind.

4 Grid-based algorithms

4.1 Finite-difference Methods

The most obvious way to discretize classical fields is to simply store their values on a fine spatial grid, and to update those values in discrete time steps. Historically, this was the first approach adopted, and it is still in widespread use today. In this subsection, we shall consider this *finite-difference* method as applied to the heat equation in order to illustrate the various tradeoffs involved. We shall see that there are three main considerations when discretizing partial differential equations on grids: First, there is the *consistency* of the difference approximation – the question of whether or not it is really an approximation to the desired partial differential equation in some well-posed limit. Second, there is the *accuracy* of the difference approximation to the continuum equation – the question of how large an error may be expected for a given discretization. Third, there is the *stability* of the difference approximation – the question of whether or not its error remains bounded as the algorithm is applied.

Consider the heat equation,

$$\frac{\partial u(\mathbf{x},t)}{\partial t} = D\nabla^2 u(\mathbf{x},t) \tag{12}$$

in a d-dimensional box of side L, so $\mathbf{x} \in [0,L]^d$, with periodic boundary conditions. We represent the spatial domain by a regular Cartesian grid with N points in each dimension, with points $\mathbf{x_j} = \mathbf{j}\Delta x$, where $\Delta x \equiv L/N$, indexed by $\mathbf{j} \in \{0,\ldots,N-1\}^d$. Likewise, we consider the system at discrete times $t_k = k\Delta t$, where $k \in \{0,1,2,\ldots\}$ and Δt is the time step. We denote the dependent variable at the gridpoint indexed by j and timestep k by $u_{\mathbf{j},k} \equiv u(\mathbf{x_j}, t_k)$.

We can use the *centered finite-difference* approximation of the Laplacian operator,

$$\nabla^2 u(\mathbf{x_j}, t_k) \approx \frac{\sum_{l=1}^d \left(u_{\mathbf{j}-\hat{\mathbf{e}}_l,k} - 2u_{\mathbf{j},k} + u_{\mathbf{j}+\hat{\mathbf{e}}_l,k}\right)}{(\Delta x)^2}, \tag{13}$$

where $\hat{\mathbf{e}}_l$ is the unit vector in dimension l. As the spatial grid spacing Δx goes to zero, this approximation is readily seen to become exact; hence the difference approximation is *consistent*. In fact, by Taylor expanding the right-hand side, we see that it is $\nabla^2 + \mathcal{O}\left((\Delta x)^2\right)$, demonstrating both *consistency* and *accuracy* to $\mathcal{O}\left((\Delta x)^2\right)$.

Likewise, a *two-point finite-difference* approximation of the time derivative is

$$\frac{\partial u(\mathbf{x_j}, t_k)}{\partial t} \approx \frac{u_{\mathbf{j},k+1} - u_{\mathbf{j},k}}{\Delta t}. \tag{14}$$

This becomes exact as the time step Δt goes to zero. By Taylor expanding $u_{\mathbf{j},k+1}$ about $t = t_k$, we see that this approximation is *accurate* to $\mathcal{O}(\Delta t)$.

The full finite-difference approximation to the heat equation is then

$$\frac{u_{\mathbf{j},k+1} - u_{\mathbf{j},k}}{\Delta t} = D\frac{\sum_{l=1}^d \left(u_{\mathbf{j}-\hat{\mathbf{e}}_l,k} - 2u_{\mathbf{j},k} + u_{\mathbf{j}+\hat{\mathbf{e}}_l,k}\right)}{(\Delta x)^2}. \tag{15}$$

This is called an *explicit* finite-difference scheme, since it may be explicitly solved for the new dependent variable at each point on the grid $u_{\mathbf{j},k+1}$,

$$u_{\mathbf{j},k+1} = u_{\mathbf{j},k} + \alpha \sum_{l=1}^d \left(u_{\mathbf{j}-\hat{\mathbf{e}}_l,k} - 2u_{\mathbf{j},k} + u_{\mathbf{j}+\hat{\mathbf{e}}_l,k}\right), \tag{16}$$

where $\alpha \equiv D\Delta t/(\Delta x)^2$. By iterating this equation, beginning with some specified initial conditions on the grid, we can evolve the difference approximation forward in time to obtain an approximation to the solution of the heat equation. We have demonstrated that this approximation is *consistent* and *accurate* to $\mathcal{O}\left((\Delta x)^2\right)$ and $\mathcal{O}(\Delta t)$. We must next assess the *stability* properties of this algorithm. Since all arithmetic done on a computer is of finite precision, a certain amount of error will naturally be introduced into these calculations.

If this error tends to grow exponentially in time, the algorithm will be unstable. We can determine this by applying a classical linear stability argument to the difference equation. Let

$$u_{\mathbf{j},k} = U_{\mathbf{j},k} + \xi_{\mathbf{j},k}, \tag{17}$$

where U is the exact solution to the difference equation, and ξ is error. Since the difference equation is linear and homogeneous, it is clear that ξ will satisfy it,

$$\xi_{\mathbf{j},k+1} = \xi_{\mathbf{j},k} + \alpha \sum_{l=1}^{d} \left(\xi_{\mathbf{j}-\hat{\mathbf{e}}_l,k} - 2\xi_{\mathbf{j},k} + \xi_{\mathbf{j}+\hat{\mathbf{e}}_l,k}\right), \tag{18}$$

and that we can solve it by Fourier analysis. Letting

$$\tilde{\xi}_{\mathbf{m},k} = \sum_{\mathbf{j}} \xi_{\mathbf{j},k} \exp\left(-\frac{2\pi i \mathbf{j}\cdot\mathbf{m}}{N}\right), \tag{19}$$

so that

$$\xi_{\mathbf{j},k} = \frac{1}{N} \sum_{\mathbf{m}} \tilde{\xi}_{\mathbf{m},k} \exp\left(+\frac{2\pi i \mathbf{m}\cdot\mathbf{j}}{N}\right), \tag{20}$$

we substitute this into our difference equation to find the evolution equation of the $\mathbf{m}$'th mode,

$$\tilde{\xi}_{\mathbf{m},k+1} = \left[1 - 2d\alpha + 2\alpha \sum_{l=1}^{d} \cos\left(\frac{2\pi m_l}{N}\right)\right] \tilde{\xi}_{\mathbf{m},k}. \tag{21}$$

Clearly, this iteration will be stable for

$$-1 \le 1 - 2d\alpha + 2\alpha \sum_{l=1}^{d} \cos\left(\frac{2\pi m_l}{N}\right) \le +1. \tag{22}$$

The right inequality is easily seen to always be satisfied; the left requirement is most stringent for the $m_l = N/2$ mode, for which it demands that $2d\alpha \le 1$, or

$$\Delta t \le \frac{(\Delta x)^2}{2dD}. \tag{23}$$

This is called the *Courant condition* for the explicit finite-difference approximation to the heat equation. It states that the time step must scale as the square of the spatial grid size. This is a stringent requirement. It means that if one halves the grid spacing and doubles the number of grid points, presumably in an effort to increase accuracy, one must also decrease the time step by a factor of *four* and iterate for four times as many time steps.

The scaling of Δt with the square of Δx is characteristic of parabolic differential equations. As another physically relevant example, we consider the Navier-Stokes equations for a three-dimensional, viscous, incompressible fluid,

$$\boldsymbol{\nabla}\cdot\mathbf{v}(\mathbf{x},t) = 0 \tag{24}$$

$$\frac{\partial\mathbf{v}(\mathbf{x},t)}{\partial t} + \mathbf{v}(\mathbf{x},t)\cdot\boldsymbol{\nabla}\mathbf{v}(\mathbf{x},t) = -\boldsymbol{\nabla}P(\mathbf{x},t) + \frac{1}{\mathrm{Re}}\nabla^2\mathbf{v}(\mathbf{x},t), \tag{25}$$

where Re is the Reynold's number. Though a rigorous stability analysis for a nonlinear equation is generally much more difficult, since the equation is parabolic we might expect a Courant condition similar to that of the heat equation. Since the inverse of the Reynold's number plays the role of the diffusivity, we have

$$\Delta t \sim \mathrm{Re}(\Delta x)^2, \tag{26}$$

so that the number of time steps needed to integrate one unit of physical time is

$$N_t \sim \frac{1}{\Delta t} \sim \frac{1}{\mathrm{Re}(\Delta x)^2}, \tag{27}$$

and the number of grid points in one unit of physical spatial distance is

$$N_x \sim \frac{1}{\Delta x}. \tag{28}$$

The computational complexity of updating the entire three-dimensional lattice one unit of physical time is then

$$C \sim N_t N_x^3 \sim \frac{1}{\mathrm{Re}(\Delta x)^5}. \tag{29}$$

We must then decide how large Δx needs to be according to accuracy considerations. Since the length scale of the smallest eddies in three-dimensional Navier-Stokes flow scale as $\mathrm{Re}^{-3/4}$ (see [Landau 1982] for details), to resolve these we must take $\Delta x \sim \mathrm{Re}^{-3/4}$ so that

$$C \sim \mathrm{Re}^{11/4}. \tag{30}$$

Thus, computational complexity scales very strongly with Reynold's number. At the time of this writing, the very highest Reynold's numbers that have been simulated are in the vicinity of 10^4 or 10^5. To increase this by an order of magnitude will require nearly three orders of magnitude more computing power. Since Reynold's numbers of interest to, for example, aircraft manufacturers are in the vicinity of 10^7, direct simulations of the Navier-Stokes equations are not an option anytime in the forseeable future; instead, various approximations are used to avoid having to track the smallest eddies.

It is clear that there is much incentive to try to circumvent the Courant condition. One way of doing this is to use *implicit finite-differencing* schemes, as opposed to the explicit method described above. To pursue our above example, suppose that we replace the finite-difference in Eq. (16) by the following, in which the right-hand side is averaged over its values at t_k and t_{k+1}; one may think of this as an estimate of the rate of change at $t_{k+1/2}$. We get

$$u_{\mathbf{j},k+1} = u_{\mathbf{j},k} + \frac{\alpha}{2}\sum_{l=1}^{d}\left[(u_{\mathbf{j}-\hat{\mathbf{e}}_l,k} - 2u_{\mathbf{j},k} + u_{\mathbf{j}+\hat{\mathbf{e}}_l,k}) + (u_{\mathbf{j}-\hat{\mathbf{e}}_l,k+1} - 2u_{\mathbf{j},k+1} + u_{\mathbf{j}+\hat{\mathbf{e}}_l,k+1})\right]. \tag{31}$$

This is called the *Crank-Nicholson* differencing scheme for the heat equation. By arguments similar to those presented above, it can be shown to be *unconditionally stable*. The limitation on the time step Δt is therefore set by *accuracy*, rather than *stability*, considerations. The drawback to implicit schemes is that it is impossible to explicitly solve for the updated dependent variable at each site. Rather, one typically obtains a *banded matrix* equation for the updated dependent variable. For example, specializing to one spatial dimension for simplicity, Eq. (31) may be written

$$-\frac{\alpha}{2}u_{j-1,k+1} + (1+\alpha)\,u_{j,k+1} - \frac{\alpha}{2}u_{j+1,k+1} = \frac{\alpha}{2}u_{j-1,k} + (1-\alpha)\,u_{j,k} + \frac{\alpha}{2}u_{j+1,k}, \tag{32}$$

which makes it clear that $u_{j,k+1}$, for $j \in \{0,\ldots,N-1\}$, is the solution to a *tridiagonal matrix* equation. (In two spatial dimensions this would be *pentadiagonal*, etc.) The gain of using a larger time step must thus be weighed against the expense of solving this type of equation. Methods for solution of banded matrix problems are discussed in the next subsection.

4.2 Banded Matrix Equations

The general tridiagonal system can be written

$$L_i X_{i-1} + D_i X_i + U_i X_{i+1} = R_i, \tag{33}$$

for $i \in \{1,\ldots,N\}$, with L_1 and U_N understood to be zero. Numerous methods for solving such systems of equations are known and in widespread use. These tend to fall in one of three categories: *Direct solvers* provide an algorithm whereby the solution to the matrix equation is obtained in a finite number of steps, carried out in a prescribed sequence; *recursive solvers* break down the problem into two or more smaller problems, which are in turn broken down, in recursive fashion, until the resulting system is small enough to solve explicitly; *iterative solvers* provide an iteration for updating the unknown which converges to the correct answer. We consider each of these in turn.

A *direct tridiagonal solver* may be derived by rewriting Eq. (33) as follows:

$$L_i\left(X_{i-1}+\frac{D_i-P_i}{L_i}X_i\right)+P_i\left(X_i+\frac{U_i}{P_i}X_{i+1}\right)=R_i, \tag{34}$$

where the P_i are as yet unspecified. We define

$$Y_i\equiv X_i+\frac{U_i}{P_i}X_{i+1}, \tag{35}$$

and determine the P_i by demanding that

$$\frac{U_{i-1}}{P_{i-1}}=\frac{D_i-P_i}{L_i}, \tag{36}$$

so that

$$L_iY_{i-1}+P_iY_i=R_i. \tag{37}$$

Thus, Eqs. (36) and (37) yield recursions for the P_i and Y_i,

$$P_i=D_i-\frac{L_iU_{i-1}}{P_{i-1}} \tag{38}$$

and

$$Y_i=\frac{R_i-L_iY_{i-1}}{P_i}. \tag{39}$$

These may be recursed forward in i from the initial condition $P_1=D_1$ (and remembering that $L_1=0$) to obtain all the P's and Y's. Once these are known, Eq. (35) yields a *backward* recursion in i for the desired unknowns,

$$X_i=Y_i-\frac{U_i}{P_i}X_{i+1}. \tag{40}$$

Since two recursive sweeps of N steps are required, note that this algorithm has computational complexity of $\mathcal{O}(N)$. A Fortran 90 implementation of the above-described algorithm might be,

```
SUBROUTINE TRIDIAGONAL_SOLVE(X, L, D, U, R)

  IMPLICIT NONE
  INTEGER :: j, n
  REAL :: p
  REAL, DIMENSION(:) :: X, L, D, U, R
  REAL, ALLOCATABLE, DIMENSION(:) :: Y

  n = SIZE(D)
  ALLOCATE(Y(n))

  p = D(1)
  IF(p == 0.0) THEN
     PRINT *, " Error:  Ill-conditioned matrix.  Pivot: 1"
     STOP
  END IF
  X(1) = R(1)/p

  DO j = 2, n
     Y(j) = U(j - 1)/p
     p = D(j) - L(j)*Y(j)
```

```
    IF(p == 0.0) THEN
       PRINT *, " Error:  Ill-conditioned matrix.  Pivot: ", j
       STOP
    END IF
    X(j) = (R(j) - L(j)*X(j - 1))/p
  END DO

  DO  j = n - 1, 1, -1
    X(j) = X(j) - Y(j + 1)*X(j + 1)
  END DO

  DEALLOCATE(Y)

END SUBROUTINE TRIDIAGONAL_SOLVE
```

One problem with the above routine is that the recursions are inherently serial. If parallelism is desired, a different strategy must be adopted. One such strategy is that of a *recursive solver* in which the problem is successively broken down into smaller pieces. Consider, for example, the following three rows of the matrix relationship in Eq. (33):

$$\begin{aligned} L_{i-1}X_{i-2} + D_{i-1}X_{i-1} + U_{i-1}X_i & = R_{i-1} \\ L_i \quad X_{i-1} + D_i \quad X_i + U_i \quad X_{i+1} & = R_i \\ L_{i+1}X_i + D_{i+1}X_{i+1} + U_{i+1}X_{i+2} & = R_{i+1}. \end{aligned} \tag{41}$$

Eliminating X_{i-1} using the first of these, and X_{i+1} using the last, the middle equation becomes

$$\begin{aligned} \left(-\frac{L_i L_{i-1}}{D_{i-1}}\right) X_{i-2} & + \left(D_i - \frac{L_i U_{i-1}}{D_{i-1}} - \frac{U_i L_{i+1}}{D_{i+1}}\right) X_i + \left(-\frac{U_i U_{i+1}}{D_{i+1}}\right) X_{i+2} \\ & = \left(R_i - \frac{L_i R_{i-1}}{D_{i-1}} - \frac{U_i R_{i+1}}{D_{i+1}}\right). \end{aligned} \tag{42}$$

Since this equation couples X_i only to X_{i-2} and X_{i+2}, this has had the effect of breaking the tridiagonal system of N equations into two tridiagonal systems of $N/2$ equations each. If N is a power of two, this process can be continued recursively, until a two-by-two system is obtained, for which the general solution is simple to write down. Since modern languages, such as Fortran 90, support recursion – subroutines that can call themselves – it is a simple matter to capture the above in code,

```
RECURSIVE SUBROUTINE NESTED_BISECTION(X, L, D, U, R)

  IMPLICIT NONE
  INTEGER :: n, nh
  REAL :: det
  REAL, DIMENSION(:) :: X, L, D, U, R
  REAL, DIMENSION(:), ALLOCATABLE :: LH, DH, UH, XH, RH

  n = SIZE(D)
  IF(n == 1) THEN
     X(1) = R(1)/D(1)
     RETURN
  ELSE IF(n == 2) THEN
     det = D(1)*D(2) - U(1)*L(2)
     X(1) = (D(2)*R(1) - U(1)*R(2))/det
     X(2) = (D(1)*R(2) - L(2)*R(1))/det
     RETURN
  ELSE IF(MOD(n, 2) == 0) THEN
```

```
      nh = n/2
      ALLOCATE(LH(nh), DH(nh), UH(nh), XH(nh), RH(nh))

      ! Solve for even variables:
      LH(2:nh) = -L(4:n:2)*L(3:n-1:2)/D(3:n-1:2)
      DH(1:nh-1) = D(2:n-2:2) - L(2:n-2:2)*U(1:n-3:2)/D(1:n-3:2) - &
                                & U(2:n-2:2)*L(3:n-1:2)/D(3:n-1:2)
      DH(nh) = D(n) - L(n)*U(n-1)/D(n-1)
      UH(1:nh-1) = -U(2:n-2:2)*U(3:n-1:2)/D(3:n-1:2)
      RH(1:nh-1) = R(2:n-2:2) - L(2:n-2:2)*R(1:n-3:2)/D(1:n-3:2) - &
                                & U(2:n-2:2)*R(3:n-1:2)/D(3:n-1:2)
      RH(nh) = R(n) - L(n)*R(n-1)/D(n-1)
      CALL NESTED_BISECTION(X(2::2), LH, DH, UH, RH)

      ! Solve for odd values:
      LH(2:nh) = -L(3:n-1:2)*L(2:n-1:2)/D(2:n-1:2)
      DH(1) = D(1) - U(1)*L(2)/D(2)
      DH(2:nh) = D(3:n-1:2) - L(3:n-1:2)*U(2:n-2:2)/D(2:n-2:2) - &
                              & U(3:n-1:2)*L(4:n:2)/D(4:n:2)
      UH(1:nh-1) = -U(1:n-1:2)*U(2:n:2)/D(2:n:2)
      RH(1) = R(1) - U(1)*R(2)/D(2)
      RH(2:nh) = R(3:n-1:2) - L(3:n-1:2)*R(2:n-2:2)/D(2:n-2:2) - &
                              & U(3:n-1:2)*R(4:n:2)/D(4:n:2)
      CALL NESTED_BISECTION(X(1::2), LH, DH, UH, RH)

      DEALLOCATE(LH, DH, UH, XH, RH)

   ELSE
      PRINT *, " Error: Problem size contains prime factors other than 2."
      STOP

   ENDIF

END SUBROUTINE NESTED_BISECTION
```

Note that the above routine treats $N = 1$ and $N = 2$ as special cases. For even $N > 2$ it divides up the problem into two smaller ones as indicated above, and calls itself for each of those two smaller problems. If an odd $N > 2$ is ever encountered, it means that the initial problem size was not a power of two.

Note also that if N contains factors of 3 as well as 2 it is straightforward to work out a variant of the above algorithm that divides the system into three subsystems of size $N/3$, and to add that to the above recursive subroutine, with the test that `MOD(N, 3) == 0`. By working out this treatment for the first several prime numbers, a routine can be created that works for any N containing only those prime factors.

