JOURNAL OF COMPUTATIONAL PHYSICS 64, 133–160 (1986)

Program for Efficient Monte Carlo Computations of Quenched $SU(3)$ Lattice Gauge Theory Using the Quasi-heatbath Method on a CDC CYBER 205 Computer

A. D. KENNEDY

Institute for Theoretical Physics, University of California, Santa Barbara, California 93106

J. KUTI

Department of Physics B-019, University of California San Diego, La Jolla, California 92093

S. MEYER

Fachbereich Physik, Abt. Theor. Physik, Universität Kaiserslautern, Postfach 3049, D-6750 Kaiserslautern, Germany

AND

B. J. PENDLETON

Department of Physics, University of California, Santa Barbara, California 93106

Received January 29, 1985; revised April 23, 1985

We describe the program SZINHUR which performs a Monte Carlo measurement of properties of lattice Quantum Chromodynamics. It uses the Quasi-Heatbath updating algorithm, which is known to reduce the correlations between successive sweeps through the spacetime lattice giving a performance improvement by a factor of roughly two over the ten-hit Metropolis procedure. The program measures the Polyakov loop and its correlation function. The program is highly vectorized and runs on a one-pipe CDC CYBER 205 at a speed of 53 μsec/link, which corresponds to an average computation rate of 93 Mflops. The program would run at almost twice this speed on a two-pipe machine.

INTRODUCTION

Lattice QCD

This paper describes a program which carries out a Monte Carlo evaluation of various properties of Lattice Quantum Chromodynamics (QCD) [1]. In its con-

0021-9991/86 $3.00

tinuum formulation QCD is thought to describe the physics of the strong interactions, and so an understanding of QCD is central to our understanding of both elementary particle and nuclear physics. Unfortunately, the coupling constant for continuum QCD is near to one at length scales of physical interest, and thus QCD is not readily amenable to the perturbative approximation methods which have proved so successful for Quantum Electrodynamics and the unified electroweak theories.

We are forced, therefore, to use numerical methods to learn about the nature of QCD and to make experimentally verifiable predictions. To do this, we approximate the spacetime continuum by a discrete lattice, which for simplicity we choose to have hypercubic symmetry. Furthermore, to have a system with a finite number of degrees of freedom we must restrict ourselves to a lattice of finite extent. One advantage of this approach, over and above that of facilitating numerical computations, is that the discrete lattice spacing regulates the ultraviolet (short distance) divergences of the quantum field theory.

One of the most useful techniques we can use to analyze the results of numerical lattice measurements is that of the renormalization group (RG) [2]. The idea is that we are not studying a particular lattice theory *per se*, but that we are studying a family of lattice approximations to the underlying continuum theory: the members of this family differ in that they have different lattice spacings, that is, they are coarser or finer meshes over the spacetime continuum. In order that these models describe the same physics we must adjust the coupling constant on the lattice and the lattice spacing simultaneously such that some physical quantity, say a correlation length, stays fixed. However, we know from perturbation theory how that coupling constant should vary with the lattice spacing as the lattice spacing goes to zero, and therefore we can test whether our lattice results are obeying this scaling relation to find out whether we are working on a lattice large and fine enough to give a good approximation to the continuum physics. To use the RG information we have to work on a series of lattices of different sizes and coupling constants until we can be sure that we are measuring the quantities of interest in the scaling window: the region in which the lattice is large enough for finite size effects to be under control and simultaneously for the coupling constant to be small enough to be out of the strong-coupling regime.

It is a matter of great physical interest also to study the properties of QCD at finite temperature [3], as this enables us to understand the thermodynamics and phase structure relevant to heavy ion collisions and the evolution of the early universe. The thermodynamics of hot QCD turns out to be equivalent to using an asymmetric lattice, in which in one direction the lattice is much shorter than in the other three (spatial) directions, and for which we impose periodic boundary conditions in the temperature direction. Quite apart from its intrinsic physical interest the bulk thermodynamic properties of QCD are easier to obtain reliably from numerical lattice computations than the highly nonlocal quantities (such as the mass spectrum) which are of physical interest at zero temperature. We can make considerable use of the RG method in this case not only by adjusting the length of

the lattice in the temperature direction, but also by observing carefully the effects of varying the spatial size of the lattice: this approach is known as finite size scaling.

Polyakov Loops

Once we have set up a lattice which approximates QCD we want to measure the behaviour of a probe which we put into the system; for instance, we would like to put two quarks into the system and measure the way that their interactions vary with their separation. This corresponds to measuring the expectation value of a nonlocal operator such as the Wilson loop. A particularly elegant variation of this approach is to measure the properties of Wilson loops which wrap around the lattice by virtue of the periodic boundary conditions, we shall call such operators Polyakov loops. Our program is set up to measure not only the expectation of a single Polyakov loop, which serves as an order parameter to distinguish the phases of finite temperature QCD, but also to measure the correlation function for two Polyakov loops as a function of their separation. This correlation function is directly related to the force between two quarks in QCD, and therefore enables us to investigate the inter-quark potential numerically. In particular, we can measure the asymptotic form of this potential (which is linear in the confining phase) and measure the string tension, which is the energy per unit length of the gluonic string which ties two quarks together inside a meson. This number, whose value is currently subject to some controversy [4–8], is vital in relating quantities measured on the lattice to the corresponding quantities measured in the real world.

One of the major limitations of these calculations from the physics viewpoint is that we have not included the effect of dynamical fermions, that is we have only included static quark loops as probes to measure the behaviour of the gluonic vacuum, but we have not investigated how virtual quarks can modify this behaviour. The introduction of dynamical quarks is a very difficult problem computationally, and one which we intend to study with later versions of the program.

Mathematical Formulation

On the computer we want to represent the lattice as a set of $SU(3)$ matrices which live on the links of a four-dimensional hypercubic lattice with (n_x, n_y, n_z, n_t) sites in the four directions. This means that a total of $72 n_x n_y n_z n_t$ real numbers are required to represent any single lattice configuration (strictly speaking we require only eight real parameters to describe an arbitrary $SU(3)$ element, but the rules for multiplying two group elements in terms of these eight numbers are prohibitively slow for practical computations, so we have counted each matrix as though it were an completely arbitrary 3×3 complex matrix with 18 real components instead). We shall call these link variables $U(x, y, z, t, \mu)$, where (x, y, z, t) specifies a lattice site and μ is one of the four link directions.