This routine is more amenable to parallel computation than the direct tridiagonal solver described above, since the arithmetic operations needed to compute the matrix coefficients of the smaller systems can be performed in parallel.

4.3 The Galerkin Approximation and Finite-Element Methods

Representing a function by its values at discrete gridpoints is, of course, only one way of approximating it. More generally, we might consider expanding a function in a series of (not necessarily orthogonal) basis functions, $\psi_\alpha(\mathbf{x})$. These may be a complete set if α is allowed to range over an infinite (discrete and/or continuous) set of values. We can then approximate the dependent variable $u(\mathbf{x}, t)$ by restricting α to a finite subset $\mathcal{A}$ of its possible values. Thus, we write

$$u(\mathbf{x},t) \approx U(\mathbf{x},t) \equiv \sum_{\alpha\in\mathcal{A}} U_\alpha(t)\psi_\alpha(\mathbf{x}). \tag{43}$$

An error will generally be incurred if we substitute the approximate dependent variable $U(\mathbf{x},t)$ in any given dynamical equation for the field $u(\mathbf{x},t)$. Returning to our example of the heat equation, this error may be defined,

$$E(\mathbf{x},t) \equiv \frac{\partial U(\mathbf{x},t)}{\partial t} - D\nabla^2 U(\mathbf{x},t). \tag{44}$$

The *Galerkin approximation* fixes the coefficients $U_\alpha(t)$ by demanding that this error be orthogonal to all of the basis functions in the set α, and hence orthogonal to the entire subspace spanned by them. In order to have some notion of orthogonality, an inner product must be defined; this is most often taken to be an integral of the product of the functions,

$$\langle f \,| g\,\rangle = \int d^3x\; W(\mathbf{x}) f(\mathbf{x}) g(\mathbf{x}), \tag{45}$$

where $W(\mathbf{x})$ is a judiciously chosen weighting factor. Thus, we demand that

$$0 = \langle \psi_\alpha \,| E\,\rangle = \left\langle \psi_\alpha \,\middle|\, \frac{\partial U}{\partial t} - D\nabla^2 U \right\rangle \tag{46}$$

for all $\alpha \in \mathcal{A}$. Substituting Eq. (43), we quickly obtain

$$A_{\alpha\beta}\frac{dU_\beta(t)}{dt} = B_{\alpha\beta}U_\beta(t), \tag{47}$$

where there are understood summations over the repeated index $\beta \in \mathcal{A}$, and where we have defined the matrix A with elements

$$A_{\alpha\beta} \equiv \langle \psi_\alpha \,|\psi_\beta\,\rangle \tag{48}$$

and B with elements

$$B_{\alpha\beta} \equiv \left\langle \psi_\alpha \,\middle|\nabla^2\psi_\beta\,\right\rangle. \tag{49}$$

As long as A is nonsingular, this can be solved for the $dU_\beta(t)/dt$ which can then be integrated by any one of a number of different numerical integration schemes. Fed back into Eq. (43), this gives the finite-element approximation to the solution $u(\mathbf{x},t)$.

As a concrete example, let us consider one spatial dimension and suppose that the functions $\psi_\alpha(\mathbf{x})$ are the linear basis splines,

$$\psi_\alpha(x) \equiv \begin{cases} (x-x_{\alpha-1})/(x_\alpha - x_{\alpha-1}) & x_{\alpha-1} \le x < x_\alpha \\ (x_{\alpha+1}-x)/(x_{\alpha+1}-x_\alpha) & x_\alpha \le x < x_{\alpha+1} \\ 0 & \text{otherwise,} \end{cases} \tag{50}$$

illustrated in Fig. 7. For the case of equally spaced gridpoints, $x_\alpha = \alpha\Delta x$ for $\alpha \in \{0,\ldots,N-1\}$, and if the weight function $W(x)$ is taken to be unity we find that A and B are the tridiagonal matrices with elements

$$A_{\alpha\beta} = \begin{cases} 2\Delta x/3 & \alpha = \beta \\ \Delta x/6 & |\alpha-\beta| = 1 \\ 0 & \text{otherwise,} \end{cases} \tag{51}$$

and

$$B_{\alpha\beta} = \begin{cases} 2/\Delta x & \alpha = \beta \\ -1/\Delta x & |\alpha-\beta| = 1 \\ 0 & \text{otherwise.} \end{cases} \tag{52}$$

Thus, Eq. (47) can be solved using one of the previously discussed tridiagonal algorithms, and the system can be integrated forward in time. Note that this system is quite similar to that of Eq. (15), except that the matrix $A_{\alpha\beta}$ has the effect of smoothing the time differencing scheme a bit.

Because the dependent variable is represented as a sum of the linear splines illustrated in Fig. 7, it is clear that the function subspace that we are considering is that of all piecewise linear functions. It is also clear that we would still get a tridiagonal system if the gridpoints were spaced irregularly. Thus, finite-element methods can be easily applied to situations in which one needs more grid refinement in one part of a spatial domain than another. Moreover, the method could be used with a smoother set of basis splines than the linear splines used above. Cubic basis splines, assuring continuity of first and second derivatives at the gridpoints, are commonly used; in this situation, the resulting matrix equation will still be banded, though it will be pentadiagonal, rather than tridiagonal.

The method assumes great importance in more than one spatial dimension, and for problems with irregular boundary conditions, such as are often encountered in engineering applications. In two dimensions, for example, it is possible to break up a domain into triangular or other polygonal regions, and define spline functions on them with the appropriate continuity properties. Tensor products of one dimensional basis splines can be used to create multidimensional basis splines, and the Galerkin approximation can still be employed to derive a finite-element method. In such situations, the A matrix is generally no longer neatly banded, though it is still likely to be *sparse* for problems involving purely local differential operators; a number of efficient schemes exist for storing and inverting such sparse matrices. Finally, we note that the method is also sometimes used for the solution of integrodifferential equations for which the A matrix is dense.

5 Spectral Methods

5.1 Spectral Representation of Data

In the applications of the Galerkin method that we examined above, we chose basis functions that were *local* – i.e., had compact support. This restriction is unnecessary. One might consider, for example, expanding a function on a periodic domain in a discrete Fourier series, and representing it on a computer by the coefficients of this series. For problems in other geometries, or with other linear operators, other basis functions may be more appropriate. While Fourier representations diagonalize the ubiquitous del operator (and hence the gradient, divergence and Laplacian) in periodic geometry, a Fourier-Bessel representation may be more appropriate for cylindrical geometry, spherical harmonics may be more appropriate for spherical geometry, and so on. Of these, only the Fourier transform with sine/cosine basis functions has a *fast* version (the so-called *Fast-Fourier Transform*, or FFT), so it is naturally the most useful in this regard.

It is easy to see that the method diagonalizes linear, homogeneous equations, reducing them to a set of independent ordinary differential equations to be integrated. The challenge is therefore to deal with inhomogeneities and/or nonlinearities which will couple the modes. The example of the heat equation, which served us well in the last section, is inadequate to illustrate this issue, so we add a nonlinear term $|\boldsymbol{\nabla} u(\mathbf{x},t)|^2$ on the left, and some inhomogeneous noise $\eta(\mathbf{x},t)$ on the right; the result is the *Kardar-Parisi-Zhang Equation*,

$$\frac{\partial u(\mathbf{x},t)}{\partial t} + \frac{1}{2}\left|\boldsymbol{\nabla} u(\mathbf{x},t)\right|^2 = D\nabla^2 u(\mathbf{x},t) + \eta(\mathbf{x},t), \tag{53}$$

which has received wide attention as a model of interfacial growth [Kardar 1986]. If we expand $u(\mathbf{x},t)$ in a set of *orthogonal* basis functions,

$$u(\mathbf{x},t) = \sum_\alpha u_\alpha(t)|\psi_\alpha\rangle \tag{54}$$

for $\alpha \in \{1,\ldots,N\}$, we find that

$$\frac{du_\alpha(t)}{dt} + \frac{1}{2}\sum_{\beta,\gamma} u_\beta(t)u_\gamma(t)\langle\phi_\alpha|\boldsymbol{\nabla}\phi_\beta\cdot\boldsymbol{\nabla}\phi_\gamma\rangle = D\sum_\beta u_\beta(t)\langle\phi_\alpha|\nabla^2|\phi_\beta\rangle + \eta_\alpha(t), \tag{55}$$

where $\eta_\alpha(t) \equiv \langle\phi_\alpha|\eta\rangle$.

To make this concrete, let us take periodic boundary conditions in a cube of side L,

$$|\phi_{\mathbf{j}}\rangle = \exp\left(\frac{2\pi i\mathbf{j}\cdot\mathbf{x}}{L}\right), \tag{56}$$

so that the basis functions are indexed by the wavenumber $\mathbf{j}$. The Laplacian operator is then diagonalized, and one sum drops out of the nonlinear term to yield

$$\frac{du_{\mathbf{j}}(t)}{dt} - \frac{2\pi^2}{L^2}\sum_{\mathbf{k}} \mathbf{k}\cdot(\mathbf{j}-\mathbf{k})u_{\mathbf{k}}(t)u_{\mathbf{j}-\mathbf{k}}(t) = -\frac{4\pi^2 D}{L^2}|\mathbf{j}|^2 u_{\mathbf{j}}(t) + \eta_{\mathbf{j}}(t). \tag{57}$$

Note that the quadratic nonlinearity in the original equation has given rise to a corresponding quadratic nonlinearity in the equation for the mode amplitudes. Suppose that we truncate our spectral representation of $u(\mathbf{x},t)$ at some finite number N of modes in each direction. For a d-dimensional spatial domain, there will thus be a total of N^d modes. It follows that the upper limit of each of the d summations on the left-hand side of Eq. (57) should be N. This yields a closed set of equations for the mode amplitudes. As long as N is chosen to be large enough that the smallest characteristic features of the flow can be represented, this is a valid numerical approximation.

On the other hand, note that the nonlinear term requires on the order of N^{2d} steps to compute, since there are N^d equations, each with d nested N-term sums. Since a Fast Fourier Transform of N^d data requires only on the order of $N^d\log(N^d)\sim dN^d\log(N)$ steps, it actually becomes less expensive to compute the gradient $\boldsymbol{\nabla} u$ in Fourier space, square it in real space, and then transform the result back to Fourier space. This strategy is called a *pseudo-spectral algorithm*. Symbolically, this procedure may be written,

$$\frac{du_{\mathbf{j}}(t)}{dt} - \frac{2\pi^2}{L^2}\mathcal{F}\left[\left|\mathcal{F}^{-1}\left(\mathbf{k}u_{\mathbf{k}}\right)\right|^2\right] = -\frac{4\pi^2 D}{L^2}|\mathbf{j}|^2 u_{\mathbf{j}}(t) + \eta_{\mathbf{j}}(t), \tag{58}$$

where $\mathcal{F}$ and $\mathcal{F}^{-1}$ denote the Fourier transform and its inverse, respectively.

5.2 Operator splitting

Multiplication of the dependent variable by a spatially inhomogeneous term can also couple the modes. A classic example of this is provided by the time-dependent Schrödinger equation,

$$i\frac{\partial\Psi(\mathbf{x},t)}{\partial t} = -\frac{1}{2}\nabla^2\Psi(\mathbf{x},t) + V(\mathbf{x})\Psi(\mathbf{x},t). \tag{59}$$

We could proceed as above to solve this equation using spectral methods, but there are other considerations: Specifically, we know that Schrödinger evolution should be *unitary*; that is, it should conserve total probability $|\Psi(\mathbf{x},t)|^2$. It would be nice to develop a numerical method that respected this constraint.

Toward this end, we write the Schrödinger equation in the following format:

$$i\frac{\partial\Psi}{\partial t} = \hat{H}\Psi \equiv \hat{T}\Psi + \hat{V}\Psi, \tag{60}$$

where $\hat{T}\equiv -\nabla^2/2$ and $\hat{V} = V(\mathbf{x},t)$ are the kinetic and potential energy operators, respectively. This equation has the formal solution

$$\Psi(t+\Delta t) = \exp\left(-i\Delta t\hat{H}\right)\Psi(t) = \exp\left[-i\Delta t\left(\hat{T}+\hat{V}\right)\right]\Psi(t). \tag{61}$$

Now we cannot write $\exp[-i\Delta t(\hat{T}+\hat{V})] = \exp[-i\Delta t\hat{T}]\exp[-i\Delta t\hat{V}]$ since the operators $\hat{T}$ and $\hat{V}$ generally do not commute. Motivated by the Baker-Campbell-Hausdorff theorem [Belinfante 1972], however, we can write

$$\exp[-i\Delta t(\hat{T}+\hat{V})] = \exp[-i\Delta t\hat{T}]\exp[-i\Delta t\hat{V}] + \mathcal{O}(\Delta t^2), \tag{62}$$

or

$$\exp[-i\Delta t(\hat{T}+\hat{V})] = \exp[-\frac{i}{2}\Delta t\hat{V}]\exp[-i\Delta t\hat{T}]\exp[-\frac{i}{2}\Delta t\hat{V}] + \mathcal{O}(\Delta t^3), \tag{63}$$

etc. If we integrate for $\mathcal{O}(\Delta t^{-1})$ time steps, the first (resp., second) of these will give rise to an accumulated error of order $\mathcal{O}(\Delta t)$ (resp., $\mathcal{O}(\Delta t^2)$). Choosing the first for simplicity, we have

$$\Psi(t+\Delta t) \approx \exp\left(-i\Delta t\hat{T}\right)\exp\left(-i\Delta t\hat{V}\right)\Psi(t). \tag{64}$$

Now the first of these operators, $\exp(-i\Delta t\hat{T})$, is diagonalized in Fourier space since

$$\exp\left(-i\Delta t\hat{T}\right) = \exp\left(\frac{i\Delta t}{2}\nabla^2\right) = \mathcal{F}^{-1}\exp\left(-\frac{i\Delta t}{2}k^2\right)\mathcal{F}, \tag{65}$$

where k is the wavevector; the second, $\exp(-i\Delta t\hat{V})$, is diagonalized in real space. We can thus write

$$\Psi(t+\Delta t) \approx \mathcal{F}^{-1}\exp\left(-\frac{i\Delta t}{2}k^2\right)\mathcal{F}\exp\left(-i\Delta t\hat{V}\right)\Psi(t). \tag{66}$$

Given the wavefunction at time t, the complete algorithm for one time step is therefore to perform a local (in real space) multiplication by $\exp(-i\Delta t\hat{V})$, Fourier transform the result, perform a local (in Fourier space) multiplication by $\exp(-i\Delta tk^2/2)$, and invert the Fourier transform. Like the KPZ equation discussed above, note that the portion of the dynamics that does not mix modes is treated in Fourier space, while the portion that does mix modes is treated in real space; that is, it is pseudo-spectral. Also note that this algorithm has the virtue of being *exactly* unitary – or at least to within roundoff error – even for finite Δt.

Note that the presence of the global Fourier transform means that the algorithm is effectively implicit, insofar as the new wavefunction at any one point depends on the old one everywhere on the grid. In fact, the situation is actually worse than it was for the implicit finite-difference schemes discussed earlier, since those could be handled by a tridiagonal solver in $\mathcal{O}(N)$ operations, where N is the number of grid points. The FFT, by contrast, will require $\mathcal{O}(N\log N)$ operations. Thus, it would be nice to have a fully *explicit*, operator-splitting methodology for integrating Schrödinger's equation, maintaining exact unitarity. Such a method was discovered only fairly recently [DeRaedt 1987]. To understand it, consider a one-dimensional, periodic, finite-difference representation of the Hamiltonian operator that was used above,

$$\begin{aligned}
\hat{H} &= -\frac{1}{2}\nabla^2 + V(\mathbf{x}) \\
&= -\frac{1}{2}\begin{pmatrix} 2 & -1 & 0 & 0 & 0 & \cdots & -1 \\ -1 & 2 & -1 & 0 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & 0 & \cdots & 0 \\ 0 & 0 & -1 & 2 & -1 & \cdots & 0 \\ 0 & 0 & 0 & -1 & 2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ -1 & 0 & 0 & 0 & 0 & \cdots & 2 \end{pmatrix} + \begin{pmatrix} V_1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & V_2 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & V_3 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & V_4 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & V_5 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & V_N \end{pmatrix} \\
&= \hat{H}_{\mathrm{even}} + \hat{H}_{\mathrm{odd}},
\end{aligned} \tag{67}$$

where we have defined $V_j \equiv V(x_j)$, and the matrix operators

$$\hat{H}_{\mathrm{even}} \equiv \frac{1}{2}\begin{pmatrix} V_1-1 & & & & & \cdots & 1 \\ & V_2-1 & 1 & & & \cdots & \\ & 1 & V_3-1 & & & \cdots & \\ & & & V_4-1 & 1 & \cdots & \\ & & & 1 & V_5-1 & \cdots & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & & & & & \cdots & V_N-1 \end{pmatrix} \tag{68}$$

and

$$\hat{H}_{\mathrm{odd}} \equiv \frac{1}{2}\begin{pmatrix} V_1-1 & 1 & & & & \cdots & \\ 1 & V_2-1 & & & & \cdots & \\ & & V_3-1 & 1 & & \cdots & \\ & & 1 & V_4-1 & & \cdots & \\ & & & & V_5-1 & \cdots & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ & & & & & \cdots & V_N-1 \end{pmatrix}. \tag{69}$$

We have purposely suppressed the zero matrix elements of these operators to emphasize the fact that both $\hat{H}_{\text{even}}$ and $\hat{H}_{\text{odd}}$ are direct sums of 2×2 matrices of the form

$$\hat{h}_j \equiv \begin{pmatrix} V_j - 1 & 1 \\ 1 & V_{j+1} - 1 \end{pmatrix}, \tag{70}$$

for j even and odd, respectively.

It is now possible to use operator splitting on the division of the Hamiltonian into $\hat{H}_{\text{even}}$ and $\hat{H}_{\text{odd}}$, rather than its division into $\hat{T}$ and $\hat{V}$. Because $\hat{H}_{\text{even}}$ and $\hat{H}_{\text{odd}}$ are direct sums of 2×2 blocks, however, their exponentiation will be likewise; that is, in order to exponentiate this Hamiltonian, all one needs to do is exponentiate a 2×2 Hermitian matrix. This is an elementary exercise, and the evolution will consist of the application of a sequence of 2×2 block unitary matrices to the wavefunction. Since these combine only neighboring values of the wavefunction on the grid, the evolution will be explicit as well as exactly unitary. This method also has the advantage of being straightforwardly parallelized [Richardson 1991]. Finally we note that, more recently, other fully explicit, exactly unitary methods have been developed for the integration Schrödinger's equation [Boghosian 1997], but they are based on a lattice-gas approach rather than on operator splitting.

5.3 Spectral-element methods and wavelets

The orthogonality of the spectral basis functions in the above examples has precluded the need to invert a tridiagonal matrix, as was necessary for the Galerkin method in Eq. (47). Moreover, sine and cosine basis functions have the virtue of diagonalizing differential operators, and possessing a "fast" transform – the FFT. At the same time, sine and cosine basis functions have certain disadvantages. All of our above examples employed very simple boundary conditions. Indeed, sine and cosine basis functions are easiest for periodic boundary conditions. Moreover, since sine and cosine functions lack the compact support of basis splines, they are awkward for representing localized features.

A relatively recent numerical technique called the *spectral element* method seeks to combine the best features of spectral methods and finite elements. Typically, it uses polyhedral elements; since these have a simple geometry, it is straightforward to adopt a spectral representation within each of them. The boundary conditions between elements give rise to a sparse matrix problem, just as they did for the Galerkin method. By making the elements larger and using more modes to compensate for any loss of resolution, however, the size of this sparse matrix problem can be reduced; this must be weighed against the benefit of using smaller, more localized elements to represent compact features.

The tradeoff, as stated above, would seem to be one between the *orthogonality* of the basis functions, useful for simplifying the solution procedure, and their *compact support*, useful for representing localized features. For example, basis splines are clearly compact but nonorthogonal, whereas sines and cosines are orthogonal but noncompact. From this point of view, one of the more exciting discoveries of the past ten years has been the fact that it is possible to construct sets of basis functions, called *wavelet bases*, that have both orthogonality and compact support. Like basis splines, these consist of different translations of a single canonical function, but they also include different scales as well. That is, a general wavelet expansion is of the form,

$$f(x) = \sum_{j,k} b_{jk} W(2^j x - k) \tag{71}$$

where $W(x)$ can be chosen to satisfy a mixture of essential and optional requirements.

To get a general idea of how a wavelet basis is constructed, suppose that we are using the basis of normalized box functions,

$$\phi_j^{(\ell)}(x) \equiv \begin{cases} 1 & j \leq x < j+1 \\ 0 & \text{otherwise} \end{cases} \tag{72}$$

where $j \in \{0, 1, 2, \ldots, 2^\ell - 1\}$, to represent piecewise constant functions on the interval $0 \leq x < 2^\ell$. We can obviate the need for the j subscript by noting that

$$\phi_j^{(\ell)}(x) = \phi^{(\ell)}(x - j) \tag{73}$$

where $\phi^{(\ell)}(x) \equiv \phi_0^{(\ell)}(x)$; that is, the basis functions for different values of j are just translations of each other.

Next we consider the coarser basis obtained by scaling this one by a factor of two. We define the scaled function

$$\phi^{(\ell-1)}(x) = \frac{1}{2}\phi^{(\ell)}\left(\frac{x}{2}\right), \tag{74}$$

and construct a basis with its translations,

$$\phi_j^{(\ell-1)}(x) = \phi^{(\ell-1)}(x-2j). \tag{75}$$

The overall factor of $\frac{1}{2}$ on the right-hand side of Eq. (74) has been included to maintain normalization. Note that this coarser basis has only half as many basis functions as the one we started with. The finer basis is therefore capable of representing everything that the coarser basis can represent and more; specifically, it can represent a factor of two finer detail. To make this more explicit, note that each member of the coarse grid can be written as the sum of two members of the finer grid,

$$\phi_j^{(\ell-1)}(x) = \frac{1}{2}\left[\phi_{2j}^{(\ell)}(x) + \phi_{2j+1}^{(\ell)}(x)\right]. \tag{76}$$

It follows that the *difference* in resolution between the finer and coarser bases – that is, the space of functions that is spanned by the finer basis but is orthogonal to the coarser basis – is spanned by the set of functions

$$W_j^{(\ell-1)}(x) = \frac{1}{2}\left[\phi_{2j}^{(\ell)}(x) - \phi_{2j+1}^{(\ell)}(x)\right]. \tag{77}$$

These functions are called the *wavelets* corresponding to the basis of Eq. (72).

In more abstract terms, let us suppose that the space of functions spanned by the $\phi_j^{(\ell)}(x)$ is called $\Lambda^{(\ell)}$, and that spanned by the $W_j^{(\ell)}(x)$ is called $\Omega^{(\ell)}$. The argument in the above paragraph has demonstrated that

$$\Lambda^{(\ell)} = \Lambda^{(\ell-1)} \oplus \Omega^{(\ell-1)}. \tag{78}$$

That is, we have decomposed the original basis into the direct sum of its coarsening and a space of wavelets. We can now repeat this process with the space $\Lambda^{(\ell-1)}$, etc. The final result will be the decomposition of the original basis into a set of wavelet functions and their translations at each scale (plus possibly a constant function at the very coarsest scale). That is,

$$\Lambda^{(\ell)} = \Lambda^{(0)} \oplus \Omega^{(1)} \oplus \cdots \oplus \Omega^{(\ell-1)}. \tag{79}$$

It is clear from the method of construction that this decomposition results in an orthonormal basis.

Note that not all bases are capable of generating wavelets in the manner explained above. A fundamental requirement, satisfied by the hat function used above, is that each basis function at a given level of the hierarchy can be expressed as a linear combination of basis functions at the next finest level. That is, the basis functions ϕ must be such that

$$\phi(x) = \sum_k c_k \phi(2x-k), \tag{80}$$

for some constants c_k. It is not difficult to see that the condition that ϕ is normalized results in the constraint

$$\sum_k c_k = 2. \tag{81}$$

Not all functions will obey this *dilation equation*. Fortunately, as noted by Strang [Strang 1989], an amazing variety of familiar families of basis functions do obey it. For example, if $c_j = 2\delta_{j,0}$, then ϕ is the Dirac delta function. Our above example showed that the box function obeys the dilation equation with $c_0 = c_1 = 1$. Likewise the piecewise-linear functions of Eq. (50) on a uniform grid as illustrated in Fig. (7) also solve the dilation equation with $c_0 = c_2 = \frac{1}{2}$ and $c_1 = 1$.

The ability of these functions to represent polynomials in x can also be expressed as conditions on the c coefficients [Strang 1989]. Specifically, if the function

$$P(\xi) \equiv \frac{1}{2}\sum_k c_k e^{ik\xi} \tag{82}$$

has a pth order pole at $\xi = \pi$ then the basis will be able to represent exactly all polynomials in x spanned by the set of monomials $\{1, x, x^2, x^3, \ldots, x^{p-1}\}$. Thus, the hat function used in the example mentioned above had $P(\xi) = \exp(ik\xi/2)\cos(k\xi/2)$ which has a pole of order $p = 1$, so it is capable of representing only piecewise constant functions.

Given a basis that obeys the dilation equation for some coefficients c, then the generalization of the hat-function argument given above shows that the corresponding wavelet basis is given by

$$W(x) = \sum_k (-1)^k c_{1-k}\phi(2x-k). \tag{83}$$

All of this formalism makes it straightforward to construct bases satisfying the dilation equation with the ability to represent polynomials of specified order, and to construct the corresponding wavelet basis. Moreover, it can be shown that such wavelet bases, owing to their hierarchical construction, possess a "fast" transform, like the FFT, so that they are of $\mathcal{O}(N\log N)$ computational complexity for N data elements.

It is interesting to note that this procedure does not always result in smooth, well-behaved basis functions. In fact, Daubechies [Daubechies 1988] showed that the choice $c_0 = (1+\sqrt{3})/4$, $c_1 = (3+\sqrt{3})/4$, $c_2 = (3-\sqrt{3})/4$ and $c_3 = (1-\sqrt{3})/4$ results in a wavelet basis that is orthogonal, is compact on $[-1, 2]$, and is capable of exactly representing any piecewise linear function. This function is illustrated in Fig. 8 (though it is differentiable almost nowhere so that any graphical representation of it is rather dubious).

It is clear that wavelet bases that are able to exactly represent higher powers of x are a straightforward generalization of the above. On the other hand, wavelets do not diagonalize differential operators, as does a Fourier representation. Though a detailed discussion of the theory and usage of wavelets is outside the scope of this article, there are by now a very large number of references on this subject. At the time of this writing, this remains a very active area of applied mathematical research.

6 Particulate methods

From the above descriptions of finite-difference, finite-element and spectral methods, the reader may have developed the impression that such methods are adequate for the simulation of any partial differential equation of interest in mathematical physics. In fact, nothing could be further from the truth. Certain partial differential equations are capable of generating structures on arbitrarily small length scales of their dependent variable(s). When we examined finite-difference approximations to the Navier-Stokes equations, for example, we discovered that the need for a grid fine enough to resolve the very smallest vortices was an important and severe restriction; in spite of this, we can feel at least somewhat fortunate that there is some nonzero lower bound on the structures developed by the Navier-Stokes equations at any finite Reynolds number. Imagine how much more difficult it is to simulate the Euler equations of inviscid fluid flow or the Poisson-Vlasov equations of plasma physics, which have no viscous terms to damp out the development of structures below a certain length scale. The structures developed by the evolution of such *Hamiltonian systems* can become arbitrarily filamentary, so that their representation on any fixed spatial grid, or using any fixed set of basis functions, is invalidated in finite time.

In spite of this seeming intractibility, it is possible to make numerical progress on such problems. One approach is to try to develop a spectral method with N modes that is a Hamiltonian system *for all* N. In this way, even the finite-dimensional truncated system is Hamiltonian, and has a symplectic phase-space structure [Symplectic]. For example, this was accomplished for Euler's ideal-fluid equations by Miller et al. [Miller 1992]. While such methods are still not capable of representing arbitrarily filamentary modes, they at least respect the various constraints imposed by Hamiltonian evolution – conservation of phase-space volume and the integral invariants of Poincaré – to numerical accuracy [Channell 1990]; this is analogous to the way that the operator-splitting method described above respected unitarity.

Another general method for simulating systems of this sort has been to abandon the spatial (or spectral) discreteness that has characterized all of the methods that we have considered thus far, and represent the field by the density of local particles, each of which is free to take its location in the continuum. That is, instead of an *Eulerian representation* of a dependent variable $\Psi(\mathbf{x_j}, t)$ at each point of a spatial grid, we might adopt a *Lagrangian representation* in which we keep track of the spatial coordinates, and perhaps weights, of some finite number of particles at each time step. This method does not require any kind of grid at all, though we note in passing that the finiteness of floating-point precision on any real computer is tantamount to the existence of a very fine grid, in that particle positions are restricted to a discrete set of values. Also, there exist hybrid algorithms, such as the *particle-in-cell* method, in which some quantities are represented on a grid and others are represented by discrete particles.

6.1 Short-range interactions

The first and most intuitive application of particulate methods is their use for tracking the positions and velocities of discrete particles, such as molecules, interacting according to a known classical force law. Such *molecular dynamics* (MD) algorithms are distinguished by their application to either *short-range* or *long-range* force laws (or both together). An example of the former is the Lennard-Jones potential,

$$V_{LJ}(r) = V_0 \left[\left(\frac{r_0}{r} \right)^{12} - \left(\frac{r_0}{r} \right)^{6} \right]. \tag{84}$$

This interaction potential drops off so sharply for large r that effects of particles outside of a certain *interaction sphere* can be simply ignored. Alternatively, one may employ a version of this potential for which this cutoff is explicitly enforced by taking

$$V(r) = \begin{cases} V_{LJ}(r) - V_{LJ}(r_1) & \text{for } r < r_1 \\ 0 & \text{otherwise,} \end{cases} \tag{85}$$

where the radius of the interaction sphere is $r_1 > r_0$. By contrast, the Coulomb potential

$$V(r) = \frac{e^2}{r}, \tag{86}$$

where e is the electric charge, can almost never be truncated because although the force decays like r^{-2}, the number of particles between distance r and $r + dr$ generally increases like r^2 so effects of distant particles cannot be ignored.

In a purely Lagrangian representation of a system of particles with short-range interactions, only the coordinates of each particle need to be represented. The data structure might therefore consist of coordinates $\mathbf{x}_j$ where $j \in \{1, \ldots, N\}$, where N is the total number of particles. The force on a particle is computed by taking the vector sum of the forces due to all of the other particles in its interaction sphere. The challenge here is to find a way to compute this in a time that is linear in N, rather than quadratic. This ought to be possible, since, for a substance at a given average density and temperature there are only a fixed characteristic number of particles in an interaction sphere. If we denote this number by ν, then we need to compute on the order of νN forces. While this is linear in N, it is not obvious how to determine which other particles are in the interaction sphere of a given particle in a time that is linear in N. For example, everybody's first try at the problem,

```
DO i = 1, N
  DO j = 1, N
    IF(DISTANCE(i, j) < r1) FORCE(i) = FORCE(i) + GET_FORCE(i, j)
  END DO
END DO
```

(where the functionality of the variously called routines is obvious from their names), will decidedly *not* work. Even though the `IF` test is satisfied only on the order of ν times for each i, just the testing to see if the particles are further than a distance r_1 apart will take on the order of N^2 operations in the above code. Since the very largest MD simulations run today use hundreds of millions of particles, no portion of the algorithm may be of order N^2 complexity.

Fortunately, there are numerous well-researched parallel algorithms for rapidly determining and updating the list of neighbors of a given particle [Allen 1987]. Interestingly, most algorithms for determining neighbors employ an Eulerian grid of some sort. If the grid is regular, each particle may easily determine which grid cell it is in, and the particles may be sorted according to cell number. "Bookkeeping" problems arise due to particles that are near cell boundaries, but they are surmountable. Moreover, once a list of neighbors has been determined for each particle, this *neighbor table* may be updated incrementally, rather than recomputed from scratch at each time step. Using all of these various tricks (and more) on large parallel computers, MD simulations with between 10^8 and 10^9 particles, and with a characteristic performance of 10^{-8} seconds per particle update (the standard figure of merit for MD simulations) have been achieved [Zhou 1997].

6.2 Molecular Dynamics

The most straightforward type of MD simulation tracks the motion of a fixed number N of particles, constrained to move inside a fixed volume V. Periodic boundary conditions are often employed to emulate a large homogeneous sample of matter. Newton's equations for these particles,

$$\dot{\mathbf{r}}_i = \frac{\mathbf{p}_i}{m_i} \tag{87}$$

$$\dot{\mathbf{p}}_i = \sum_{j\neq i}^{N} \mathbf{F}(i \leftarrow j), \tag{88}$$

where $\mathbf{F}(i \leftarrow j)$ is the force on particle i due to particle j, derivable from a time-independent potential $V(\mathbf{r})$, will conserve the total energy

$$E = \sum_i^N \frac{p_i^2}{2m_i} + \frac{1}{2}\sum_{i\neq j} V\left(|\mathbf{r}_i - \mathbf{r}_j|\right). \tag{89}$$

The state of the system thus samples the microcanonical ensemble. The average *kinetic* energy, and hence the temperature will fluctuate. Typically, some desired property, such as a virial coefficient, a dielectric constant, or the shear viscosity, is time-averaged over the dynamics. For example, if the desired quantity is denoted by χ, the quantity

$$\langle\chi\rangle = \lim_{\tau\to\infty} \frac{1}{\tau}\int_0^\tau dt\ \chi(t) \tag{90}$$

is accumulated during the run. Its relationship to the true equilibrium value of this quantity thus rests on the assumption of ergodicity.

The first MD simulations were investigations of hard-disk fluids, carried out by Alder and Wainwright [Alder 1957] in the late 1950's. In a later paper [Alder 1967], these authors used such simulations to establish the existence of *long-time tails* in particles' velocity autocorrelation functions; that is, the quantity $\langle\mathbf{v}(0)\mathbf{v}(t)\rangle$, averaged over a large ensemble of particles, was found not to decay exponentially in time, as had been the common assumption, but rather to decay according to a power law. This unexpected result, of substantial physical import, was a great triumph of the MD method.

The number of particles that could be simulated at that time was in the tens or hundreds; as noted above, it is now in the hundreds of millions. Though this is undeniably a drastic improvement, it needs to be put into perspective: It is still only 10^{-15} moles of material. In terms of physical time, one time step of an MD simulation is on the order of 10^{-15} seconds. The very longest runs can carry out 10^7 time steps, for a total of about 10 nanoseconds of physical time. This is long enough for the energy to equipartition, and to compute many equilibrium properties of interest, transport coefficients, etc. Certain phenomena, however, such as protein folding and the dynamics of molecular clumps in glassy materials, occur on much longer time scales and MD has been inadequate for their investigation. This is particularly true of *soft condensed matter* applications, such as the development of micelles from dissolved surfactant, the response of liquid crystals to external fields, and the Brownian fluctuations of colloidal particles. Such *complex fluids* develop *mesoscale* structures – with characteristic lengths and times that are much larger than a mean-free path and a mean-free time, respectively – that dominate their macroscopic behavior.

Much work has been done over the last twenty years to accelerate the thermalization of molecular dynamics simulations. Two principal methodologies have been adopted for this purpose. One is to occasionally alter the particles' positions and/or momenta in stochastic fashion; the challenge here is to do this in a way that allows the particles to sample the phase space more rapidly than strict adherence to the Newtonian trajectories would allow, but to avoid doing such extensive damage to those trajectories that dynamic properties, such as time correlation functions and transport coefficients, are rendered meaningless. A number of recent methods adopt this strategy; these include, but are not limited to, Lattice-Gas Automata [Frisch 1986, Frisch 1987], Dissipative Particle Dynamics [Hoogerbrugge 1992, Espanol 1995], the Langevin method [Toral 1995], the Lattice Boltzmann Equation [Chen 1998], Direct-Simulation Monte Carlo [Bird 1994, Garcia 1994], Hybrid Monte Carlo [Heermann 1990] and Smoothed-Particle Hydrodynamics [Monaghan 1992]. The first of these will be discussed further in Subsection 6.5.

Another method of accelerating the calculation of certain quantities of interest is to use an ensemble other than the microcanonical one (constant E, V and N) described above. For example, Nosé and Hoover [Nosé 1986] have shown how to modify particulate equations of motion so that the particles are no longer constrained to the energy surface, but rather provide a sampling of the full canonical ensemble at specified temperature.

6.3 Long-range interactions

Long-range interactions are ubiquitous in physics. Charged particles in a plasma may interact according to the Coulomb potential. Interactions between stars in a galaxy, and indeed between galaxies in the expanding universe, may be modelled by the Newtonian gravitational potential. Somewhat less well known is the fact that particulate methods with long-range interactions may also be used to model incompressible fluid flow. For example, defining the *vorticity* of a fluid as the curl of its hydrodynamic velocity

$$\boldsymbol{\omega} \equiv \boldsymbol{\nabla} \times \mathbf{v}, \tag{91}$$

and noting that

$$0 = \boldsymbol{\nabla} \cdot \mathbf{v} \tag{92}$$

for an incompressible fluid, we recognize the similarity of these equations to those of magnetostatics: Velocity plays the role of the magnetic field $\mathbf{B}$ (whose divergence vanishes), and vorticity plays the role of the current density $\mathbf{J}$ (equal to the curl of the velocity). It follows that vortex filaments give rise to velocity fields according to the same Biot-Savart law that describes how current filaments give rise to magnetic fields. We can thus represent the fluid motion by vortex filaments in three dimensions (or point vortices in two dimensions), rather than the usual scheme of representing the velocity directly. Furthermore, if we take the curl of both sides of the Navier-Stokes equation for momentum conservation, Eq. (25), we can derive the equation of motion of these vortex filaments (or points). It turns out that each moves in the velocity field of all the other vortices present, and also has a random velocity component to model the viscous diffusion of momentum. To move the vortices at each step therefore requires summing the Biot-Savart law for the velocity at each one due to all of the others. Thus, the *vortex method* reduces the problem of incompressible fluid flow to that of long-range "particle" interactions [Gustafson 1991].

The most obvious way to sum up all of the forces is then something like

```
DO i = 1, N
  DO j = 1, N
    FORCE(i) = FORCE(i) + GET_FORCE(i, j)
  END DO
END DO
```

Clearly, this has order N^2 complexity, but this seems insurmountable since the forces are long range, and every particle really does interact with every other. In one of the more counterintuitive algorithmic results to be recently discovered in this field, it has been shown [Rokhlin 1987] that – for the special case of the Coulomb force – the required summation can be accomplished to machine precision with order N computational complexity! This *Fast Multipole Algorithm* makes use of the fact that forces due to distant particles can be approximated by a multipole expansion. It applies this observation in a recursive fashion: We first subdivide the spatial domain into eight parts using three mutually perpendicular planes in three dimensions (four parts using two perpendicular lines in two dimensions), and continue to subdivide the parts in the same way to obtain an *oct-tree* division of the spatial domain in three dimensions (*quad-tree* in two). The two-dimensional case is illustrated in Fig. 9. From its coordinates, it is straightforward for each particle to determine which "leaf" it occupies on such a tree, and the number of branchings is taken such that there is on the average of only a few particles per leaf. The algorithm begins by computing the multipole moments of each leaf. It then takes advantage of the fact that, for the Coulomb potential, multipole moments of "child" and "sibling" domains can be easily combined to get multipole moments of "parent" domains. Thus, it works its way to the root of the tree, computing the multipole moments of each division at each level. Next, it recursively works its way back to the leaves, using the multipole moments of distant "parent" domains to obtain their contributions to the potential in every "child" domain. When it arrives at the leaves, it has the potential (or the force) at each leaf due to all of the other sites on the finest grid. Finally, it adds the contribution due to the other particles in the same cell. Since there are 4^j domains at level j and $\log_4 N$ levels in the tree, the number of operations performed is on the order of

$$\sum_{j=0}^{\log_4 N} 4^j = \frac{4^{\log_4 N} - 1}{4 - 1} \sim N \tag{93}$$

for a fixed desired accuracy. In practice, this accuracy can be made greater than machine precision, resulting in an algorithm that is, for all intents and purposes, exact.