In terms of the link variables the Polyakov loop may be written as

$$P(U; x, y, z) \equiv \mathrm{Tr} \sum_{t=0}^{n_t - 1} U(x, y, z, t, \mu = 0),$$

where $\mu = 0$ is the temperature direction on the lattice. The trace takes into account the periodic boundary conditions imposed on the lattice in the temperature direction. This formula specifies the value of a Polyakov loop on a given background U configuration, but what we must calculate is the expectation value of P when averaged over all possible U configurations with a measure defined by the action S, that is

$$\langle P(x, y, z) \rangle \equiv \frac{1}{Z} \int (dU)\, P(U; x, y, z)\, e^{-\beta S(U)},$$

where β is related to the lattice coupling constant (and the temperature), and Z is the normalization constant

$$Z = \int (dU)\, e^{-\beta S(U)}.$$

For the action S we are free to choose any functional of the gauge field U which has the correct continuum limit; we use the simplest such action which was introduced by Wilson,

$$S(U) = \sum_{\square} \left(1 - \tfrac{1}{3} \operatorname{Re} \operatorname{Tr} \prod_{l \in \partial \square} U(l) \right),$$

where the sum extends over all the elementary squares (plaquettes) $\square$ on the lattice, and the product over the U matrices is taken around the perimeter (boundary) $\partial \square$ of plaquette $\square$ (the operation of taking the real part obviates the need for specifying or summing over the orientation of the plaquettes, so the sense in which we take the products of links around the boundary of a plaquette is irrelevant). This action has the important feature that it is local, meaning that if we change the value of U on only one link, say $l_0 = (x_0, y_0, z_0, t_0)$, then the change in the action may be computed by just looking at the plaquettes neighbouring this link:

$$\Delta S(U) = \Delta \sum_{\square \in \partial^* l_0} \left(1 - \tfrac{1}{3} \operatorname{Re} \operatorname{Tr} \prod_{l \in \partial \square} U(l) \right)$$

with $\partial^* l_0$ being the coboundary of l_0, namely the set of all plaquettes which include l_0 in their boundary. In four dimensions the sum is over exactly six plaquettes. Many other choices of action are possible, for example, the heat-kernel action or Symanzik's improved action [9, 10]: this latter is chosen so as to eliminate the difference between the lattice action and the continuum action to one order higher in perturbation theory at the cost of being less local (it involves next-to-nearest neighbour plaquettes in ΔS). Although the use of such alternatives to the Wilson action are more expensive in computation time and have not yet been shown to yield results markedly closer to the continuum limit, they certainly warrant further study.

Monte Carlo Method

Let us now consider how we perform the integral over U configurations in practice. Clearly we cannot perform a direct numerical integration, because there is an eight-dimensional group-space integral for each link, and thus for even a modestly small 10^4 lattice the U integral is 320,000 dimensional. We use a Monte Carlo (MC) technique [11] to evaluate the integral, and by the use of a variety of tricks we can reduce the computation to a tractable amount. The MC method evaluates the integral by selecting a sequence of U configurations at random and measuring the integrand on each member of the sequence. It is easy to show that the average value of an operator on a randomly chosen configuration is just the integral of the operator over the space of all U configurations, and that the average of the values measured on a sequence of T random configurations is again the desired integral but with a variance which falls like V/T, where V is the variance of a single measurement. Clearly we can obtain the desired answer with an arbitrarily small variance by making T sufficiently large, that is, by averaging over a large enough sample of statistically independent randomly chosen gauge field configurations. However, while this is necessary to produce useful answers we are limited to sampling at most of the order of a few thousand configurations given a reasonable amount of computer time, so to reduce the statistical uncertainties in the answer it is vital to make the variance V as small as possible.

There are a variety of techniques known for variance reduction, of which the one most useful in the present case is importance sampling. For this we generate U configurations not with respect to a uniform measure on the $SU(3)$ group manifold (Haar measure), but with respect to a measure which more closely resembles the function we are trying to integrate. If we were able to generate configurations according to the integrand itself (which *a fortiori* would have to be positive semidefinite everywhere) we could in principle make V zero. At first sight generating such an algorithm which would generate configurations with respect to an essentially arbitrary measure seems very difficult, but in fact it can be done quite easily. The trick is to use a Markov process which generates configurations which asymptotically tend to the desired distribution. After iterating this Markov process for a sufficiently long time (it converges exponentially in theory), we can assume that all subsequent configurations are being generated with the desired distribution.

There are only two drawbacks to this scheme: First, the configurations which are generated are correlated with each other. This means that the variance in the final result does not fall like V/T but only as VT'/T, where T' is the correlation time which is the number of steps between effectively statistically independent configurations. T' can be calculated from the values measured on individual configurations by dividing the data into blocks and performing some simple χ^2 tests on the block averages. It is immediately seen that reducing the correlation time by a factor of two, for example, is equivalent to doubling the number of configurations generated or to having a program which executes twice as fast.

Second, the Markov chain generates configurations with the correct relative

probability, but it tells us nothing about the absolute normalization. What we must do is to split the integrand into two factors $f(U) \cdot g(U)$, generate configurations via a Markov process with distribution proportional to $g(U)$, and then measure the value of $f(U)$ on these configurations. The resulting values for $\langle f \cdot g \rangle$ are then normalized relative to $\langle g \rangle$. As is usual we choose $g(U)$ to be the exponential of the action and $f(U)$ to be the operator we want to measure, such as the Polyakov loop P, but this decomposition leaves something to be desired. It is perfectly fine when we are measuring the energy, which is basically the mean plaquette or the expectation value of the action itself

$$\langle S \rangle = \frac{1}{Z} \int (dU)\, S(U)\, e^{-\beta S(U)}$$

because the importance sampling is quite good (roughly speaking $Se^{-\beta S}$ is not too different a function from $e^{-\beta S}$ itself); however, when we want to measure a nonlocal quantity such as the correlation function between two Polyakov loops, $\langle P(x, y, z) \cdot P(x', y', z') \rangle$, then we find that the statistical errors start becoming uncomfortably large. Unfortunately no better means of implementing importance sampling is known at present.

Markov Processes

Let us consider in a little more detail the Markov process we use to generate the probability distribution according to the function $g(U)$. A Markov process is defined by a transition probability $P(U, U')$, which is the probability of generating configuration U' given that the previous configuration of the Markov chain was U. The fact that each initial configuration U has to be transformed into some final configuration by the action of P implies that P must satisfy the relation

$$\sum_{U'} P(U, U') = 1.$$

Suppose we apply a single step of the Markov process to a configuration U which was selected with probability distribution $Q(U)$, then we will end up with a configuration U' with distribution

$$Q'(U') = \sum_{U} Q(U)\, P(U, U').$$

We define a probability distribution Q to be a fixed point of P if $Q' = Q$. We need to choose P in such a way that g is a fixed point. What is even more useful is the fact that if P is strongly ergodic, which means that any initial configuration can be taken to any final configuration with some nonvanishing probability by a single Markov step (formally we write this condition as $P(U, U') > 0$, $\forall U, U'$), then not only are we guaranteed that there is a unique fixed point, but also that iterating the Markov process from any initial probability distribution (and hence any initial con-

figuration) will cause the distribution of the configurations generated to tend to the fixed point. As we mentioned before the convergence rate is exponential, but unfortunately the theoretical bound on the exponent is not useful in practice: we are left to decide when we have reached the equilibrium distribution by heuristic methods.