6.4 Particle-in-cell methods

Even with the Fast Multipole Algorithm, the number of Coulombic particles that may be simulated on a computer is far less than the corresponding number for short-range force laws. There is thus great interest in approximate methods with reduced computational complexity. An important example of such a method is the *particle-in-cell* algorithm. This is a hybrid Eulerian/Lagrangian technique, in which one keeps track of particle coordinates, but uses a grid to compute the fields by which these particles move.

As an example, let us consider a particle-in-cell simulation of a pure electron plasma confined by magnetic and electrostatic fields in a Penning trap [Chu 1993]. The plasma is confined in cylindrical geometry, with the magnetic field in the z direction. Each particle undergoes a very rapid gyration in the xy plane due to the magnetic field. Superposed on this is a bounce motion in the z direction due to the confining electrostatic potential. If we average over the cyclotron and bounce periods, we can consider the evolution of the density of gyration/bounce centers as these drift perpendicularly to the magnetic field,

$$\frac{\partial n(\mathbf{x}, t)}{\partial t} + \mathbf{v}_{\mathbf{E}\times\mathbf{B}} \cdot \boldsymbol{\nabla} n(\mathbf{x}, t) = 0. \tag{94}$$

Here we have defined the $\mathbf{E} \times \mathbf{B}$ drift velocity,

$$\mathbf{v}_{\mathbf{E}\times\mathbf{B}} \equiv c\frac{\mathbf{E}\times\mathbf{B}}{B^2} = \frac{c}{B}\hat{\mathbf{z}} \times \boldsymbol{\nabla}\Phi(\mathbf{x}, t), \tag{95}$$

which is a well-known systematic drift of charged particles confined in crossed electric and magnetic fields. In this case, the electric field is due to the plasma itself,

$$\nabla^2\Phi(\mathbf{x}, t) = -4\pi(-e)n(\mathbf{x}, t). \tag{96}$$

The boundary condition is that the potential must be zero at the grounded outside wall. This is illustrated in Fig. 10.

In a particle-in-cell simulation, we keep track of the position coordinates of each particle. Since we have averaged over the motion z direction, our problem is really two dimensional, so we keep two coordinates (e.g., r and θ) for each particle. At each time step we "smear" these particles onto a grid to get a charge density. This is done by the linear interpolation scheme illustrated in Fig. 11. Note that a polar coordinate grid is used. The charge is divided among the four gridpoints at the corners of its cell, and we then divide by the area of that cell to get the local charge density.

Once the charge density is known on the grid, we can solve the Poisson equation, Eq. (96), to get the potential on the grid. This may be done by Fourier transformation in the θ direction (since it is periodic), and by solving a tridiagonal system of linear equations in the r direction. By finite differencing this potential, we can get the electric field components on the grid. Using the same weighting factors that were used to spread the charge from the particles to the grid points, we can now interpolate to get the field at each particle's position. Each particle can then compute its local $\mathbf{E} \times \mathbf{B}$ drift velocity, and take a step along the characteristics of Eq. (94),

$$\frac{d\mathbf{x}}{dt} = \mathbf{v}_{\mathbf{E} \times \mathbf{B}}. \tag{97}$$

The resulting simulation technique is capable of following the evolution of the plasma as it breaks down into wispy filaments that would be very difficult to capture in a purely Eulerian or spectral representation. Fig. 12 illustrates a particle-in-cell simulation with $131,072$ particles moving on a spatial grid of dimension 256×256.

A wide body of literature is available on this simulation technique. A particularly nice review of plasma physics applications of the particle-in-cell method can be found in the book by Birdsall and Langdon [Birdsall 1985].

6.5 Lattice-Gas Hydrodynamics

In this subsection, we review a procedure for simulating the *nonequilibrium* behavior of viscous fluids. This technique, called *lattice-gas hydrodynamics*, deserves special mention because it introduces an entirely new paradigm for using the computer to simulate a physical system. Ironically, it is a return to von Neumann's *cellular automata* computational paradigm.

Hydrodynamic lattice gases take the same "minimalist" approach to fluids that the Ising model takes to ferromagnets. The object is not a precise model of the dynamics at the finest scales, but rather to invent a fictitious microdynamics whose *coarse-grained* behavior – in the thermodynamic limit – lies in the same universality class as the phenomenon under study.

At first glance, this approach is not unreasonable. Many different fluids found in nature obey the Navier-Stokes equations of viscous fluid dynamics; for example, oil and air both obey these equations in the proper regime, in spite of the fact that their molecular dynamics are radically different. Clearly there is something universal about the Navier-Stokes equations in the hydrodynamic limit. Indeed, kinetic theory tells us some of the reason for this: The *Chapman-Enskog* expansion in the Knudsen number, by which one derives the form of the hydrodynamic equations from the Boltzmann kinetic equation, yields the Navier-Stokes equations for a wide variety of fluids whose particles interact with short range forces, as long as those interactions conserve mass and momentum.

A lattice gas is then an attempt to define the simplest system of interacting particles on a lattice, whose collisions conserve mass and momentum, and thus whose coarse-grained behavior can be expected to obey the Navier-Stokes equations. We work on a triangular lattice [Frisch 1986] in two dimensions; since this has a coordination number of six, we take a dependent variable of six bits per site, reflecting the presence or absence of a particle moving along each of the six lattice vectors leaving the site. At each tick of the clock, the particles stream along their respective lattice vectors and collide with the other incoming particles at their new site. The collisions are required to conserve mass and momentum. The only nontrivial collisions for this two dimensional model are enumerated in Fig. (14).

By taking block space and/or time averages of the velocities of these particles, one can derive a local hydrodynamic velocity. Kinetic theoretical arguments can then be used to demonstrate that this velocity field satisfies the Navier-Stokes equations. Note that this approach is fundamentally different from a finite-difference approximation of the Navier-Stokes equations. The intent here is not to start with the continuum equations and discretize them, but rather to start with a fundamentally discrete system of particles, and make them interact in such a way that the Navier-Stokes equations are *emergent*, just as they are for natural fluids. That is, lattice-gas automata, like molecular dynamics, are a particulate *simulation* of the physical system under study, rather than the *numerical solution* of hydrodynamic partial differential equations. Because of this approach, LGA share (at least) two important properties of real fluids: First, they have absolutely *exact* conservation laws – down to the last bit in the simulation. Second, approximate methods for computing their transport coefficients (i.e., predicting their viscosity using the Boltzmann molecular chaos approximation) are often inaccurate. Consequently, practitioners of this method often have to measure such bulk properties in controlled flow situations, sometimes called *numerical experiments*, in order to calibrate runs for more interesting geometries or Reynolds numbers. This is, of course, exactly analogous to what is done by real experimentalists.

The triangular lattice is required for *isotropy* in two dimensions. Indeed, earlier versions of hydrodynamic lattice-gases were tried on Cartesian grids in the 1970's [Hardy 1973, Hardy 1976], and were demonstrably anisotropic. The method became feasible for computational fluid dynamics only when it was shown by Frisch, Hasslacher and Pomeau in 1986 [Frisch 1986] that going from a Cartesian to a triangular grid did not just improve anisotropy – it eliminated it altogether! The reason for this is tied up with the symmetry properties of the lattice, and beyond the scope of this article. Lattices with requisite symmetry exist in three dimensions [Frisch 1987].

It is possible to introduce the Monte Carlo sampling of collision outcomes into this method, in order to hasten the approach to a local thermodynamic equilibrium for systems of interacting particles. By introducing multiple particle species and different types of interactions and by Boltzmann-weighting collision outcomes, immiscible fluid flow [Rothman 1994], coexisting liquid and gas phases [Appert 1991], and amphiphilic fluids [Boghosian 1996] have all been modelled using this technique.

From a philosophical point of view, it is fascinating that the lattice-gas approach has effectively reduced hydrodynamics to logic. It simulates fluid motion by purely logical operations on bits – the kind of thing that a computer can do exactly, without approximation. Moreover, unlike floating-point representations of data, all of the bits present play essentially the same role in the simulation; this quality is sometimes referred to as "bit democracy," and vaguely indicates that the computer is being used in a more natural and efficient way than it is in a floating-point simulation. Indeed, the success of the lattice-gas paradigm has spurred more general studies of discrete particle mechanics on lattices. In particular, Baez and Gilliam [Baez 1994] have used ideas from algebraic geometry to generalize the methods of Lagrangian and symplectic mechanics to systems that are discrete in time, space and dependent variable – including cellular automata, but also including more general discrete systems. They show how to construct action principles and derive an analog of the Euler-Lagrange equations for such systems. Their methods hold the promise of an entirely new way to discretize physical problems for computer implementation, and it is likely that this will be an active area of research in the coming years.

7 Statistical Methods

The study of particulate methods naturally leads us to study the statistical methods of computational physics. It is well known that stochastic methods are often useful for the estimation of multidimensional integrals. For example, students in introductory statistics courses are taught to estimate the value of π by placing points (x, y) randomly in the square $-1 \leq x < +1$, $-1 \leq y < +1$, and determining what fraction of the time they lie within the unit circle $x^2 + y^2 < 1$. As the number of points used goes to infinity, this fraction will approach $\pi/4$. A Fortran 90 program to implement this procedure might be

```
REAL FUNCTION ESTIMATE_PI(n)

  IMPLICIT NONE
  INTEGER :: i, n, count
  REAL :: x, y

  count = 0

  DO i = 1, n
     CALL RANDOM_NUMBER(x)
     x = 2.0*(x - 0.5)
     CALL RANDOM_NUMBER(y)
     y = 2.0*(y - 0.5)
     IF(x**2 + y**2 < 1) count = count + 1
  END DO
```

```
  estimate_pi = 4.0*FLOAT(count)/FLOAT(n)

END FUNCTION ESTIMATE_PI
```

This procedure is estimating the multidimensional integral

$$\int_{-1}^{+1} dx \int_{-\sqrt{1-x^2}}^{+\sqrt{1-x^2}} dy. \tag{98}$$

Suppose that we had checked to see if the points were in the circle of radius 1/2, rather than the unit circle. Then the fraction of time that points were in the circle would have been $\pi/16$, and we could have multiplied by 16 to get π. This algorithm seems equivalent, but it is actually much worse. This is because the number of points that actually fall into the circle of radius 1/2 is much smaller (by a factor of 4) than the number which fall into the unit circle, thereby worsening the statistics. In fact, it is an elementary exercise to determine that the fractional error of the result is $\sqrt{\frac{1-p}{pn}}$, where n is the number of samples, and $p \equiv \pi R^2/4$ is the probability of the particle actually falling into the circle of radius $R \leq 1$. There is thus a strong incentive to increase p as much as possible, so the unit circle is preferable to that of radius 1/2. Indeed, if we could find a way to make $p = 1$, the fractional error would vanish altogether. We could, for example, try to use the inscribed circle of a pentagon, hexagon, etc., rather than a square. Unfortunately, it is much more difficult to efficiently sample points in these polygons than in a square, so there is a point of diminishing returns. This is, however, a special case of a more general rule: Monte Carlo methods work best when the samples can be taken from the problem's natural distribution function. We shall see this issue reappear in the following subsection.

Note that the standard deviation of the result for n points sampled goes as $n^{-1/2}$, so that something on the order of 10^8 points would be required to obtain π to four decimal places. While this may not be a particularly good way of obtaining the digits of π, it is often the only method available to estimate multidimensional integrals of the sort encountered in statistical physics. Recall that the equilibrium properties of a statistical mechanical system can be computed from the *partition function*,

$$Z = \sum_s e^{-\beta H(s)}, \tag{99}$$

where β is the inverse temperature, the sum extends over all possible states s of the system, and $H(s)$ is the energy of the system in state s. The challenge in computing Z is that the sum over states is typically huge. In fact, for systems with a continuum of states, this sum may be replaced by an integral, but in either case it is likely to be multidimensional so that the set of possible states s is enormous. The example of the Ising model, considered in the following subsection, illustrates this point.

7.1 Spin Systems

The *Ising model* is the prototype of a class of statistical physics models known as *spin systems*. It was developed by Lenz [Lenz 1920] in 1920 as a model of ferromagnetism, and posed as a problem to his student, Ising, who solved the one dimensional version of the model in 1925 [Ising 1925]. The model considers n "spins" described by $s_i \in \{-1, +1\}$ for $i = 1, \ldots, n$, on a lattice of D dimensions. In its simplest incarnation, the system's Hamiltonian has the form

$$H(s) = \frac{J}{2} \sum_{(i,j)} s_i s_j - B \sum_i s_i, \tag{100}$$

where the first sum is taken over all *neighboring pairs* of spins on the lattice. If b is the *coordination number*, the number of neighbors of any point on the lattice, then the first sum has bn terms; since this double counts the energy, a factor of 1/2 is included with the coupling constant J in front of this sum. The second sum models an external field of strength B. The problem is to find the partition function of the system,

$$Z = \sum_{s_1=-1}^{+1} \sum_{s_2=-1}^{+1} \cdots \sum_{s_n=-1}^{+1} e^{-\beta H(s)}, \tag{101}$$

as a function of βJ and βB. While Ising's $D = 1$ solution is elementary, Onsager's $D = 2$ solution for $B = 0$ required almost another twenty years [Onsager 1944], and is decidedly more complicated. The $D = 2$ solution for $B \neq 0$ is a relatively recent development, first published by Zamalodchikov [Zamalodchikov 1989] in 1989. The problem for $D = 3$ is outstanding, even for $B = 0$, and our knowledge of the behavior of Z in three dimensions rests almost entirely on numerical simulation.

At the root of the problem is the fact that the sum in Eq. (101) contains 2^n terms, and we would like to have the result in the *thermodynamic limit* as $n \to \infty$. While we may be able to perform this sum numerically for twenty spins (2^{20} terms in the summation) without much problem, it is clear that the problem is completely intractable if the spins number in the hundreds. Moreover, important equilibrium features, such as phase transitions, are sharply defined only in the thermodynamic limit. To see this, we compute the internal energy of the $D = 2$ Ising model by numerically summing Eq. (101) (and its derivative with respect to β) for 2×2, 3×3 and 4×4 lattices, and we plot these alongside Onsager's exact result in Fig. 13.

In analogy to our computation of π, we might suppose that we could *sample* the value of the internal energy by the following procedure:

1. Initialize the totals T_1 and T_2 to zero.
2. Perform the following loop N times:
   (a) Randomly choose a state s.
   (b) Compute the energy $H(s)$.
   (c) Increment T_1 by $e^{-\beta H(s)}$.
   (d) Increment T_2 by $H(s)e^{-\beta H(s)}$.
3. Estimate the energy by the ratio T_2/T_1.

This seems straightforward to implement, even for very large lattices, but it has a fatal problem: For all but the highest of temperatures, the vast majority of the total energy is accounted for by a miniscule fraction of the set of all possible states. Thus, if we sample this set *uniformly* in step 2(a) of the above algorithm, we will almost never encounter the most important states, and our variance will be unacceptably high.

This is reminiscent of the problem we had when we made the radius of the circle smaller in the algorithm for calculating π in the last subsection. Here, as there, the solution is to try to sample the states from the problem's natural distribution function. That is, we would ideally like to sample the states s with probability

$$f(s) = \frac{e^{-\beta H(s)}}{Z}. \tag{102}$$

Then we could simply add up the energies sampled, and divide by the total number of samples to obtain the average internal energy. It seems that we cannot do this directly, however, since the probability depends on the partition function Z, and that is, after all, one of the things that we are trying to use the Monte Carlo method to compute in the first place.

7.2 Metropolis Monte Carlo

The apparent impasse described in the last subsection was broken by an ingenious algorithm due to Metropolis *et al.* [Metropolis 1953] in 1953. Since then, a number of variants of this algorithm have been developed, but we describe the original one here.

The algorithm allows one to begin with a randomly selected state, but describes a Markov process for selecting all subsequent states. That is, if the system is currently in state s, the probability that this Markov process will transition it to a new state s' is called $K(s' \leftarrow s)$ and is specified by the algorithm in such a way that the time-asymptotics of the Markov chain will sample the system with some desired probability function $f(s)$. It accomplishes this by demanding that the Markovian dynamics obey the condition of *detailed balance*, which ensures that it will converge to a thermodynamic equilibrium with distribution f. This condition may be written

$$f(s)K(s' \leftarrow s) = f(s')K(s \leftarrow s'). \tag{103}$$

The Metropolis selection process occurs in two steps: In the first, the new *trial state* s' is selected with some probability $T(s' \leftarrow s)$. In the second, the trial state is accepted with probability $A(s' \leftarrow s)$; otherwise it is rejected and the system returns to the original state s. Thus, we may write

$$K(s' \leftarrow s) = T(s' \leftarrow s)A(s' \leftarrow s), \tag{104}$$

so the condition of detailed balance becomes

$$f(s)T(s' \leftarrow s)A(s' \leftarrow s) = f(s')T(s \leftarrow s')A(s \leftarrow s'), \tag{105}$$

which may in turn be written

$$\frac{A(s' \leftarrow s)}{A(s \leftarrow s')} = \frac{f(s')T(s \leftarrow s')}{f(s)T(s' \leftarrow s)} \equiv Q(s' \leftarrow s). \tag{106}$$

Note that the identity

$$Q(s' \leftarrow s) = \frac{1}{Q(s \leftarrow s')} \tag{107}$$

is manifest from the definition of Q. We would like to be able to specify the desired distribution f; this is usually a Gibbsian equilibrium distribution for statistical physics applications. We would also like to specify the probability T of choosing a trial state in some convenient fashion. Thus, we would like to regard Q as an independent variable, and demand that the acceptance probabilities satisfy the detailed-balance condition. To this end, we demand a functional relationship between the acceptance probability A and the independent variable Q; that is, we demand

$$A(s' \leftarrow s) = F(Q(s' \leftarrow s)) \tag{108}$$

for some function F. It immediately follows that

$$\frac{F(Q(s' \leftarrow s))}{F\left(\frac{1}{Q(s' \leftarrow s)}\right)} = Q(s' \leftarrow s), \tag{109}$$

or, since this should hold for any Q, we look for functions F that obey

$$F(z) = zF\left(\frac{1}{z}\right) \tag{110}$$

for all $z \geq 0$. Fortunately, there is no shortage of functions that obey this condition. The one most often used in simulations of the Metropolis algorithm is

$$F(z) = \min(1, z), \tag{111}$$

though

$$F(z) = \frac{z}{1+z} \tag{112}$$

also works, and there are many other possibilities as well.

We can now give a full statement of the algorithm for the sampling of a Gibbsian distribution, such as are encountered in statistical physics. The distribution is of the form of Eq. (102) for a system with Hamiltonian H and partition function Z. Since we do not know the latter, we cannot sample directly from this distribution. We can, however, sample a trial state from $T(s' \leftarrow s)$ and form the quantity Q described above, since this involves only ratios of the distribution f. In the simplest algorithms of this sort, the selection of a trial state is subject only to the restriction that

$$T(s' \leftarrow s) = T(s \leftarrow s'), \tag{113}$$

in which case

$$Q(s' \leftarrow s) = \frac{f(s')}{f(s)} = e^{-\beta[H(s')-H(s)]}. \tag{114}$$

The acceptance probability is then

$$A(s' \leftarrow s) = F(Q(s' \leftarrow s)) = \min\left(1, e^{-\beta[H(s')-H(s)]}\right). \tag{115}$$

The Markov process is then simple to describe: If one is in state s, and proposes to move to a new state s', one first computes the change in energy $\Delta H(s' \leftarrow s) = H(s') - H(s)$ associated with this move. If it is negative – so that the move is to a state with less energy – one accepts the move unconditionally. If, on the other hand, $\Delta H(s' \leftarrow s)$ is positive – so that the move is to a state with greater energy – one accepts this move only with a probability $\exp[-\beta\Delta H(s' \leftarrow s)]$. Alternatively stated, one generates a uniformly distributed random number $r \in [0,1]$, and accepts the move to state s' only if $r < \exp[-\beta\Delta H(s' \leftarrow s)]$ (since this will always be true if $\Delta H(s' \leftarrow s) < 0$). Note that for high temperatures this criterion is likely always to be accepted, so the sampling of states is really uniformly distributed; for low temperatures, on the other hand, the algorithm actively prevents one from visiting states with energies that are too high. In practice, since the randomly selected starting state is not likely to be particularly probable, one must "thermalize" this algorithm for some number of Markov steps before beginning to measure expectation values.

Using the Metropolis algorithm one can, for example, determine the equilibrium properties of a gas of classical molecules with arbitrary interaction potential. Recall that an analytic treatment of this problem results in an Ursell-Mayer cluster expansion [Reichl 1980]. A serial algorithm for this problem is easy to describe: Begin with a random configuration of molecules, and try to move them at random, one at a time. For each proposed move, calculate the change of energy that would result, and accept or reject that move according to the Metropolis criterion. After a while, the molecules will reach thermal equilibrium, and one can begin to take samples of any thermodynamic quantity of interest.

In recent years, a wide variety of improvements on this basic algorithm have been proposed. The most basic of these is the *biased sampling* of trial configurations, favoring those that are most consistent with the natural distribution function of the system under study – to the extent that it is known. This generally gives up the equality of $T(s' \leftarrow s)$ and $T(s \leftarrow s')$, but this can be done if their ratio is included in the Q factor that is used to determine acceptance, as indicated in Eq. (106).

The logical conclusion of the biased sampling procedure would be to bias the generation of new configurations toward the natural distribution function so successfully that every trial configuration is accepted with unit probability. Remarkably, this is sometimes possible. Swendsen and Wang [Swendsen 1987] showed how it could be done with the Ising and Potts models by sampling only those new configurations that differ from the original by the flipping of *clusters* of like spins. In this way they were able to dramatically reduce the problem of *critical slowing down* that plagues these methods for systems near a critical point. At criticality the correlation length tends to diverge, and so particles can be correlated with other particles that are arbitrarily far away. In this situation, the random flipping of individual spins is a "bull in a china shop" approach: Far more often than not, it badly damages the sensitive network of natural correlations, and results in the trial configuration being rejected. Since the particles can be correlated on all scales, the solution is to flip them in clusters at corresponding scales, and this is precisely what Swendsen and Wang did. The sampling of the clusters can be accomplished by laying down "bonds" between each site with a probability that depends on the energy of the bond and the temperature; sites connected by an unbroken network of bonds are then said to be in the same cluster. For the Ising model, for example, the probability of such a bond is zero is the spins are not aligned, and $1 - e^{-2\beta J}$ if they are. Once the clusters are determined, they are flipped with probability 1/2 (in the unmagnetized version of the model). All of this constitutes the sampling of a new configuration, which is then unconditionally accepted.

Subsequent work by Kandel and Domany [Kandel 1989] generalizes the Swendsen-Wang procedure by introducing a hierarchical cluster algorithm capable of dealing with a wide variety of Hamiltonians. More recently, Machta [Machta 1995] has developed an *invaded cluster* algorithm which is capable of sampling a critical system without prior knowledge of the critical temperature, and with no apparent critical slowing down.

7.3 Lattice Gauge Theory

There is a close similarity between statistical physics and quantum physics, since the Feynman path integral formulation of the latter subject tells us that the complex amplitude for a particle at point $\mathbf{x}_1$ at time t_1 to propagate to point $\mathbf{x}_2$ at time t_2 is

$$Z[(\mathbf{x}_2, t_2) \leftarrow (\mathbf{x}_1, t_1)] = \int d[\mathbf{x}(t)] \exp\left[i \int_{t_1}^{t_2} dt\ L(\mathbf{x}, \dot{\mathbf{x}})\right], \tag{116}$$

where the outer *path integral* is over all paths connecting the two points in spacetime, and L is the Lagrangian. Since a classical field $\psi(\mathbf{x}, t)$ is just a function of the spacetime coordinates, the analog of the above for a field is just

$$\mathcal{Z}[\psi_2(\mathbf{x}, t) \leftarrow \psi_1(\mathbf{x}, t)] = \int d[\psi(\mathbf{x}, t)] \exp\left[i \int dt \int d^3x\ \mathcal{L}(\psi, \boldsymbol{\nabla}\psi, \partial_t\psi)\right], \tag{117}$$

where $\mathcal{L}(\psi, \boldsymbol{\nabla}\psi, \partial_t\psi)$ is the Lagrangian density of the field, so $L = \int d^3\mathcal{L}$. One can then take expectation values of any (time-ordered) quantity by weighting it with the complex measure

$$\frac{1}{\mathcal{Z}[\psi_2(\mathbf{x}, t) \leftarrow \psi_1(\mathbf{x}, t)]} \exp\left[i \int dt \int d^3x\ \mathcal{L}(\psi, \boldsymbol{\nabla}\psi, \partial_t\psi)\right], \tag{118}$$

in exact analogy to Eq. (102). In fact, by analytically continuing the time coordinate to the complex plane, $t = ix_4$, we can write the above in the form

$$\frac{1}{\mathcal{Z}} \exp\left[-\int d^4x\ \mathcal{L}(\psi, \boldsymbol{\partial}\psi)\right], \tag{119}$$

where $\mathbf{x}$ is now understood to be a spacetime coordinate, and $\boldsymbol{\partial}$ is the four-gradient. The appearance of the manifestly covariant spacetime volume element d^4x makes this formulation natural for relativistic field theories.

By discretizing the Lagrangian density onto a lattice, we can extract vacuum expectation values of any quantity of interest using the Metropolis procedure. Just as we did when introducing the finite element method, we restrict the independent variable to a lattice $\mathbf{x}_\mathbf{j} = \mathbf{j}\Delta x$, where Δx is the lattice spacing. The four-gradient is then approximated by

$$\partial_\alpha\psi(\mathbf{x}_\mathbf{j}) \approx \frac{1}{\Delta x}\left[\psi(\mathbf{x}_{\mathbf{j}+\hat{\mathbf{e}}_\alpha}) - \psi(\mathbf{x}_\mathbf{j})\right]. \tag{120}$$

Gauge field theories, such as *quantum electrodynamics* (QED) or *quantum chromodynamics* (QCD), are associated with an underlying Lie group G; these are $U(1)$ for QED and $SU(3)$ for QCD. A set of group generators, denoted by λ^a, obey closed commutation relations,

$$[\lambda^a, \lambda^b] = if^{abc}\lambda^c, \tag{121}$$

where the f^{abc} are the structure constants of G (for this choice of generators). For example, if G is $SU(2)$, we could take the λ's to be (half) the Pauli matrices, in which case the structure constants are just the components of the Levi-Civita tensor $f^{abc} = \epsilon^{abc}$. An arbitrary element of the Lie algebra of G may then be expanded in terms of these generators; thus, the *gauge potentials*, which take their values in the Lie algebra, may be written

$$\mathbf{A}(\mathbf{x}) = \sum_a \mathbf{A}^a(\mathbf{x})\lambda^a. \tag{122}$$

Let us discretize the gauge field by storing $\mathbf{A}(\mathbf{x})$ at each lattice site. Exponentiation of the elements of the Lie algebra give elements of G, so that

$$U(\mathbf{j}, \alpha) \equiv \exp\left[\frac{ig\Delta x}{2} \sum_a \lambda_a \hat{\mathbf{e}}_\alpha \cdot \mathbf{A}^a(\mathbf{x}_\mathbf{j})\right], \tag{123}$$

is an element of G that can be *associated with the link* connecting lattice site $\mathbf{x}_\mathbf{j}$ with its neighboring site $\mathbf{x}_\mathbf{j} + \hat{\mathbf{e}}_\alpha$. Here g is the coupling constant of the theory. The Lie group elements associated with the links have the virtue that their product around any closed path on the spacetime lattice is gauge invariant. In terms of these quantities, Wilson [Wilson 1971] showed that the discretized action ($S \equiv \int d^4\, \mathcal{L}$) of quantum chromodynamics could be written

$$S = -\frac{1}{g^2} \sum_{\text{plaquettes}} \text{Trace}\,(U_p)\,, \tag{124}$$

where U_p is the product of the four elements of G around the edges of a closed unit square, or *plaquette*,

$$U_p = U(\mathbf{j}, \alpha)U(\mathbf{j} + \hat{\mathbf{e}}_\alpha, \beta)U(\mathbf{j} + \hat{\mathbf{e}}_\alpha + \hat{\mathbf{e}}_\beta, -\alpha)U(\mathbf{j} + \hat{\mathbf{e}}_\beta, -\beta), \tag{125}$$

and where the sum is taken over all such plaquettes on the spacetime lattice. Note that the U matrices are associated with *directed* links, so that

$$U(\mathbf{j} + \hat{\mathbf{e}}_\beta, -\beta) = U^\dagger(\mathbf{j}, \beta). \tag{126}$$

One then computes expectation values of operators O depending on the $\mathbf{U}$ matrices (any gauge-invariant operator can be thus expressed) by

$$\langle O(\mathbf{U})\rangle = \frac{1}{Z}\int d[\mathbf{U}]e^{-S}. \tag{127}$$

This is exactly the kind of quantity that we were computing in the last subsection, in the context of statistical physics, and can thus be estimated using the Metropolis procedure. Changing the gauge potential at any lattice point affects only those plaquettes that actually pass through the point, so that the change in the lattice action is a straightforward local calculation.

Because the coupling constant g of QCD is large, it is not useful as an expansion parameter, as it is in QED. Thus, many of the known consequences of QCD have been determined by numerical simulations of the sort described above. In particular, the form of the interquark potential has been estimated in this way, as have baryon masses, and the method has been used to investigate the elusive "glueball." A large literature exists on algorithms for lattice gauge theory, and on their supercomputer implementation; see [Dasgupta 1996] for details on the Fortran 90 implementation.

8 The Future of Computational Physics

8.1 The State of the Art

In the previous sections, we have traced the evolution of the relationship between computation and physics. By implementing many of the methods described on the largest supercomputers available today, extremely large computations have been performed. Some of the largest computations thus performed have been compiled into a list of "World Scientific Records" by Stauffer [Stauffer 1996], and we mention some highlights here:

- At the time of this writing, the largest $D = 2$ Ising simulations on record employ a $1,000,000 \times 1,000,000$ grid (10^{12} sites) [Linke 1995], and the largest with $D = 3$ employ a $3072 \times 3072 \times 3072$ grid (2.9×10^{10} sites) [Munkel 1993]. The fastest such simulations can perform Metropolis updates on 11 billion sites per second [Ito 1996].
- *Exact* solutions of the $D = 3$ Ising model, using computer-assisted symbolic manipulation methods for carrying out the algebra, have been accomplished for a $5 \times 5 \times 6$ lattice (150 spins) [Guttmann 1994]. (Note that a brute-force approach to this problem would involve the summation of 2^{150} terms – an impossible task.)
- As mentioned earlier, molecular dynamics simulations with between 10^8 and 10^9 particles, and with a characteristic performance of 10^{-8} seconds per particle update have been achieved [Zhou 1997] on large parallel supercomputers.
- Quantum chromodynamics simulations have achieved 7 billion updates per second [Sexton 1995].
- Floating-point fluid dynamics calculations in three dimensions have been performed on $512 \times 512 \times 512$ grids [Oran 1992].

8.2 Quantum Computation

As the above-mentioned doubling of computer performance every 1.5 years continues unabated at the time of this writing, one can expect the above performance records to continue to improve substantially. When it became clear that this could not continue forever (see Fig. (1)), computer manufacturers exploited parallelism so that the trend would continue. Most of the effort in the supercomputer industry today centers on making parallel clusters of computers on fast networks easy to use; one method of doing this is to develop efficient and elegant parallel programming paradigms, embodied in high-level languages, such as Fortran 90.

In spite of their success at warding off the problem for the time being, Fig. (1) still gives one the feeling that supercomputer manufacturers and users are heading for a "brick wall" in the not-too-distant future, and raises the question of whether or not any powerful new strategies for physical computation will come to the rescue, as did parallelism in the 1980's. In a development that has many a computer scientist scrambling to (re)learn quantum physics, one of the most interesting, exciting and potentially explosive new computational paradigms currently under consideration centers on the computational properties of quantum mechanical systems, and is called *quantum computation*.

Recall that all of classical computation theory, as prototyped by the Turing machine model, is carried out by logical operations on sequences of bits – quantities that take their values in the set $\{0, 1\}$. By contrast, the fundamental element of information in a quantum computer is the *qubit* (pronounced "Q-Bit") or *quantum bit*. Like its classical counterpart, the measurement of a single qubit can yield one of two results, $|0\rangle$ or $|1\rangle$. Unlike its classical counterpart, however, a qubit can be in a complex superposition of these two states,

$$c_0|0\rangle + c_1|1\rangle, \tag{128}$$

where the coeffients obey the normalization condition $|c_0|^2 + |c_1|^2 = 1$. To borrow from the parlance of quantum mechanics, when the qubit is measured, it "collapses" to a classical bit of value 0 with probability $|c_0|^2$, and of value 1 with probability $|c_1|^2$. As long as we do not measure the bit, however, its complex amplitude may combine and interfere with those of other qubits.

Just as a classical computer manipulates classical bits and and combines them according to various logical operations, called *gates*, a quantum computer manipulates qubits and combines them according to unitary operations, called *quantum gates*. We note that even a single qubit with a unary quantum gate is capable of behavior that is totally unfamiliar from the classical theory of computation. For example, suppose that the unary quantum gate is the application of the unitary operator

$$\frac{1}{\sqrt{2}}\begin{pmatrix} +1 & +1 \\ -1 & +1 \end{pmatrix} \tag{129}$$

in the representation of Eq. (128). If we begin with a qubit that is assuredly (perhaps because of a measurement) in state $|0\rangle$, one pass through this quantum gate will yield

$$\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle. \tag{130}$$

If we were to perform a measurement at this point, the result would be 0 or 1, each with probability 1/2. Thus, a random bit generator is trivially realized on a quantum computer, unlike the classical case where we had to take a great deal of care to deterministically generate sufficiently uncorrelated pseudorandom numbers.

The fact that a measurement of the qubit yields a random bit does *not* mean that the quantum gate has destroyed the information that was in the originally prepared qubit. Indeed, quantum gates can never destroy information precisely because they are unitary; in this respect they are analogous to the *reversible gates* of classical computation, described earlier in this article. Of course, the measurement process is nonunitary and would indeed destroy the information, but suppose that instead of measuring the outgoing bit, we were to pass it back throught the very same quantum gate one more time. The result would be $-|1\rangle$; that is, the result, if measured, would yield 1 with unit probability. Thus, two passes through the above unary quantum gate yield the classical "NOT operation, which is perfectly reversible. This has led one author [Hayes 1995] to refer to the above quantum gate as "the square root of NOT."

Inspired by oddities of this sort, physicists and computer scientists began to examine what computations might be like on such computers back in the 1980's [Feynman 1982, Feynman 1986, Deutsch 1985, Deutsch 1989]. A central, if somewhat ill-posed, hope was that quantum computers would somehow be capable of calculations completely inaccessible to classical computers. At first, it is rather difficult to see what is meant by this. Consider, however, the above-described computation of an actual *random* (as opposed to pseudorandom) bit; this is clearly something outside the purview of a classical computer. An even more striking example is provided by DiVincenzo's [Barenco 1995] observation that two-qubit quantum gates are sufficient to build a universal quantum computer; recall that three-bit gates were necessary to do this with classical computation.

The net result of these observations is that quantum computation is a new paradigm of computation – completely distinct from the Turing machine paradigm. Until the investigation of quantum computation, it was implicitly believed that the axiomatization of computation by the Turing machine paradigm was a unique mathematical construction. While Turing's ideas are often described with the cartoon of a "head" sliding along a "paper tape," this was considered to be nothing more than an intuitive picture to aid the understanding of an abstract mathematical concept. From a philosophical point of view, the observation that this is *not* true, and that the power of a computational paradigm does depend at least in part on its underlying physical realization, is just as profound as the nineteenth century observation that euclidean geometry is not a unique mathematical construction, and the early twentieth century discovery that the most natural geometry of the real world is in fact noneuclidean.

While these issues are undeniably interesting, they seemed to be of exclusively academic interest until Shor proved that the integer factoring problem, long suspected to be of exponential complexity on a classical computer, could be solved in polynomial time on a quantum computer [Shor 1994]. Though the details of Shor's demonstration lie outside the scope of this article, one might intuitively imagine that a problem can be subdivided into exponentially many subproblems, and these may be superposed and "parallel processed" owing to the linearity of quantum evolution. Shor showed that by selectively "interfering" the results in a certain way, it is possible to extract a desired final result in a time that is exponentially faster than the corresponding classical computation.

It is impossible to overstate the impact that such a result might have. As noted earlier, simulations of N quantum particles are necessarily of a classical complexity that is exponential in N. In analogy with Shor's result, it has been shown that such simulations are possible in polynomial time on a quantum computer, and an explicit quantum computational algorithm has even been developed for this purpose [Lloyd 1996, Boghosian 1997]. Indeed, the use of one quantum system to simulate another has a certain intuitive appeal, as it forcefully underscores the point that *all computation is physical*, and bodes for a future in which computation is regarded as the use of one physical system to emulate another.

Nowhere, however, is the result of Shor's observation more acutely felt than in the field of encryption. Though encryption algorithms are not commonly thought of as part of computational physics, and are therefore outside the scope of this article, suffice it to say that most all business and intelligence applications of encryption are predicated on the exponential difficulty of factoring large numbers. It is mostly for this reason that there has been an explosion of work in this area in the few years since Shor's paper appeared.

Given that a quantum computer is demonstrably able to do these things, one may wonder why one is not being built today. In fact, a number of groups are actively trying to do exactly this, but it is difficult. The difficulty lies in the great care that must be taken to preserve the unitary evolution of the qubits involved. If the environment of the computer performs any "measurement" whatsoever on the qubits before they have completed their unitary evolution, they will collapse in a nonunitary fashion, and results that depend on their interference will no longer follow. In other words, the qubits must be kept coherent for the duration of the computation. This consideration sets upper bounds on the length of a computation that may be performed on a quantum computer, and hence the number of cycles in any algorithm that may be carried out. This bound is analogous to that of sensitivity to errors in trajectories such as we encountered in our earlier discussion of classical, reversible "billiard-ball" computers; since that problem had a solution, we might expect that this one does as well. Indeed, within the past year a number of interesting results have emerged on *quantum error correction* [Shor 1996], and it has been shown that the reconstruction of a *damaged* qubit is possible in principle. It is hoped that this will provide some relief from the coherence requirements. In the meantime, researchers are trying to use a number of different physical systems to realize quantum computation [Turchette 1995].

9 Conclusion

We have surveyed the history and speculated about the future of the relationship between physics and computation. Though it is impossible to achieve completeness in any general survey of this sort, our intent has been to focus on examples that illustrate the principal issues involved, with emphasis on recent developments. We have surveyed physical computation, computational complexity, algorithms, languages, applications of computation to physics, and applications of physics to computation. Like any other important branch of physics, we have found that computational physics is interesting on many different levels; like any of the great revolutions of physics, we see that some of its most significant challenges and opportunities lie in questioning foundations and tenets long imagined to be inviolable.