We should not let the mathematical formalism outlined above obscure what can actually happen to such a Markov process. The main danger is that the system will get stuck in a metastable state: Intuitively we may imagine the distribution g to have several peaks separated by large valleys, then from some initial state we may rapidly climb one of the peaks and not leave it (by a large low-probability fluctuation) for a long time. This corresponds to the possibility that although P is strongly ergodic in the strict sense, it nevertheless has some very small components. This disease may be avoided to some extent by starting from widely different initial conditions, but it is quite possible that there are regions of the space of configurations which contribute importantly to the integrals which are hard to reach from almost anywhere in the space (e.g., a small high peak surrounded by deep wide trenches). Of course, we should not overlook the fact that the structure of g is telling us something about the physics of the situation as well, the existence of two peaks is evidence for a phase transition in the infinite volume continuum limit, and thus if we see values for the Polyakov loop which clump together in two regions with occasional jumps from one clump to the other we have a sensitive means of measuring the value of β_c (i.e., the transition temperature).

How do we construct a transition probability P which has g as a fixed point? What we want is a method which alters the initial configuration locally (i.e., link by link) so that we can take advantage of the locality of the action S. Suppose we can find a method of updating a single link l such that the transition probability $P_l(U, U')$ has g as a fixed point (U and U' differ only by the value of the $SU(3)$ matrix on the link l); this gives us a Markov step which preserves g but is manifestly not strongly ergodic. We can now use the simple fact that if P_l and $P_{l'}$ both have g as a fixed point, then so will the composite transformation $P_l \cdot P_{l'}$ (P_l followed by $P_{l'}$). Therefore, if we step sequentially through every link on the lattice applying the Markov step defined by P_l, the cumulative effect of the entire sweep through the lattice may itself be viewed as a single step of a Markov process which not only preserves g but also is strongly ergodic (each individual link update is assumed to be ergodic insofar as the new link variable may take any value in $SU(3)$ with a nonzero probability, and therefore it is possible to reach any new configuration of U matrices after sweeping once through the entire lattice).

For most lattice models there are two popular methods of generating an update for a single link.[1] The first is the Metropolis algorithm [13], which randomly chooses a new $SU(3)$ matrix U' subject to the conditions that U' and $U'^{\dagger}$ must be equiprobable, and that any $SU(3)$ matrix can be reached after enough steps, and then accepts or rejects the change depending on the change ΔS in the action. The method is based on the fact that it satisfies the criterion of detailed balance,

[1] The true $SU(3)$ heatbath method [12] is impractically slow.

$g(U)\,P_l(U, U') = g(U')\,P_l(U', U)$, which is easily shown (by integrating over U) to imply that g is a fixed point of P_l. Define the quantity ξ to be the ratio $g(U')/g(U)$, then we define

$$P_l(U, U') \equiv \begin{cases} 1 & \text{if } \xi \geqslant 1 \\ \xi & \text{if } \xi < 1, \end{cases}$$

and detailed balance is immediately satisfied. The quantity ξ is readily computed for the case $g(U) = e^{-\beta S(U)}$, as then $\xi = e^{\beta \Delta S}$. Unless β is small (which is known as strong coupling, because β is inversely proportional to the (square of) the QCD coupling constant) this method tends to produce a low acceptance rate for changes, or, to put it another way, we spend most of the computer time suggesting changes which are then not used. To increase the acceptance rate it is usual to generate a candidate U' uniformly from a small region of group space around U, and then to repeat this procedure several times so as to get a sufficiently large change $U' - U$ cumulatively. One advantage of such a multiple hit strategy is that the computation can be factored into the computation of what we call a *staple* (the sum of three-link products contributing to ΔS) which is common to each hit on a given link, and the final evaluation of ΔS and performance of the acceptance–rejection step.

The second method is the heatbath algorithm, which generates the new link with a probability $P_l(U, U') \propto g(U')$. This has the great advantage of generating the new link variable with exactly the probability required by g, and that the new link value is independent of the old one. This serves to reduce the correlation time T' which is of importance for the reasons outlined before. Sadly, for $SU(3)$ the amount of computation required to generate a new link variable according to the heatbath distribution is quite prohibitive. For the case of $SU(2)$, however, it is relatively straightforward to generate the required distribution; this led Cabibbo and Marinari [14] to suggest an algorithm which has many of the advantages of the heatbath method but is computationally tractable. Their method, which we shall call the *Quasi-Heatbath* (QH) algorithm, consists of considering two noncommuting $SU(2)$ subgroups of $SU(3)$. We consider a new $SU(3)$ matrix U' which can be reached from the old one U by the action of an element of the first subgroup. Using an $SU(2)$ heatbath algorithm due to Creutz [15] it is easy to generate this U' according to the full heatbath distribution restricted to the accessible set of $SU(3)$ matrices. The same procedure is now repeated for the second $SU(2)$ subgroup. As each step in this procedure has g as a fixed point, and as the combination of the two steps is weakly ergodic (i.e., if repeated often enough any $SU(3)$ matrix is accessible) it follows that this is a valid procedure to generate the desired Markov chain. Empirical tests [16, 14] have shown that the QH algorithm with two subgroup hits is about a factor of two better than a ten hit Metropolis update. Although the QH algorithm is more difficult to vectorize, and certainly much more complicated to code, than the Metropolis method we believe that the corresponding decrease in T' well justifies its use. Using vectorization techniques described later we have been able to make the QH program run as fast as, if not faster than,

corresponding ten-hit Metropolis programs; further speed improvements in the updating time are not crucial because the purely arithmetic evaluation of the staple which is required for all updating methods currently takes about 50% of the CPU time, even when fully vectorized.