Acknowledgements

The author would like to thank Peter V. Coveney, Norman H. Margolus, Claudio Rebbi and Francis Alexander for their careful critical reading of the draft version of this article, and for the valuable comments they provided.

A Glossary

- **Abacus:** Mechanical computing aid developed in Middle East at least 2500 years ago in which numerical data is represented by beads sliding on wires.
- **Aiken, Howard:** Harvard University professor who developed the Mark I computer just before and during World War II.
- **Analytical Engine:** The first programmable general-purpose mechanical digital computer, designed by Charles Babbage in England in the mid-nineteenth century.
- **Banded Matrices:** Sparse matrices whose elements vanish except on certain diagonals, and which arise naturally from finite-difference problems.
- **Bit:** Variable taking its value from a set of two elements, usually denoted $\{0, 1\}$ or $\{\texttt{false}, \texttt{true}\}$. The smallest unit of digital information.
- **Byte:** Eight bits.
- **Babbage, Charles:** Nineteenth century British mathematician who designed the first programmable, general-purpose mechanical digital computer, called the Analytic Engine.
- **Cellular Automaton:** Dynamical system that is discrete in space, in time, and in its dependent variable.
- **Church, Alonzo:** Twentieth-century American logician who worked on the early theory of computation.
- **Church-Turing Hypothesis:** Hypothesis, named after Alan Turing and Alonzo Church, that any universal computer can simulate and be simulated by a Turing machine.
- **Courant condition:** Condition for numerical stability when integrating partial differential equations, usually imposed as an upper bound on the time step.
- **Digital:** The representation of numbers as $\sum_j d_j B^j$, where $d_j \in \{0, \ldots, B-1\}$. Here B is called the *base* of the representation.
- **Dissipative particle dynamics (DPD):** A particulate method for accelerating molecular dynamics simulations, using idealized collisions that conserve momentum exactly, but letting the energy fluctuate within the canonical ensemble at some fixed temperature. DPD is similar to the Nosé-Hoover method, but conserves momentum.
- **Finite-difference:** Method of integrating partial differential equations in which classical fields are discretized on a grid and partial derivatives are represented by differences of neighboring values.
- **Finite-element:** Method of integrating partial differential equations in which classical fields are represented as sums of localized spline functions.
- **Floating-point numbers:** The computer representation of a real number by some number of bits of mantissa and exponent and by a sign bit. The format for this representation has been standardized by the IEEE.
- **Galerkin approximation:** Approximation at the heart of many finite-element methods that demands that the error incurred must lie outside of the space of basis functions used to represent the dependent variable.
- **Game of Life:** Two-dimensional cellular automaton with one bit per site and simple nearest-neighbor rules, invented by John Horton Conway in the early 1970's, that is provably capable of universal computation.
- **Gödel, Kurt:** Logician who discovered Gödel's Incompleteness Theorem in the 1930's.

- **Gödel's Incompleteness Theorem:** Theorem stating that any axiomatic system capable of representing truths of number theory is necessarily *incomplete*, in that it contains truths which cannot be proven from its axioms.
- **Halting Problem:** "Computer science version" of the Gödel Incompleteness Theorem which states that there is no a priori way of knowing whether or not a given computer program is a legitimate algorithm, in that it is guaranteed to produce results and halt.
- **Ising Model:** Prototypical statistical spin system defined on a grid at each site of which there is one bit of data, corresponding to a spin pointing either up or down. The system Hamiltonian is defined to be quadratic in the products of neighboring spin states.
- **Lattice Boltzmann method:** Similar to lattice-gas automata, but keeps track of discrete-velocity, single-particle distribution function only, and lacks kinetic fluctuations. The distribution is represented by a floating-point number for each of the velocities present in the model.
- **Lattice gas automata:** Particulate hydrodynamic method in which particles are represented by individual bits moving about on a lattice, and suffering collisions that maintain exact mass and momentum conservation. Can be used as an alternative to molecular dynamics for the study of universal properties that depend on the presence – but not the details – of an underlying kinetic model.
- **Lattice gauge theory:** Quantum gauge field theory on a lattice. Provides a Monte Carlo algorithm for sampling the expectation values of such theories.
- **Lovlace, Ada Augusta:** First computer programmer, and contemporary and coworker of Charles Babbage.
- **Mark I:** Electronic computer developed by Howard Aiken of Harvard University during World War II.
- **Metropolis method:** Markov chain method for the sampling of a large statistical ensemble without prior knowledge of its partition function.
- **Molecular dynamics:** The simulation of the dynamics of molecules to discern the bulk properties of materials.
- **Monte Carlo:** Computational method that uses random numbers to statistically estimate a desired quantity.
- **Napier's bones:** Seventeenth-century computational aid that used logarithmically spaced divisions to multiply. Predecessor of the slide rule.
- **Nosé-Hoover Method:** A particulate method for accelerating molecular dynamics simulations that lets particles explore the canonical, rather than microcanonical, ensemble.
- **Operator splitting:** Method of integrating partial differential equations by splitting the time-development operator into several pieces that can be treated separately.
- **Parallel computation:** The coordinated use of many processors to share the work of and thereby accelerate a calculation.
- **Particle-in-cell method:** Hybrid between grid and particle method, whereby individual particles move about according to self-consistent fields that are computed on a spatial grid.
- **Qubits:** Elemental data of quantum computation that take their values in the Hilbert space spanned by the various classical states that the system can have. A single qubit is thus a complex superposition of $|\texttt{true}\rangle$ and $|\texttt{false}\rangle$.
- **Quantum computation:** Computational paradigm that is an alternative to the Turing machine model, and that involves the unitary manipulation of *qubits*.

- **Slide rule:** Computational aid that uses logarithmically spaced divisions to multiply and divide. In widespread use until the 1970's.
- **Sparse matrices:** An $N \times N$ matrix with $\mathcal{O}(N)$ nonvanishing elements.
- **Spin systems:** Statistical models, usually consisting of a lattice of "spins," each interacting with their neighbors according to some specified Hamiltonian. Closely related to cellular automata.
- **Symplectic structure:** Structure of the equations of classical mechanics that ensures the preservation of the invariant integrals of Poincaré, and an action principle based on a Lagrangian function. Invertible symplectic structures also give rise to a Poisson-bracket structure of the equations of motion. In recent years, numerical integration techniques have been developed to preserve the symplectic structure of the equations of motion, with the philosophy that this is probably more important than conserving other quantities such as the energy.
- **Theory of Computation:** The theory of what computers can and cannot do, from both mathematical and physical considerations.
- **Turing, Alan:** Twentieth-century British mathematician who provided the first rigorous mathematical description of the computational process.
- **Turing machine:** Mathematical axiomatization of the computational process, developed by Alan Turing in the 1930's, and picturesquely described by a "machine" sliding along a "paper tape" imprinted with various symbols, and taking actions based on the values of those symbols.
- **Universal computation:** The capability to simulate any Turing machine.
- **Von Neumann, John:** Twentieth-century mathematician who studied the theory of computation, invented the principal computer architecture used to this day, and worked on the theory of cellular automata.

B List of Works Cited

References

[Alder 1957] Alder, B.J., Wainwright, T.E. (1957), J. Chem. Phys. **27**, 1208.

[Alder 1967] Alder, B.J., Wainwright, T.E. (1967), Phys. Rev. Lett. **18**, 988.

[Allen 1987] Allen, M.P., Tildesley, D.J. (1987), Computer Simulation of Liquids, Clarendon Press, Oxford.

[Analog] Note that *analog computers*, such as analog recurrent neural nets, are often said to be super-Turing; for the moment, however, we limit our discussion to digital computing paradigms.

[Appert 1991] Appert, C., Rothman, D.H., Zaleski, S. (1991), Physica D **47**, 85 (1991); Chen, H., Doolen, G.D., Lee, Y.C., Rose, H. (1989), Phys. Rev. A **40**, 2850.

[Baez 1994] Baez, J.C., Gilliam, J.W. (1994), Lett. Math. Phys. **31**, 205-212.

[Barenco 1995] See Barenco, A., Bennett, C.H., Cleve, R., DiVincenzo, D., Margolus, N.H., Shor, P., Sleator, T., Smolin, J., Weinfurter, H. (1995), Phys. Rev. A **52**, 3457-3467.

[Belinfante 1972] See, e.g., Belinfante, J.G., Kolman, B. (1972), A Survey of Lie Groups and Lie Algebras with Applications and Computational Methods, Society for Industrial and Applied Mathematics.

[Benioff 1982] Benioff, P. (1982), Int. J. Theor. Phys. **21**, 177; J. Stat. Phys. **29**, 515; Phys. Rev. Lett. **48**, 1581.

[Bennett 1973] Bennett, C. (1973), IBM J. Res. Develop. **6**, 525-532.

[Bennett 1982] Bennett, C. (1982), Int. J. Theor. Phys. **21**, 905-940.

[Bennett 1987] Bennett, C. (1987), Sci. Am. 108 (November, 1987).

[Bird 1994] Bird, G. A. (1994), Molecular Gas Dynamics and the Direct Simulation of Gas Flows (Clarendon, Oxford).

[Birdsall 1985] Birdsall, C.K., Langdon, A.B. (1985), Plasma Physics via Computer Simulation, McGraw-Hill.

[Boghosian 1996] Boghosian, B.M., Coveney, P., Emerton, A. (1996), Proc. Roy. Soc. A **452**, 1221.

[Boghosian 1997] Boghosian, B.M., Taylor, W. (1997), to appear in Phys. Rev. E.

[Cannon 1969] Cannon, L.E. (1969), A Cellular Computer to Implement the Kalman Filter Algorithm, PhD Thesis, Montana State University.

[Channell 1990] Channell, P.J., Scovel, C. (1990), Nonlinearity **3**, 231-259.

[Chen 1998] Chen, S. (1998), in Annual Review of Fluid Mechanics **30**, Annual Reviews, Inc., Palo Alto, California. (To appear in 1998.)

[Chu 1993] Chu, R.Y-Y. (1993), Theoretical Studies of Pure Electron Plasmas in Asymmetric Traps, Ph.D. Thesis, Department of Physics, Massachusetts Institute of Technology.

[Cooley 1965] Cooley, J.W., Tukey, J.W. (1965), Mathematics of Computation, **19**, 297.

[Dasgupta 1996] Dasgupta, I., Levi, A.R., Lubicz, V. and Rebbi, C. (1996), Comp. Phys. Comm. **98**, 365.

[Daubechies 1988] Daubechies, I. (1988), Comm. Pure Appl. Math. **XLI**, 909.

[DeRaedt 1987] DeRaedt, H. (1987), Comput. Phys. Rep. **7**, 1.

[Deutsch 1985] Deutsch, D. (1985), Proc. Roy. Soc. London **400A**, 96-117.

[Deutsch 1989] Deutsch, D. (1989), Proc. Roy. Soc. London **425A**, 73-90.

[Espanol 1995] Espanol, P. (1995), Phys. Rev. E **52**, 1734-1742.

[Feynman 1982] Feynman, R. (1982), Int. J. Theor. Phys. **21**, 467-488.

[Feynman 1986] Feynman, R. (1986), Found. Phys. **16**, 507-531.

[Fredkin 1982] Fredkin, E., Toffoli, T. (1982), Int. J. Theor. Phys. **21** 219-253.

[Frisch 1986] Frisch, U., Hasslacher, B., Pomeau, Y. (1986), Phys. Rev. Lett. **56**, 1505.

[Frisch 1987] Frisch, U., d'Humières, D., Hasslacher, B., Lallemand, P., Pomeau, Y., Rivet, J.-P. (1987), Complex Systems **1**, 75.

[Garcia 1994] Garcia, A.L. (1994), Numerical Methods for Physics, (Prentice Hall, Englewood Cliffs).

[Gustafson 1991] See, e.g., Gustafson, K.E., Sethian, J.A. (1991), eds., Vortex Methods and Vortex Motion, (Society for Industrial and Applied Mathematics, Philadelphia).

[Guttmann 1994] Guttmann, A.J., Enting, I.G. (1994), J. Phys. A **27**, 8007.

[Hardy 1973] Hardy, J., de Pazzis, O., Pomeau, Y. (1973), J. Math. Phys. **14**, 1746.

[Hardy 1976] Hardy, J., de Pazzis, O., Pomeau, Y. (1976), Phys. Rev. A **13**, 1949.

[Hayes 1995] Hayes, B. (1995), American Scientist **83**, 304-308 (July/August 1995).

[Heermann 1990] Heermann, D.W., Nielaba, P., Rovere, M. (1990), Comp. Phys. Comm. **60**, 311.

[Hillis 1985] Hillis, D.W. (1985), The Connection Machine, M.I.T. Press, Cambridge, MA.

[Hillis 1993] Hillis, D.W., Boghosian, B.M. (1993), Science **261**, 856.

[Hoogerbrugge 1992] Hoogerbrugge, P.J., Koelman, J.M.V.A. (1992), Europhys. Lett. **19**, 155-160.

[Ising 1925] Ising, E. (1925), Z. der Physik **31**, 253.

[Ito 1996] Ito, N. (1996), Int. J. Mod. Phys. C **7**, 99.

[Kahan 1996] Kahan, W. (1996), Beastly Numbers, and A Test for Correctly Rounded SQRT, Technical Reports, Department of Electrical Engineering and Computer Science, University of California, Berkeley.

[Kandel 1989] Kandel, D., Domany, E. (1991), Phys. Rev. B **43**, 8539.

[Kardar 1986] Kardar, M., Parisi, G., Zhang, Y.-C. (1986), Phys. Rev. Lett. **56**, 889.

[Khuller 1991] See, for example, Khuller, S., Mitchell, S.G., Vazirani, V.V. (1991) in Proceedings of the 18th International Colloquium on Automata, Languages and Programming (Lecture Notes in Computer Science 510), Springer-Verlag, Berlin, 728-738.

[Landau 1982] Landau, L.D., Lifschitz, E.M. (1982), Fluid Mechanics, Pergamon Press.

[Landauer 1961] Landauer, R. (1961), IBM J. Res. Devel. bf 5 183-191.

[Lenz 1920] Lenz, W. (1920), Phys. Zeitschrift **21**, 613.

[Linke 1995] Linke, A., Heermann, D.W., Altevogt, P., Siegert, M. (1995), Physica **A222**, 205.

[Lloyd 1996] Lloyd, S. (1996), Science **273**, 1073-1078.

[Machta 1995] Machta, J., Choi, Y.S., Lucke, A., Schweizer, T., Chayes, L.V. (1995), Phys. Rev. Lett. **75**, 2792.

[Margolus 1984] Margolus, N. (1984), Physica **10D** 81-95.

[Metropolis 1953] Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A., Teller, E. (1953), J. Chem Phys. **21**, 1087.

[Miller 1992] Miller, J., Weichman, P.B., Cross, M.C. (1992), Phys. Rev. A **45**, 2328, Appendix A.

[Monaghan 1992] Monaghan, J.J. (1992), Annual Review of Astronomy and Astrophysics **30**, 543-574.

[Moore 1990] Moore, C. (1990), Phys. Rev. Lett. **64**, 2354.

[Moore 1991] Moore, C. (1991), Nonlinearity **4**, 199.

[Munkel 1993] Munkel, C. *et al.* (1993), Physica **A193**, 540.

[Nosé 1986] Nosé, S. (1986), Mol. Phys. **57**, 187-191.

[Onsager 1944] Onsager, L. (1944), Phys. Rev. **65**, 117.

[Oran 1992] Oran, E.S., Boris, J.P. (1992), Computers in Physics **7**, 523.

[PourEl 1982] Pour-El, M.B., Richards, I. (1982), Int. J. Theor. Phys. **21**, 553.

[Reichl 1980] See, for example, Reichl, L.E. (1980), A Modern Course in Statistical Physics, University of Texas Press, Chapter 11.

[Richardson 1991] Richardson, J.L. (1991), Comput. Phys. Comm. **63**, 84-94.

[Rokhlin 1987] Rokhlin, V., Greengard, L. (1987), J. Comp. Phys. **73**, 325-348.

[Rothman 1994] Rothman, D.H. and Zaleski, S. (1994), Rev. Mod. Phys. **66**, 1412.

[Schwartzman 1996] Schwartzman, S. (1996), Quantum **6**, 4.

[Sexton 1995] Sexton, J., Vaccarino, A., Weingarten, D. (1995), Phys. Rev. Lett. **75**, 4563.

[Shor 1994] Shor, P.W. (1994) in Proc. 35th Annual Symposium on the Foundations of Computer Science, ed. Goldwasser, S., IEEE Computer Society Press, Los Alamitos, California (1994) 124.

[Shor 1996] Shor, P.W. (1996), Fault-tolerant quantum computation, AT&T preprint, `quant-ph/9605011`, and references therein.

[Stauffer 1996] Stauffer, D. (1996), Int. J. Mod. Phys. C **7**, 759.

[Strang 1989] Strang, G. (1989), SIAM Review **31**, 614-627.

[Swendsen 1987] Swendsen, R.H., Wang, J.-S. (1987), Phys. Rev. Lett. **58**, 86-88.

[Symplectic] That is, the time rate of change of any phase-space function F is given by $\{F, H\}$ where H is a Hamiltonian, and $\{\cdot,\cdot\}$ is a Poisson bracket that is bilinear, antisymmetric and obeys the Jacobi identity.

[Toral 1995] See, e.g., Toral, R. (1995) in Third Granada Lectures in Computational Physics: Proceedings of the III Granada Seminar on Computational Physics, held at Granada, Spain, 5-10 September 1994, Garrido, P.L., Marro, J., eds., (Springer-Verlag, Berlin, New York, 1995).

[Turchette 1995] See, e.g., Turchette, Q.A., Hood, C.J., Lange, W., Mabuchi, H., Kimble, H.J. (1995), Phys. Rev. Lett. **75** 4710; Mattle, K., Weinfurter, H., Kwiat, P.G., Zeilinger, A. (1996), Phys. Rev. Lett. **76**, 4656-4659; Cory, D., Fahmy, A., Havel, T. (1997), Proc. Nat. Acad. Sci. USA **94**, 1634; Wineland, D.J., Monroe, C., Meekhof, D.M., King, B.E., Leibfried, D., Itano, W.M., Bergquist, J.C., Berkeland, D., Bollinger, J.J., Miller, J. (1997), to appear in Proc. Roy. Soc. A; also see references therein.

[Wilson 1971] Wilson, K. (1971), Phys. Rev. B **4**, 3174; Phys. Rev. B **4**, 3184.

[Zamalodchikov 1989] Zamalodchikov, A.B. (1989), Int. J. Mod. Phys. **A4**, 4235.

[Zhou 1997] Zhou, S.J., Beazley, D.M., Lomdahl, P.S., Holian, B.L. (1997), Phys. Rev. Lett. **78**, 479-82; also see review of this article in Marder, M. (1997), Nature **386**, 219-220.

C Further Reading List

1. Allen, M.P., Tildesley, D.J. (1987), Computer Simulation of Liquids, Clarendon Press, Oxford.
2. Bennett, C.H. (1995), Quantum information and computation in Physics Today, October 1995.
3. Birdsall, C.K., Langdon, A.B. (1985) Plasma Physics via Computer Simulation, McGraw-Hill.
4. Coveney, P.V., Highfield, R. (1995), Frontiers of Complexity, Fawcett Columbine, New York.
5. Frenkel, D., Smit, B. (1996), Understanding Molecular Simulation, From Algorithms to Applications, Academic Press.
6. Kenway, R.D., Pawley, G.S. (1987), Computational Physics, Proceedings of the Thirty-Second Scottish Universities' Summer School in Physics, Published by the Scottish Universities' Summer School in Physics.