VECTORIZATION TECHNIQUES

Parallel Updates

We now turn to the details of how the algorithms described above are implemented to take advantage of the features of the CYBER 205 computer.[2] One of the features of the CPU is that to obtain the high computational rate of which it is capable (200 Mflops for a 2 pipe machine using 32-bit half-precision floating-point arithmetic) it is necessary to make use of the vector processing instructions. This means that we must arrange for identical operations to be performed successively on a large number of consecutive elements of an array of numbers; indeed, to approach the asymptotic flow rate mentioned above it is necessary to operate on vectors at least a few hundred components in length. While this can be rather awkward to arrange, and certainly makes the code somewhat more obscure than it would otherwise be, it has the added advantage that the time spent in address calculation, loop control, etc., using the scalar processor is a negligible fraction of the total CPU time used (especially when the scalar instructions can be scheduled to be executed in parallel with the vector arithmetic). An unexpected consequence of this is that there is little benefit to be obtained in a highly vectorized program from coding it in assembly language rather than a high level language like FORTRAN. It is interesting to mention here that all the timings given for our program are in the absence of the instruction scheduling optimization, as we were never able to use this option due to compiler bugs: indeed, the timings are essentially unchanged for lattices of a physically interesting size even if we do not use the optimizer at all.

It is immediately apparent from the discussion of Markov processes given before that the order in which we update the links on the lattice is totally immaterial as far as convergence of the algorithm is concerned (although the convergence rate may be affected). In particular, there is no reason why we cannot update a whole family of links simultaneously, provided only that the updates are independent; that is, the value of a link being updated is not used in the computation of the staple for any other link in the family. A standard method of finding such a family of independently updateable links is to divide the sites of the lattice into two classes which we call even and odd parities. The parities alternate on the lattice in such a way that every even site is connected by the eight links in its coboundary only to odd sites, and vice versa. The situation is quite analagous to the pattern of black and white squares on a (two-dimensional) chessboard. Consider now the set of links in a given direction emanating from sites of a given parity, no link in this set contributes

[2] Reference [17] describes another program addressing similar problems using the CYBER 205.

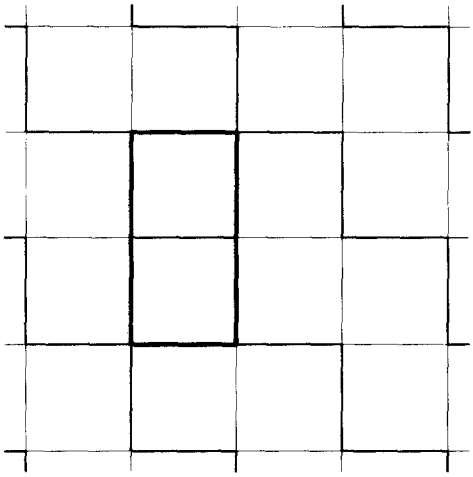

FIG. 1. An illustration of a "staple" and of the parity structure on a two-dimensional lattice. Light lines represent even links, medium lines represent odd links, and dark lines show the links belonging to the staple of a typical odd link.

to the staple required for the updating of any other link from the set, as may be seen from Fig. 1.

To make use of this set we store the lattice in two arrays, one for even parity links and the other for odd parity ones. We use the first array subscript to index the lattice site within these classes, and the subsequent subscripts (which vary less rapidly in FORTRAN) to index the colour components and direction of the U matrices. This means that, say, all the (2, 3) colour components in the z direction are in consecutive memory locations. Furthermore, we store the real and imaginary parts of the matrix elements in separate arrays and do not use the FORTRAN COMPLEX data type at all. All of these arrangements serve to guarantee that we may use the CYBER 205's vector instructions to update all the links in the set described above simultaneously.

Helical Boundary Conditions

In the discussion so far we have studiously avoided specifying the boundary conditions (BCs) to be imposed at the edges of the lattice, other than that the lattice must be periodic in the temperature direction. It is conventional to use periodic BCs in the spatial directions as well, but there is no fundamental reason why this has to be so: for the physical interpretation of lattice QCD results (with the notable exception of finite size scaling analysis) the spatial size is assumed infinite, or at least sufficiently larger than the temperature direction that the finite size effects may be ignored. In order to implement periodic BCs using the vectorization technique outlined above, however, it becomes necessary to perform random GATHER and SCATTER operations. While these are available in the hardware instruction reper-

toire of the CYBER 205, they are relatively slow operations and require extra temporary storage space for the gathered vectors. The actual speed of a GATHER operation depends on how many memory bank conflicts occur, which in turn depends on the nature of the index vector, but a rate of about 1.5 cycles (30 nsec) per element seems reasonable. This is to be compared with a rate of 0.5 cycles (10 nsec) per arithmetic operation on a one-pipe machine using half-precision arithmetic, and 0.25 cycles (5 nsec) for a two-pipe CPU.

With these timings in mind, it seems desirable to avoid the use of GATHER operations when possible, and in the present case it is quite easy to do so. What we do is to assign an integer index to each site on the lattice according to the rule

$$\mathcal{N}(x, y, z, t) \equiv x + n_x y + n_x n_y z + n_x n_y n_z t \qquad (\mathrm{mod}\ n_x n_y n_z n_t),$$

which is, of course, the same rule as is used to map a multidimensional array into the one-dimensional addressing scheme of a computer memory. To reach a neighbouring site from (x, y, z, t) in any of the four lattice directions we just have to add a fixed displacement to the starting site's index, namely

$$\left.\begin{aligned} \mathcal{N}(x+1, y, z, t) &\equiv \mathcal{N}(x, y, z, t) + 1, \\ \mathcal{N}(x, y+1, z, t) &\equiv \mathcal{N}(x, y, z, t) + n_x, \\ \mathcal{N}(x, y, z+1, t) &\equiv \mathcal{N}(x, y, z, t) + n_x n_y, \\ \mathcal{N}(x, y, z, t+1) &\equiv \mathcal{N}(x, y, z, t) + n_x n_y n_z, \end{aligned}\right\} \quad (\mathrm{mod}\ n_x n_y n_z n_t).$$

More generally, the index of a site reached from (x, y, z, t) by displacement vector (x', y', z', t') is

$$\mathcal{N}(x+x', y+y', z+z', t+t') \equiv \mathcal{N}(x, y, z, t) + \mathcal{N}(x', y', z', t') \qquad (\mathrm{mod}\ n_x n_y n_z n_t).$$

This scheme automatically imposes what we call helical BCs, which have the desirable property of making vectorized arithmetic entirely straightforward (a neighbour of a given site can be found by a single addition irrespective of the location of the site relative to the boundary. An intuitive picture of the meaning of helical BCs may be obtained by looking at the "extended zone diagram" for the two-dimensional analogue (we just draw a rectangle for our lattice volume and next to it the images of the lattice due to the BCs), for periodic BCs the diagram is just a regular pattern of unstaggered squares (Fig. 2). Note that the lattice is still periodic in the last direction, and that this necessary temperature periodicity is explicitly imposed by specifying that the above congruences are to be calculated modulo the number of lattice sites.