D Figures

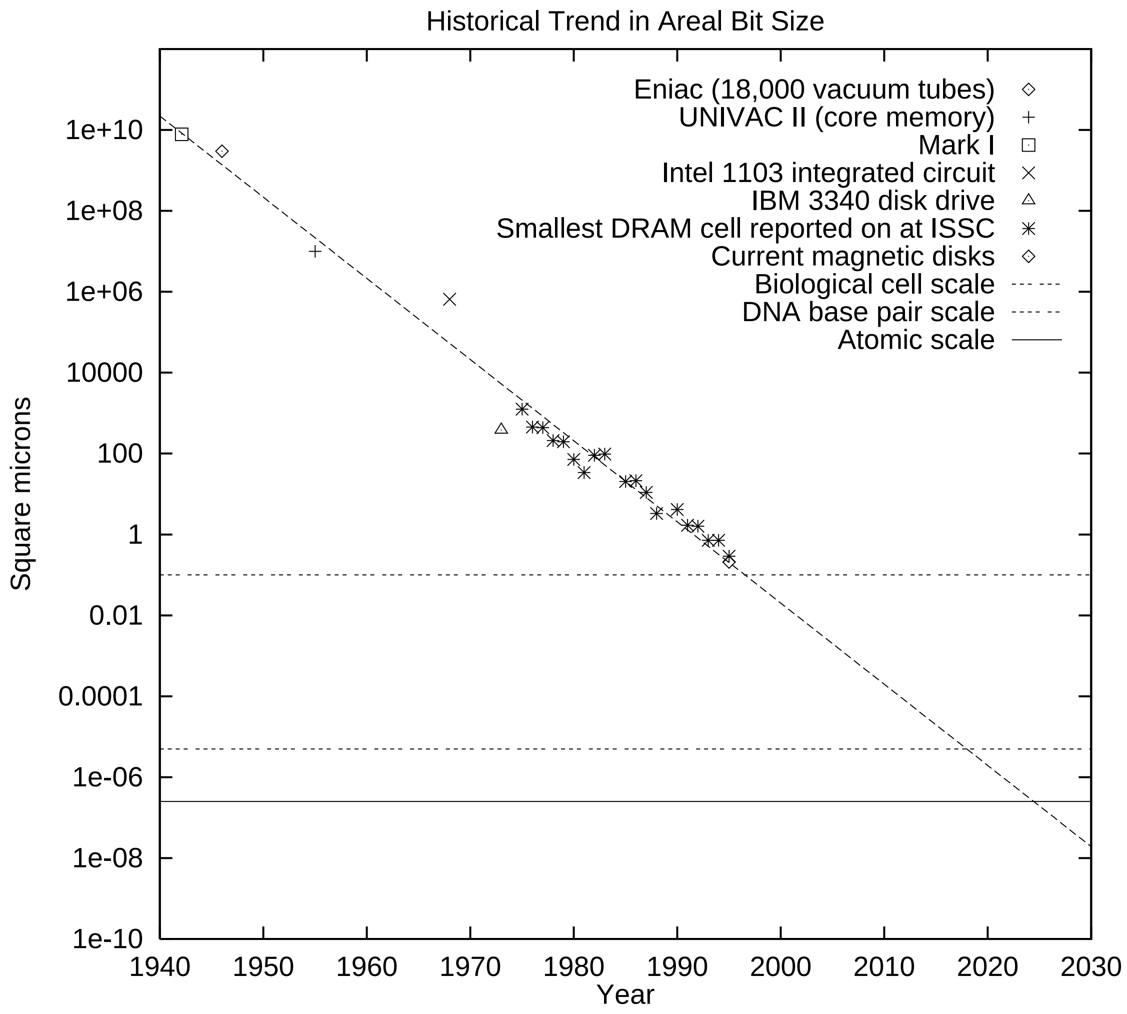

Figure 1: **Areal size required to store one bit of information as a function of time** (Courtesy of Norman Margolus, M.I.T. Laboratory for Computer Science)

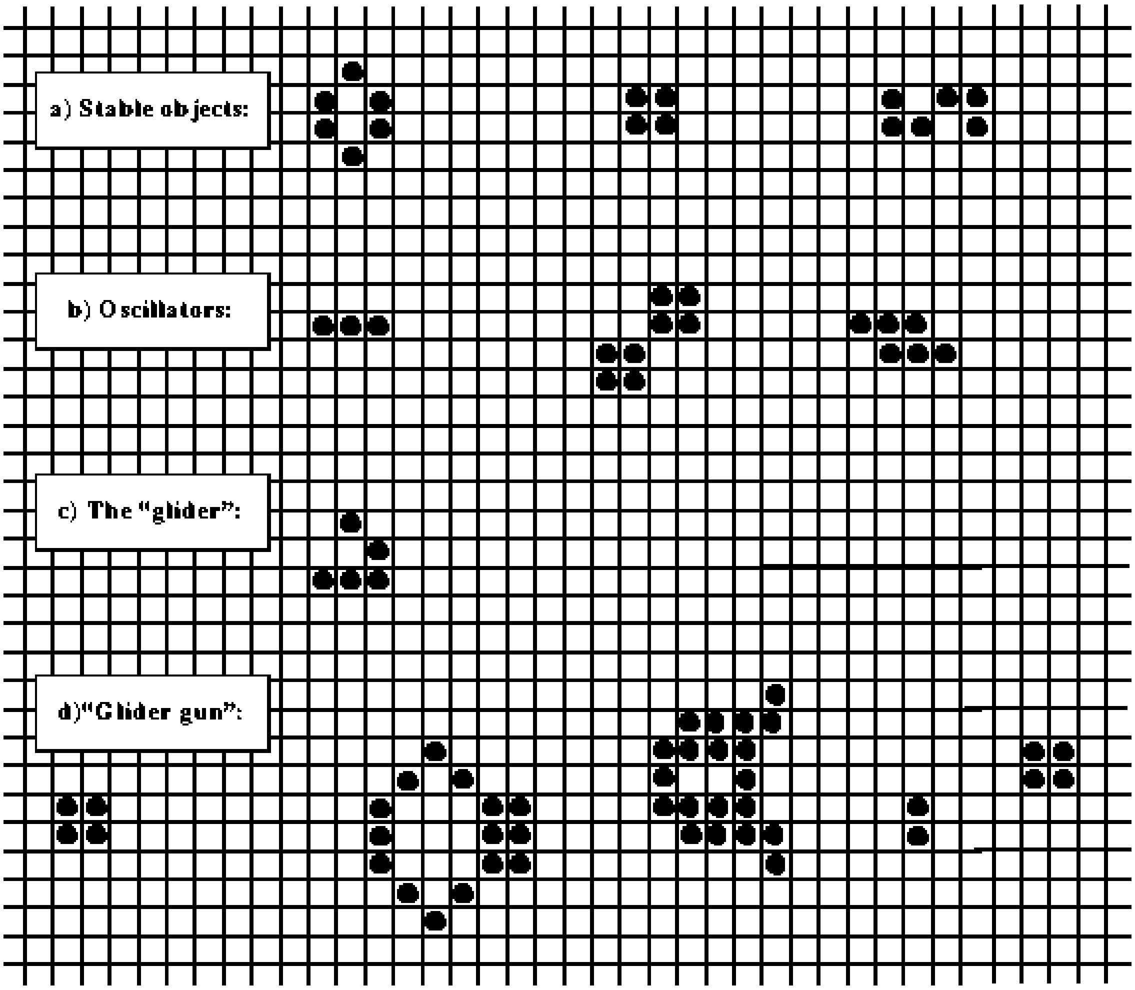

Figure 2: **Coherent structures of the Game of Life:** (a) stable patterns, (b) oscillatory patterns, (c) translating patterns, (d) the "glider gun"

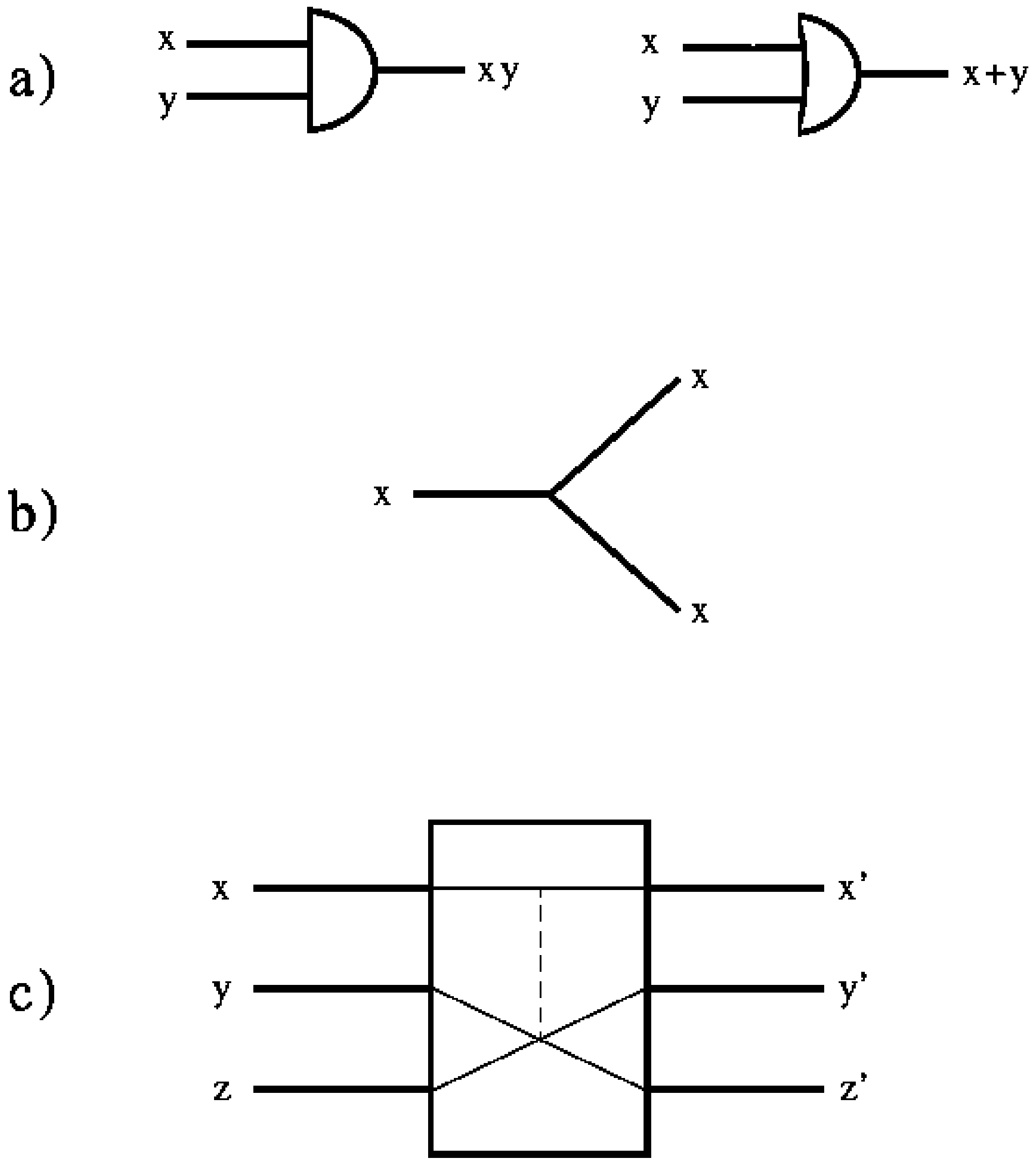

Figure 3: **Irreversible and reversible logic gates:** (a) "AND" and "OR" gates (arithmetic shown is understood to be boolean), (b) "FANOUT" gate, and (c) reversible, universal gate.

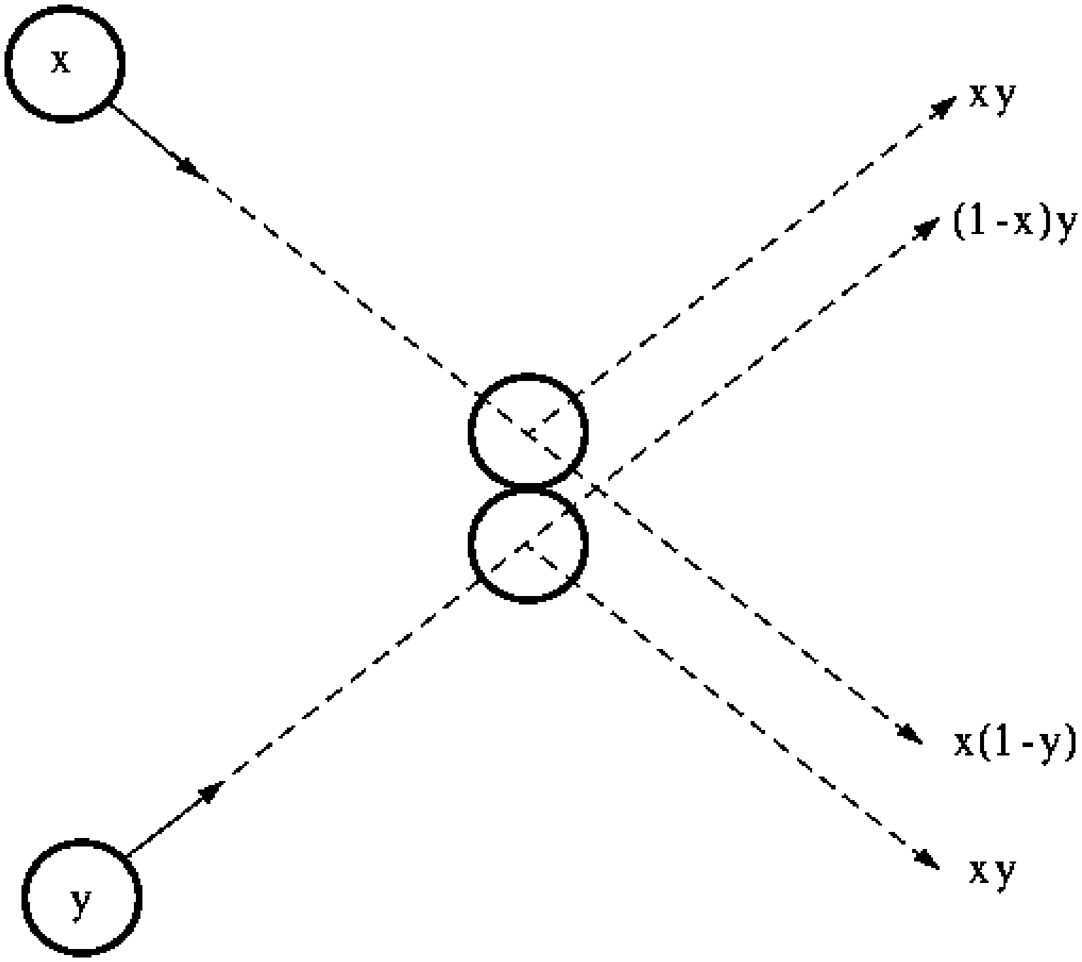

Figure 4: **Billiard ball model of computation**: The presence or absence of the two balls encodes the bits x and y, respectively. Note that the outgoing, post-collision trajectories yield the logical functions xy, $x(1-y)$ and $(1-x)y$ (arithmetic is understood to be boolean).

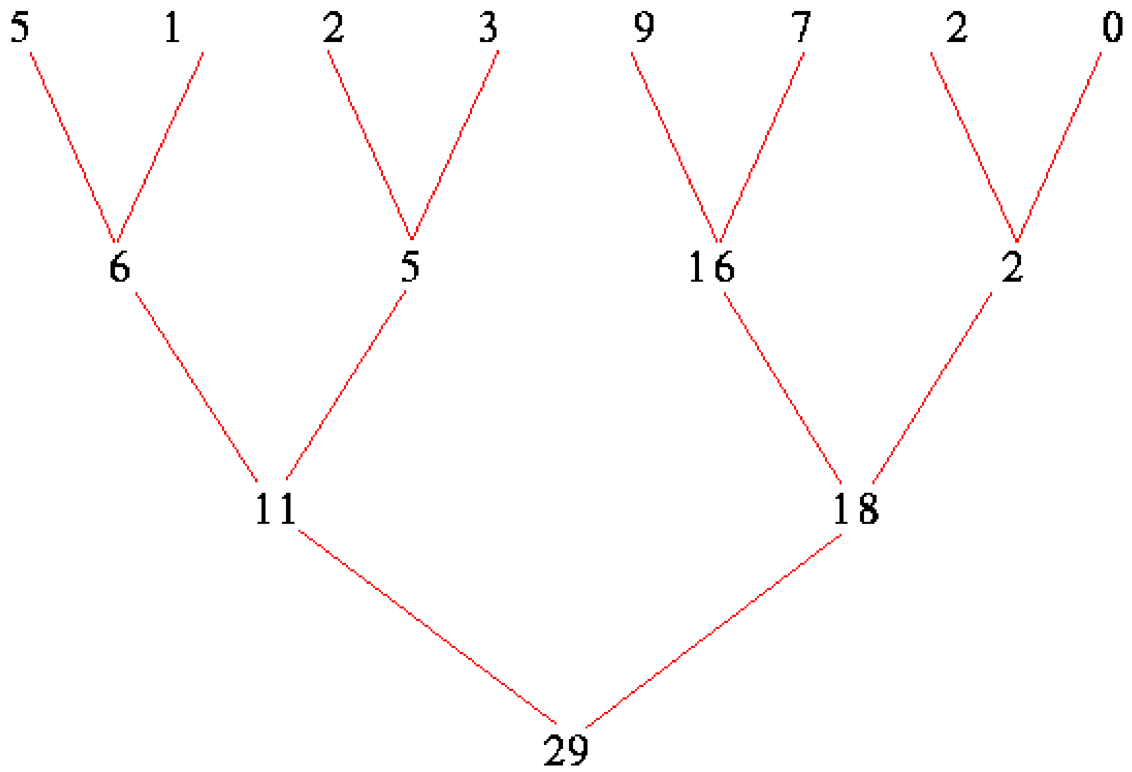

Figure 5: **Tree algorithm for parallelizing summation:** Numbers are added together in a "tree" data structure, so that eight of them may be combined in only three parallelizable stages.

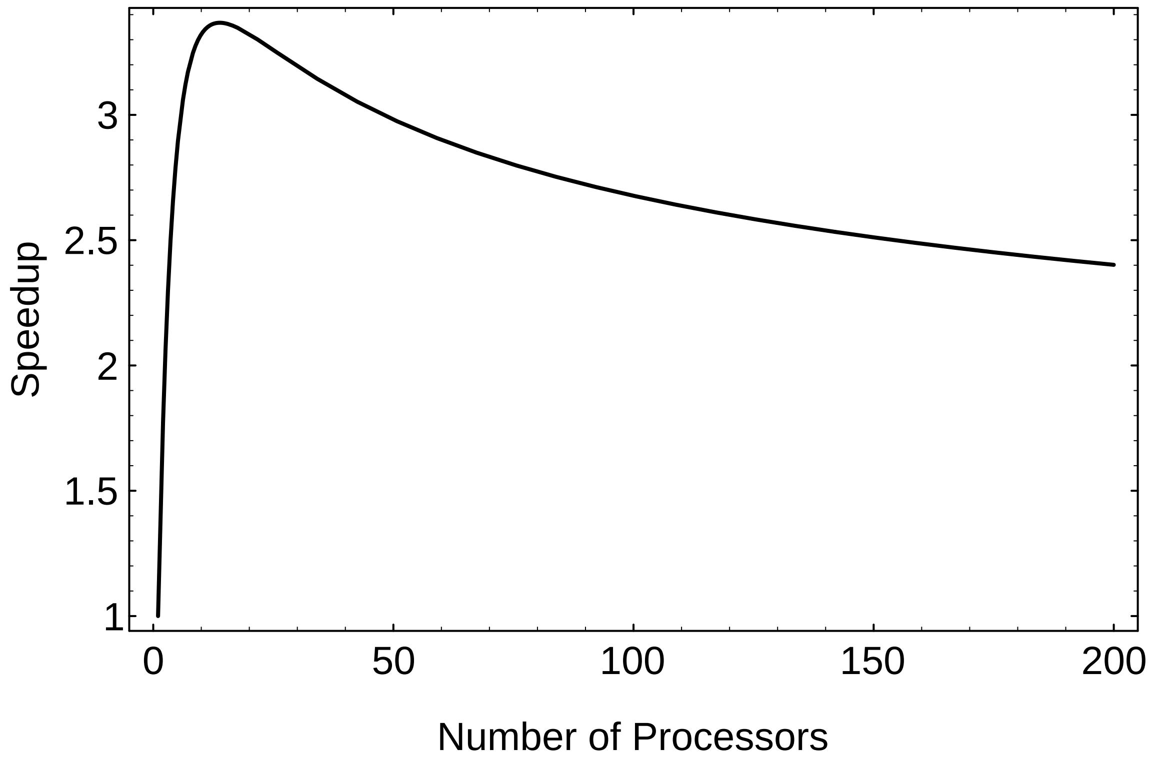

Figure 6: **Speedup versus number of processors**, illustrating Amdahl's Law, for $N = 100$ and $c_2/c_1 = c_3/c_1 = 5$.