In order that the parity structure and the helical BCs be compatible with each other, we must ensure that the parity is well-defined globally over the whole lattice. This is easily done by defining the parity of a site (x, y, z, t) in terms of its index

$$\pi(x, y, z, t) \equiv \mathcal{N}(x, y, z, t) \qquad (\mathrm{mod}\ 2),$$

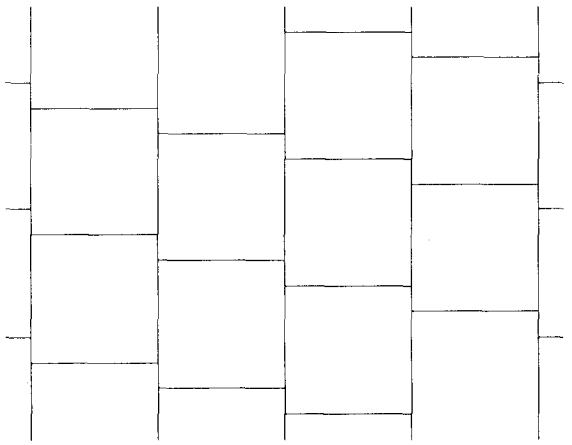

FIG. 2. The "extended zone diagram" for two-dimensional helical boundary conditions.

which is well defined provided that $n_x n_y n_z n_t$ is even. As we further require that nearest neighbour sites in any direction have opposite parities, and

$$\left.\begin{aligned}
&\pi(x+x',\, y+y',\, z+z',\, t+t') \\
&\quad \equiv \mathcal{N}(x+x',\, y+y',\, z+z',\, t+t') \\
&\quad \equiv \mathcal{N}(x,\, y,\, z,\, t) + \mathcal{N}(x',\, y',\, z',\, t') \\
&\quad \equiv \pi(x,\, y,\, z,\, t) + \pi(x',\, y',\, z',\, t'),
\end{aligned}\right\} \quad (\mathrm{mod}\ 2),$$

$$\left.\begin{aligned}
&\pi(1,0,0,0) \equiv 1, \\
&\pi(0,1,0,0) \equiv n_x, \\
&\pi(0,0,1,0) \equiv n_x n_y, \\
&\pi(0,0,0,1) \equiv n_x n_y n_z,
\end{aligned}\right\} \quad (\mathrm{mod}\ 2),$$

we must have n_x, n_y, and n_z all odd (and thus n_t even). We have shown that helical BCs satisfy all the necessary conditions provided that we have a lattice whose spatial dimensions are odd and whose temperature dimension is even: this is not too great a constraint, especially if we note that with periodic BCs we are constrained to a lattice all of whose dimensions are even.

The implementation of the periodicity in the temperature direction requires that we either always calculate the index of a neighbouring site modulo the lattice size, or that we keep up-to-date copies of the bottom t-slice of the lattice above the top

t-slice, and vice versa. We have opted for the latter alternative, despite the fact that it requires more storage space, because otherwise we would have to use GATHER instructions. The copying of these slices for the BCs takes a negligible amount of CPU time compared to the update and measurement times. We should also mention that the index defined above is not used directly in the program, because we store the even and odd links of the array separately, but the transformation to the actual array subscripts required poses no new problems of principle.

Vector-Scalar Quasi-Heatbath Implementation

We now turn to the details of the vectorized implementation of the QH algorithm. It is commonly believed that stochastic methods are ill-suited to parallel or vector processors because of their fundamentally random nature; we hope to explain why this is not necessarily true, and in particular how we can make use of the sparse vector instruction capability of the CYBER 205 combined with its fast scalar processor to implement an algorithm involving many random decisions.

First, let us specify the steps involved in applying the QH algorithm to update a link matrix U. The mathematical background to the algorithm has already been discussed, so we will just present the algorithm here as a recipe.

(1) Compute the staple Σ for the link U.

(2) Compute the submatrix Ξ of ΣU corresponding to the appropriate $SU(2)$ subgroup.

(3) Set

$$\xi = \tfrac{1}{2}\sqrt{\mathrm{Det}(\Xi - \Xi^\dagger + \mathbb{1}\ \mathrm{Tr}\ \Xi^\dagger)}.$$

(4) Set

$$\tilde{U} = \frac{1}{2\xi}(\Xi - \Xi^\dagger + \mathbb{1}\ \mathrm{Tr}\ \Xi^\dagger).$$

(5) Set

$$a_0 = \frac{1}{2\beta\xi}\ln[\rho_1(1 - e^{-4\beta\xi}) + e^{-4\beta\xi}] + 1,$$

where $0 \leqslant \rho_1 \leqslant 1$ is a uniformly distributed (pseudo) random number.

(6) Generate another uniform random number $0 \leqslant \rho_2 \leqslant 1$, and if $\rho_2 > \sqrt{1 - a_0^2}$ then go back to step (5).

(7) Generate a uniform random number $-1 \leqslant \rho_3 \leqslant 1$, and then set $a_3 = \rho_3\sqrt{1 - a_0^2}$.

(8) Generate another uniform random number, this time $0 \leqslant \rho_4 \leqslant 1$, and set

$$a_1 = \sqrt{1 - a_0^2}\sqrt{1 - \rho_3^2}\cos(2\pi\rho_4),$$
$$a_2 = \sqrt{1 - a_0^2}\sqrt{1 - \rho_3^2}\sin(2\pi\rho_4).$$

(9) Set $\tilde{V} = \tilde{U}^{-1}A$, where A is the $SU(2)$ matrix constructed by multiplying a_0 by the unit matrix and a_1, a_2, and a_3 by the appropriate Pauli matrices;

$$A = a_0 \mathbb{1} + i \sum_{j=1}^{3} a_j \sigma_j.$$

(10) The updated link variable is $U' = UV$, where V is the $SU(3)$ matrix made from the $SU(2)$ submatrix $\tilde{V}$, e.g.,

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & & \\ 0 & & \tilde{V} \end{pmatrix}$$

Clearly, all of the steps except for (5) and (6) are straightforward numerical computation. Step (1), in particular, involves adding together six three-link products of $SU(3)$ matrices: this requires 2,466 floating point operations.

The remaining arithmetic steps (i.e., all except steps (5) and (6), but including the precomputation of constants used in step (5)) require only the efficient evaluation of several mathematical functions, in particular the sine and cosine functions (of which we compute only one and extract the other using a square root—recall that square root extraction is a hardware operation on the CYBER 205 which has a flow rate almost equal to that of multiplication or addition when vectorized), and the exponential function. We use the standard FORTRAN library functions for these, as any improvement by hand coding would only increase the speed by a percent or less.

We also make great use of the built-in random number generator, using the "vectorized" full-precision version VRANF in the FORTRAN library. This is a linear congruential generator with a multiplier of 84,000,335,758,957 and no additive term, and works modulo 2^{47}. We have verified that this generator satisfies the criteria of the spectral test up to at least nine dimensions. Unfortunately, VRANF is in fact a scalar loop, and is therefore almost an order of magnitude slower than it should be (it also drops one number from the sequence when switching between scalar and vector calls): We have, therefore, rewritten the same generator using vector instructions.

The part of the algorithm whose vectorization is not obvious is the accept–reject steps. On a scalar computer we would just generate candidate a_0 values for one link until one was accepted, and then move on to the next link. To vectorize this procedure we are forced to take a different approach. Recall that we obtain a sufficient vector length by simultaneously updating all the links of a given parity in a given direction; we generate a candidate a_0 for each such link in parallel (i.e., using vector instructions) and generate a bit vector called REJECT using the CYBER 205's vector comparison operation. This array contains a 1 bit if the corresponding candidate a_0 fails the acceptance criterion. Next, we use the sparse vector instructions (invoked by the FORTRAN calls Q8VCMPRS, Q8VXPND,

and Q8VCTRL) to compress the coefficient vectors used to compute a_0 so that they contain entries only for links which have not yet had their a_0 value accepted. Then we generate another candicate a_0 for only these links, expand the resulting vector back to the original length, combine the new a_0 values with those already accepted, and recompute the bit vector REJECT. This vectorized loop is then iterated until almost all the a_0 values have been accepted. The time consuming part of the calculation is the generation of new a_0 values, which involves the evaluation of a logarithm; the vectorization scheme outlined above ensures that this critical part of the computation is only performed for those links which still need to be updated, and therefore it suffices to make the accept–reject steps as fast as the arithmetic steps in the algorithm. To indicate how successful this vectorization scheme is, we observe that the accept–reject steps take almost exactly twice as long to execute when we use a version of the FORTRAN compiler library which calculates logarithms using full-precision operations.

In the preceding paragraph we stated that we continue iterating the vectorized accept–reject loop until almost all the links were updated: If we had been using a true parallel processor this would be the best we could hope for, but on the CYBER 205 we can make use of the presence of a very fast scalar processor too. The problem is that for the values of β which arise in practice the acceptance rate in step 6 is about 25%, so the fraction f of reject links after t attempted updates is $f \approx e^{-t \ln(4/3)} \approx e^{-0.3t}$; this means that while 50% of the links have been updated after about 2.4 attempts on average, it would take ten times as many trials to reduce f to 0.1%. To get around this law of diminishing returns we keep track of the number of 1 bits in REJECT (using the Q8SCNT function) and when this falls below a predetermined number we switch to a scalar loop which steps through the remaining rejected links and generates a_0 values for each in turn until they are accepted. Of course, the scalar loop is much slower than the vector one, but it does not have the overhead of vector start-up times and the time required to expand and compress the vectors, so there is a cross over point at which scalar processing becomes cheaper: empirically this turns out to be for a vector length of around 60, although the exact value used is not at all critical. The total time spent in the scalar loops is, of course, much less than the time spent in the vector loop.

Typical relative times spent in the various parts of the program are indicated in

TABLE I

Fraction of Time Spent in Various Parts of the Program

Computation of staple	44%
Other arithmetic steps	15%
Vector accept/reject algorithm	30%
Scalar accept/reject algorithm	2%
Measurement of Polyakov loops	6%
Measurement of mean plaquette	2%
Reunitarization	1%

Table I. The timings depend on the lattice size and shape, the frequency with which measurements are made and so on, so the numbers are only a rough guide. In particular, the measurement of Polyakov loop correlation functions can take about 30% of the total time if n_t is small.

Reunitarization and Gauge Transformations

A variety of features are built into the program to ensure that numerical rounding errors are kept under control, and to check that the program is working properly.

The principle consequence of rounding errors is that after several sweeps through the lattice, the link variables drift away from the group manifold. An $SU(3)$ matrix may be parameterized as

$$\begin{pmatrix} \mathbf{u} \\ \mathbf{v} \\ \mathbf{u}^* \times \mathbf{v}^* \end{pmatrix},$$

where $\mathbf{u}$ and $\mathbf{v}$ are two orthonormal three component complex vectors with respect to the natural sesquilinear inner product. To reimpose the constraints implied by this parameterization, every so often the subroutine RESET is invoked to reunitarize the link variables. This is done in three steps,

(i) the first row of the matrix is rescaled so that its length (with respect to the natural metric) is one,

(ii) the second row is made othogonal to the first and then renormalized, and finally

(iii) the last row is constructed from the first two according to the parameterization given above.

During this procedure we keep track of the mean square deviation of the U matrices from unitarity, and the maximum value of this parameter is printed out at the end of each run. We have found that reunitarizing every five sweeps is more than adequate to keep the rounding errors under control (the maximum RMS deviation is of the order of one part in a million, which is roughly what is to be expected from using 32-bit arithmetic precision), and reunitarizing this often does not increase the computation time significantly.

It is useful to verify that all the parts of the program preserve the underlying gauge symmetry of QCD. This may be done by performing a random gauge transformation using the subroutine GAUGE and checking that physical-measured quantities are unchanged (up to the number of significant digits, which in our case is about six). The gauge transformation merely consists in generating a random $SU(3)$ matrix at each lattice site, and then multiplying each link by the gauge trans-

formation matrices on its boundary. It is rather hard to generate truly random $SU(3)$ matrices, in the sense of their being distributed uniformly with respect to Haar measure, but this is not necessary in the present case as physical observables must be invariant under any gauge transformation.

The program also contains subroutines to save and restore on disk the lattice configuration along with all the useful parameters and statistical accumulators associated with it (DISK), subroutines to initialize a lattice to a totally ordered or disordered configuration (START), and various other utility subprograms.

Future Plans

The previous sections described the current version of the program, which has been very extensively tested both against previous Monte Carlo results and analytic calculations such as strong coupling expansions where these are known. We have used the program to measure the nature and location of the phase transition [18] and the quark–antiquark potential near the continuum in lattice QCD [19]. We now turn to the improvements which will be incorporated in the next version of the program which is currently being developed. The principal change is that the program will be able to run on almost arbitrarily large lattices without incurring an intolerable paging overhead.

Improvements in Matrix Multiplication Algorithm

In the current version of the program the $SU(3)$ matrix multiplications are carried out in the obvious manner, assuming that the matrices involved are arbitrary 3×3 complex matrices. Naturally, however, we do not use the COMPLEX FORTRAN data type, because this causes the real and imaginary parts of a complex variable to be stored consecutively in memory, which prevents proper vectorization; instead we write all the operations in terms of the real and imaginary parts explicitly. Nevertheless, certain further optimizations are possible, first, because it is not optimal to treat complex matrix multiplication in the same way as real matrix multiplication, that is, by replacing the real additions and multiplications by complex ones; and second, because we can make use of the fact that the matrices are all elements of the defining representation of the group $SU(3)$, and therefore satisfy certain constraints.