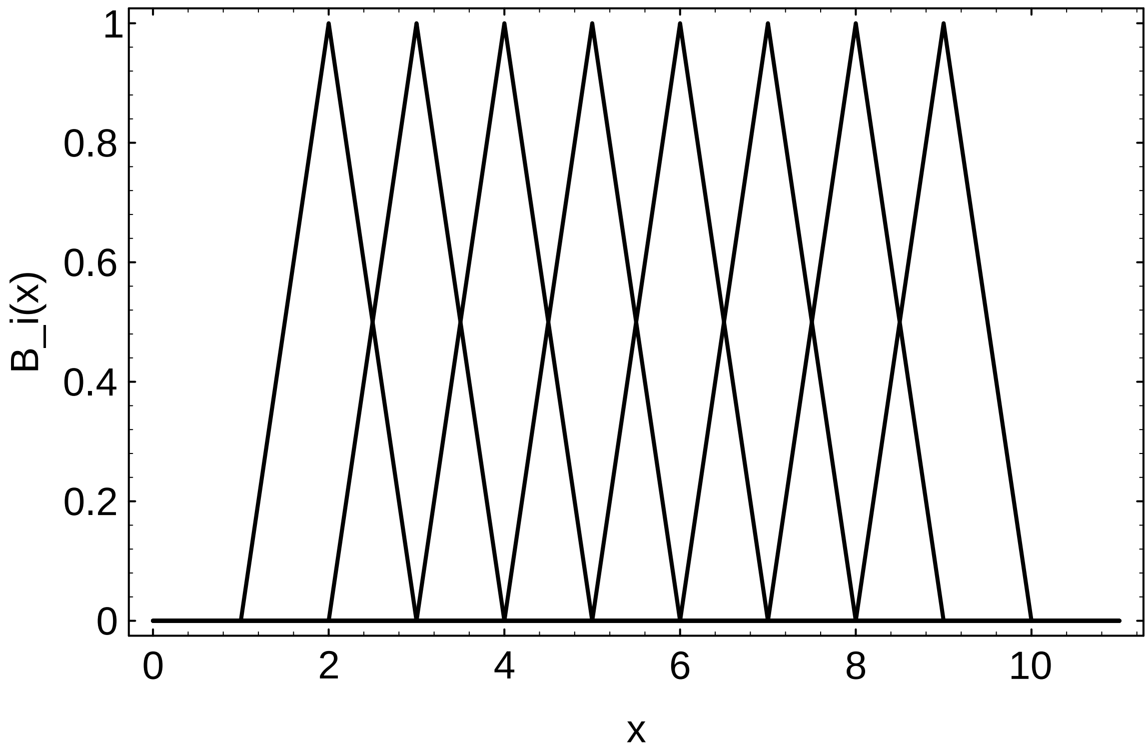

Figure 7: **Linear basis splines** for finite-element method in one dimension. Function $B_j(x)$ increases linearly from 0 at $x = x_{j-1}$ to 1 at $x = x_j$, and then decreases back to zero at $x = x_{j+1}$. Functions $B_2(x)$ through $B_9(x)$ are shown, for equally spaced gridpoints, $x_j = j$. These span the space of piecewise linear functions.

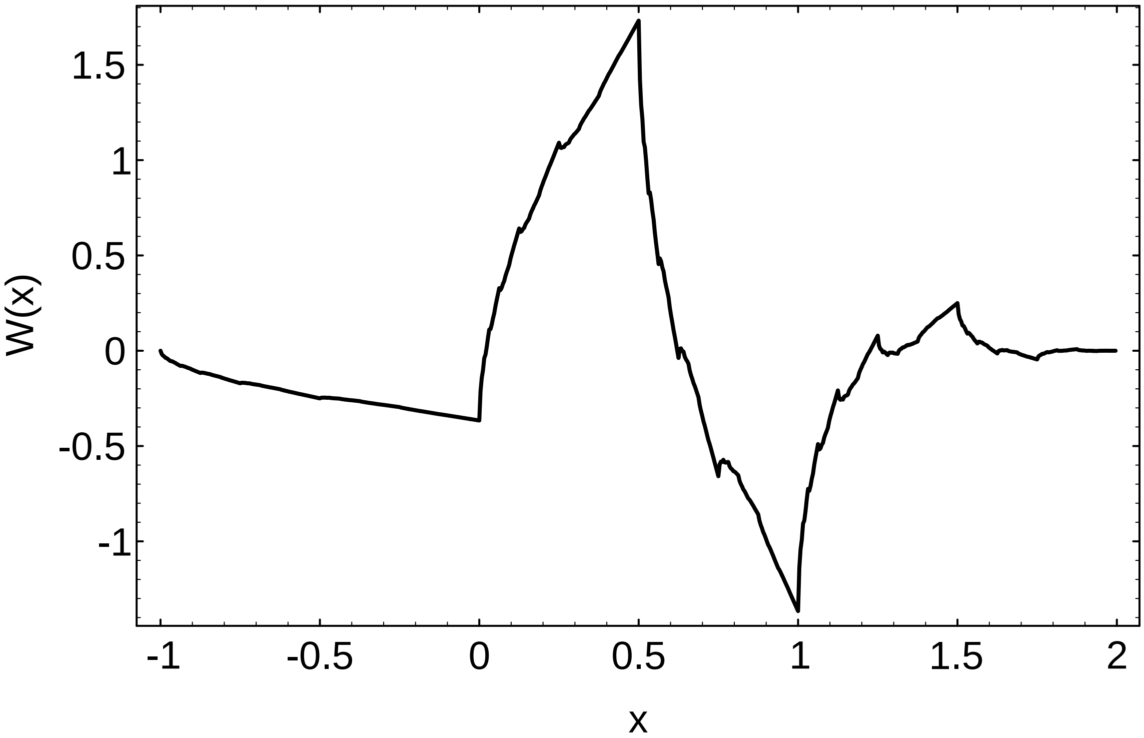

Figure 8: **Daubechies' Wavelet Function:** In spite of this graphical representation, it should be noted that this function is not smooth and is differentiable almost nowhere.

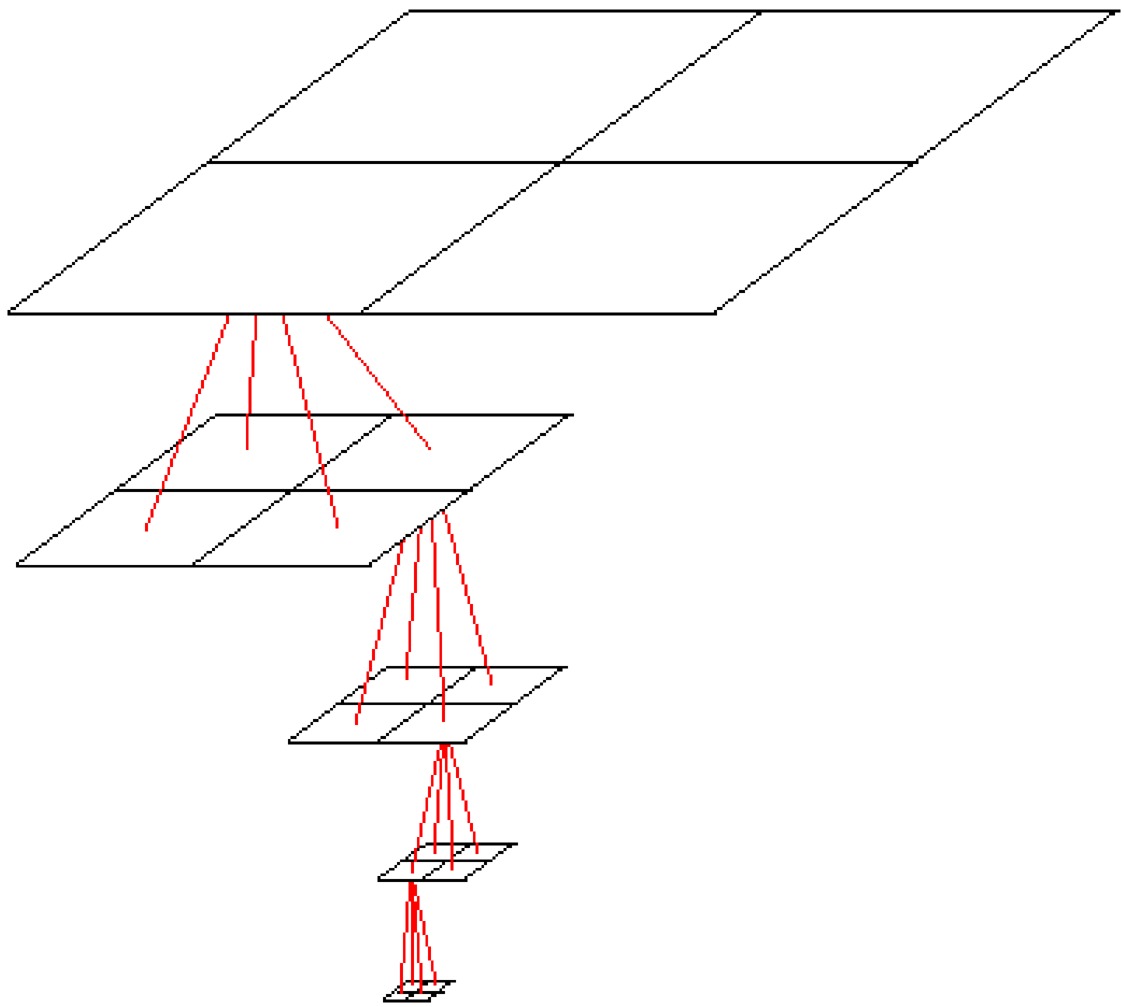

Figure 9: **Quad-Tree Division of Spatial Domain**

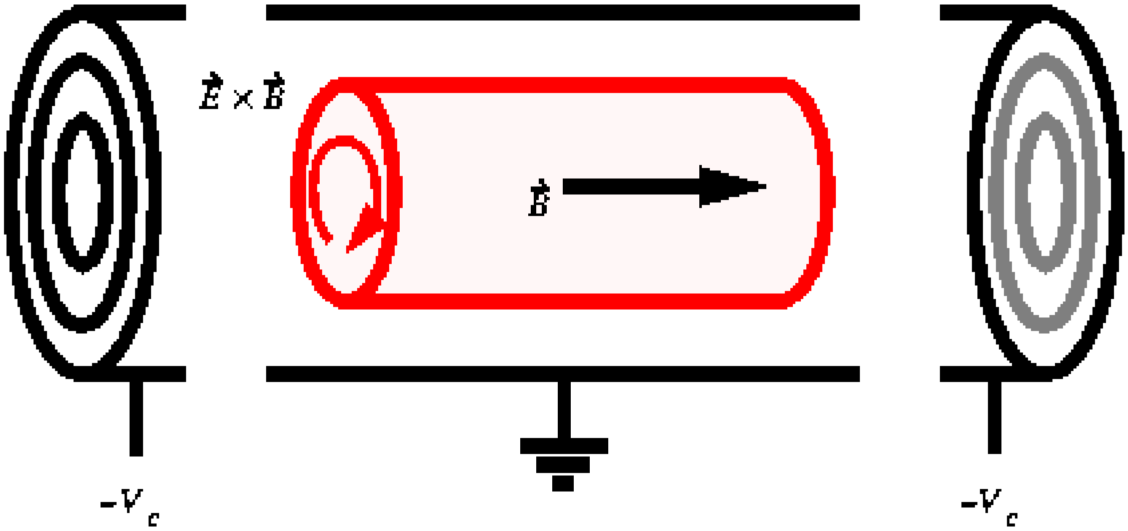

Figure 10: **Penning Trap** for Pure-Electron Plasma: The plasma is confined radially by an axial magnetic field **B**, and axially by an applied electrostatic potential V_c. It drifts azimuthally with the $\mathbf{E} \times \mathbf{B}$ drift velocity.

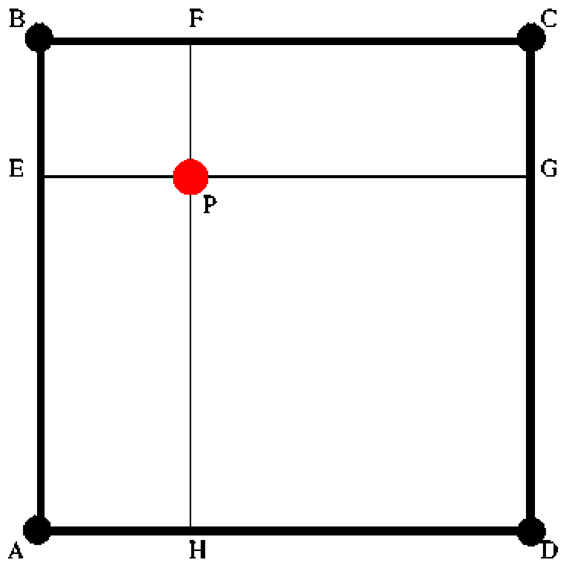

Figure 11: **Linear Interpolation of Particles to Grid**: The particle at position P contributes a fraction of its charge $a(PFCG)/a(ABCD)$ to gridpoint A, $a(PGDH)/a(ABCD)$ to gridpoint B, $a(PHAE)/a(ABCD)$ to gridpoint C, and $a(PEBF)/a(ABCD)$ to gridpoint D; here $a(ABCD)$ denotes the area of the quadrilateral $ABCD$.

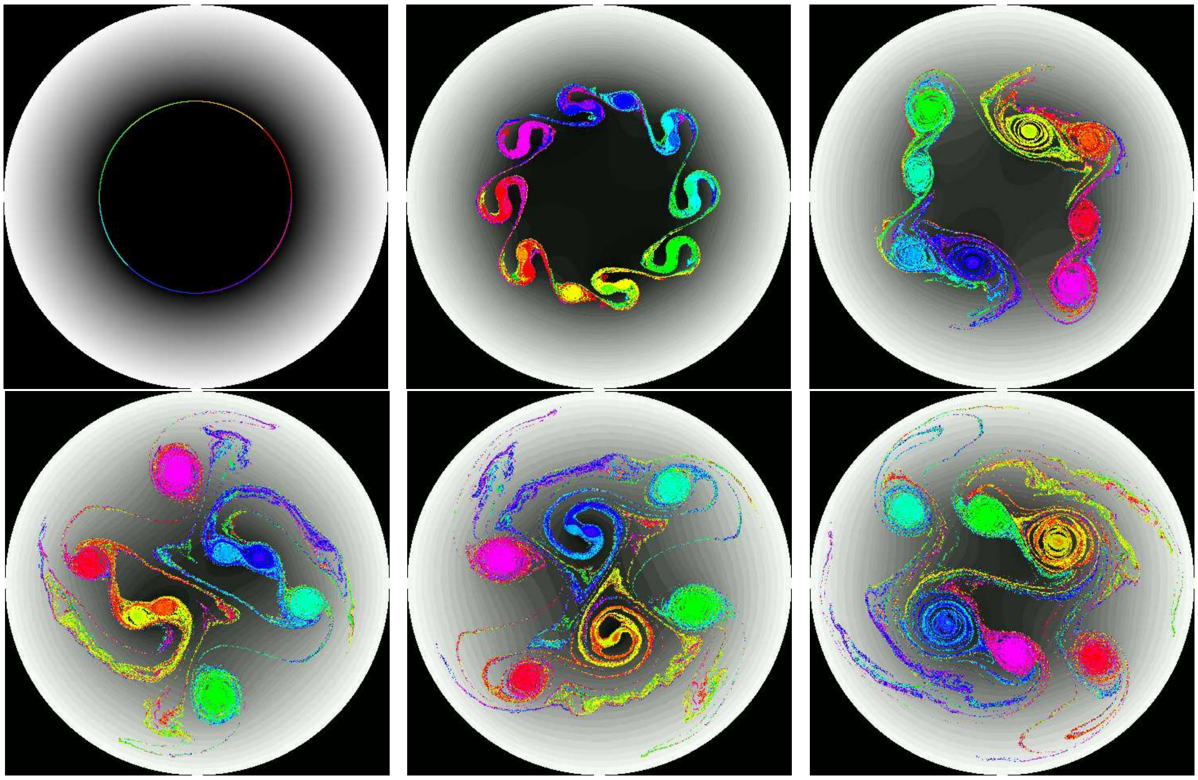

Figure 12: **Evolution of Pure-Electron Plasma in a Penning Trap by the Particle-in-Cell Method:** The plasma particles are represented by colored dots, and initialized on a ring of constant radius. The particles' colors are determined according to their initial position on the ring, and the particles retain their color as they evolve. The greyscale in the background represents the electrostatic potential. (Graphics by Erik Brisson, Scientific Computation and Visualization Group, Boston University.)

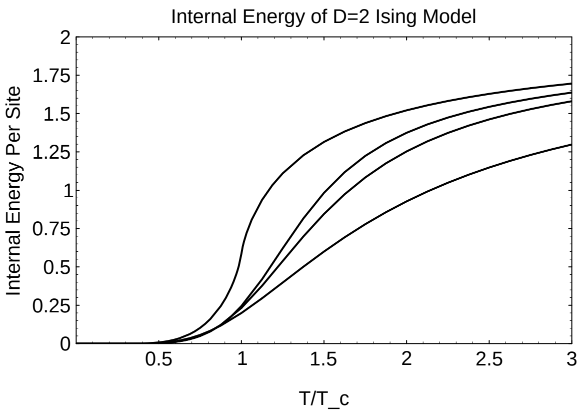

Figure 13: **Internal Energy of Two Dimensional Ising Model:** The curves are for 2×2, 3×3 and 4×4 lattices, and for an infinite lattice, in increasing order. The slope discontinuity at the critical temperature $T/T_c = 1$ is obtained only in the thermodynamic limit.

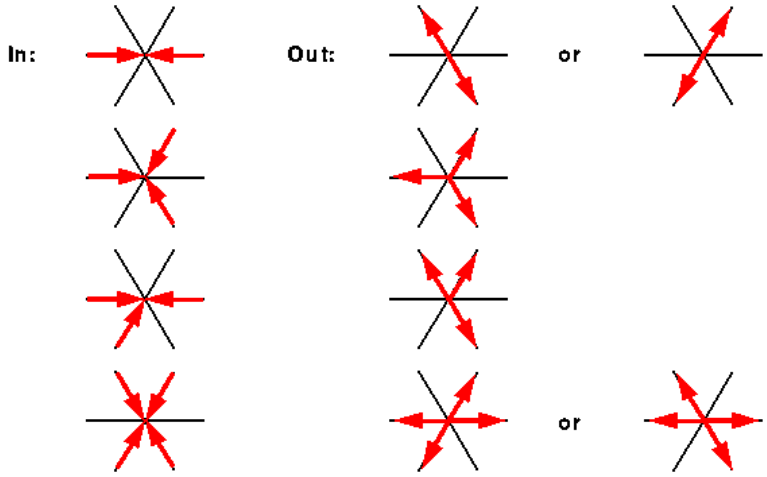

Figure 14: **Nontrivial Collisions of a Hydrodynamic Lattice Gas in Two Dimensions:** The collisions shown conserve mass and momentum. In all other situations, the particles pass through each other without interacting.