The number of operations required for the various calculations involved in computing the staple are summarized in Table II, where (α) is the naive method of multiplying two complex numbers and (β) is a trick to reduce the number of multiplications by factorization:

$$(a + ib)(c + id)$$

$$= (ac - bd) + i(ad + bc) \qquad (\alpha)$$

TABLE II

The Number of Floating-Point Operations Required for Various Arithmetic Computations[a]

Computation	Operations	
	+	×
Complex scalar addition	2	0
Complex scalar multiplication (α)[a]	2	4
Complex scalar multiplication (β)[a]	5	3
$n \times n$ real matrix addition	n^2	
$n \times n$ complex matrix addition	$2n^2$	
$n \times n$ real matrix multiplication	$n^2(2n-1)$	
$n \times n$ complex matrix multiplication (α)	$8n^3-2n^2$	
$n \times n$ complex matrix multiplication (β)	$6n^3+2n^2$	

[a] Methods (α) and (β) are explained in the text.

multiplication takes 180 operations using method (β) as opposed to 198 by method (α), which means one can get a 10% speed increase by using method (β).

The other trick which can be used to speed up the arithmetic is to compute only the first two rows of the product matrix, because the third row can be reconstructed from the first two using the parameterization of an $SU(3)$ matrix given before.

Large Lattices

The current version of SZINHUR has been used extensively on lattices of various sizes, but it generates many page faults when it is used on a lattice size which is too large to fit into real memory. It is important to study the physics on somewhat larger lattices, and therefore we are implementing the following method of minimizing the I/O overhead.

The basic idea is, as always, to maximize the amount of computation which can be performed on that part of the lattice which is resident in real memory. Naturally, if the size of the lattice is larger than the size of the memory then it is unavoidable that the whole lattice will have to be paged in and out of memory at least once per sweep; this indicates that it is efficient to perform several updates on each link of the lattice while it is resident in real memory. On the other hand, it should be clear from the theoretical discussion of the Monte Carlo method given above that we must perform many sweeps through the lattice to generate statistically independent equilibrium configurations; to put it another way, updating parts of the lattice separately will bring them to local equilibrium, but global equilibrium is possible only by many sweeps through the entire lattice. The lesson here is that while performing multiple updates on a given section of the lattice is desirable from the viewpoint of writing an efficient program to run on large lattices, this should be done only in moderation if we want to reach global equilibrium, which was the reason for using a large lattice in the first place. We see, therefore, that there is a limit to

the amount of useful computation that can be performed on a large lattice for a given amount of I/O data transfer. Fortunately, even on a one megaword memory machine we should still be able to reduce the performance degradation due to I/O to a factor of 1.25 or less (N.B., while the I/O overhead increases the cost of running the program, it does not affect the update times; to this extent the update times do not give a complete specification of the performance of the program).

We should also point out here that although the CYBER 205 has virtual memory, this feature does not solve the problem of running on a large lattice. It does serve a useful purpose, however, insofar as the amount of real memory available to the program is larger than would be otherwise, because unneeded system routines and parts of the Monte Carlo program itself which are not currently being executed can be paged out of real memory.

To explain why virtual memory does not solve the major problem it is useful to make clear the distinction between virtual addressing and paging. Virtual addressing is implemented by the CYBER 205 hardware, and translates an address in the essentially infinite virtual address space into a real memory address. This translation is made in discrete pages of fixed size (64K words for large pages and 2K words for small pages). On a one megaword machine this means that only 13 large pages or 416 small pages (or some combination of the two) are available. The utility of virtual addressing is that virtual addresses need not be contiguous, and we can associate each page with an arbitrary region of virtual space. Paging, on the other hand, is a service of the operating system which, when informed by the hardware that a given virtual address requested does not have a corresponding real memory location, swaps a page from real memory with the image of the desired page from the paging disk.

From our point of view virtual addressing and paging pose conflicting constraints. The cost of a page fault is 32 msec for overhead plus 1 msec per block data transfer time; this means that a small page fault costs about 18 μsec/word, whereas a large page costs only 2.4 μsec/word. For a performance degradation of less than 1.25 we must perform more than 3,350 Flops (small pages) or 450 Flops (large pages) on every number (halfword) paged in. This consideration indicates that large pages reduce the I/O cost of the program by a factor of 7.5 relative to small pages. On the other hand, the nature of lattice QCD requires that we know the values of all the link variables contributing to the plaquettes in the coboundary of any link being updated. As each such link has 18 real components, and the vectorization method described before requires a long vector for each lattice direction and parity, we must have at least 144 vectors resident in real memory concurrently. It is clear that even using small pages we are very hard pressed to have enough pages to fit these different vectors into memory. The idea would be that each vector has a length corresponding to the number of lattice sites, but that we would work on only a section of the lattice at a time, allowing the system to fetch and map each section into memory automatically. Vectorizability requires that each of the vectors is contiguous in virtual memory, and hence the resident part of each vector must take up a whole page. The reason why it is not easy to do this, when we have 416 small

pages and need only 144 vectors, is that we must ensure that the neighbouring sections of the lattice which are referred to but not updated while working on a given section are available in real memory too. Working space for vector temporaries is not negligible either.

After weighing all these considerations, we concluded that the best way to enable the program to work on large lattices in an efficient way is to ignore the virtual memory, and to perform the I/O manipulations explicitly. We mean by this that we translate explicitly from an external data structure in which sections of the lattice are mapped into pieces of sufficient size, say large pages, into an internal one in which the vectors representing the components of the link variables are contiguous and compactly mapped into the available real memory. The actual I/O operations may be performed using the CYBER 205 implicit I/O facilities, or they may be performed using explicit I/O requests, this distinction is not important for the basic design of the program.

The disadvantages of this method are threefold,

(i) the code becomes more complicated as it has to perform the memory mapping itself, rather than using the existing implicit I/O facilities,

(ii) data has to be moved from a buffer into the working arrays, which takes time, and

(iii) the amount of available space is reduced by the need for buffers.

An advantage is that the program can store only the top two rows of the $SU(3)$ matrices on disk, and reconstruct the last row as part of its memory mapping, which reduces the amount of data transfers to disk to be reduced by $\frac{1}{3}$. We should point out that disadvantage (ii) is not very serious, as we are going to perform a large number of useful operations on each number input anyhow.

The helical boundary conditions make the splitting of the lattice into sections very easy, because the updating algorithm does not need any information about the position of the spatial boundaries of the lattice. This is because the BCs are imposed automatically by the way that the link variables are stored in memory. The size of a section (the part of the lattice resident in real memory at a given time) is determined only by the available real memory size, and not by the lattice size: this is to be contrasted with the situation using periodic BCs, where either one is constrained to a particular spatial lattice size or one has to keep track of the boundary conditions in a much more difficult way.

Fermions

The program computes physical quantities for quenched QCD, that is the theory of strong interactions between quarks mediated by dynamical gluons, but it does not include the effect of dynamical quarks. While there is some reason to believe that this approximation does not fundamentally change the nature of the theory, it would be much more satisfactory to include the effect of dynamical fermions. Several methods of doing this have been tried, but none of them are truly satisfac-

tory. We have a new stochastic method which we have tested successfully for simple two-dimensional models which we intend to implement for QCD. This will involve a (large) new subroutine to be invoked for every link update, but the rest of the program, such as measurement routines, reunitarization, random gauge transformations, etc., will be unchanged. The reason why including fermion effects is so hard is that they involve nonlocal quantities on the lattice, and thus the update time will be much larger than for the pure glue sector of the theory, perhaps by as much as a factor of 1,000. This being so, efficient implementation of the program becomes even more crucial, and in this case we can make use the GATHER and SCATTER instructions. Even so, we will have to be content with small lattices and relatively fewer sweeps through the lattice, but we hope to be able to make meaningful measurements of at least a few of the simpler properties of the full theory.

APPENDIX

In this Appendix we shall derive the basic results about the convergence of [redacted] tribution.

Markov Processes

Consider a system Ω which can be in any one of a family of states $\mathscr{S}$. At time t Ω has probability $Q_t(s)$ of being in state $s \in \mathscr{S}$, where $Q_t: \mathscr{S} \to [0, 1]$ is the *probability distribution* of Ω at time t. If Ω is in state $a \in \mathscr{S}$ at time t, then it has probability $P(b \leftarrow a)$ of being in state $b \in \mathscr{S}$ at time $t+1$. The operator $P: \mathscr{S} \to \mathscr{S}$ defines a *Markov Process.* We have immediately the following normalization conditions:

$$\sum_{a \in \mathscr{S}} Q_t(a) = 1 \qquad (\Omega \text{ must be in } \textit{some} \text{ state}), \tag{A1}$$

$$\sum_{b \in \mathscr{S}} P(b \leftarrow a) = 1 \qquad (\text{each state } a \text{ has to go somewhere}). \tag{A2}$$

Q is a *fixed point* of P if the following relation holds

$$\sum_{a \in \mathscr{S}} P(b \leftarrow a)\, Q(a) = Q(b) \qquad (\forall b). \tag{A3}$$

Detailed balance is satisfied iff

$$P(b \leftarrow a)\, Q(a) = P(a \leftarrow b)\, Q(b) \qquad (\forall a, b). \tag{A4}$$

THEOREM. *Detailed balance is sufficient (but not necessary) for Q to be a fixed point of P,*

$$\sum_a P(b \leftarrow a)\, Q(a) = \sum_a P(a \leftarrow b)\, Q(b) = Q(b).$$

If P_1 and P_2 are (arbitrary) Markov processes, then so is

$$P(a \leftarrow b) \equiv \sum_c P_2(a \leftarrow c)\, P_1(c \leftarrow b), \tag{A5}$$

because (i) $P_2(a \leftarrow c) \in [0, 1]$ and $P_1(c \leftarrow b) \in [0, 1] \Rightarrow P(a \leftarrow b) \in [0, 1]$ (clearly $P(a \leftarrow b) \geqslant 0$, and $\sum_c P_2(a \leftarrow c)\, P_1(c \leftarrow b) \leqslant \sum_c P_1(c \leftarrow b) = 1$ by (A2)), and (ii)

$$\begin{aligned}\sum_a P(a \leftarrow b) &= \sum_{a,c} P_2(a \leftarrow c)\, P_1(c \leftarrow b) \\ &= \sum_c \left\{ \sum_a P_2(a \leftarrow c) \right\} P_1(c \leftarrow b) \\ &= \sum_c P_1(c \leftarrow b) = 1,\end{aligned}$$

hence P satisfies (A2).

THEOREM. *If Q is a fixed point of both P_1 and P_2, then it is also a fixed point of P.*

$$\begin{aligned}\sum_a P(b \leftarrow a)\, Q(a) &= \sum_a \sum_c P_2(b \leftarrow c)\, P_1(c \leftarrow a)\, Q(a) \\ &= \sum_c P_2(b \leftarrow c) \sum_a P_1(c \leftarrow a)\, Q(a) \\ &= \sum_c P_2(b \leftarrow c)\, Q(c) = Q(b).\end{aligned}$$

We may define a metric on the space of probability distributions by defining the *distance*

$$d(Q_1, Q_2) \equiv \sum_{a \in \mathscr{S}} |Q_1(a) - Q_2(a)|. \tag{A6}$$

We may easily verify that this definition satisfies the necessary axioms,[3]

$$d(Q_1, Q_1) = 0; \qquad d(Q_1, Q_2) > 0 \quad (Q_1 \neq Q_2) \qquad \text{(Positivity)}; \tag{A7}$$

$$d(Q_1, Q_2) = d(Q_2, Q_1) \qquad \text{(Symmetry)}; \tag{A8}$$

$$d(Q_1, Q_2) + d(Q_2, Q_3) \geqslant d(Q_1, Q_3) \qquad \text{(Transitivity)}. \tag{A9}$$

We shall define P to be *strongly ergodic* if

$$P(a \leftarrow b) > 0 \qquad (\forall a, b). \tag{A10}$$

[3] The required inequality is a special case of the Minkowski inequality. For $x_i, y_i \in \mathbb{R}$ we have $|x_i| + |y_i| \geqslant |x_i + y_i| \Rightarrow \sum_i |x_i| + \sum_i |y_i| \geqslant \sum_i |x_i + y_i|$; hence $\sum_a |Q_1(a) - Q_2(a)| + \sum_a |Q_2(a) - Q_3(a)| \geqslant \sum_a |Q_1(a) - Q_2(a) + Q_2(a) - Q_3(a)| = \sum_a |Q_1(a) - Q_3(a)|$.

We may then prove

THEOREM. *A strongly ergodic Markov process P is a contraction mapping with respect to the metric d,*

$$d(PQ_1, PQ_2) \leqslant \alpha d(Q_1, Q_2) \qquad (\forall Q_1, Q_2), \tag{A11}$$

where $\alpha < 1$.

For $Q_1 = Q_2$ the result (A11) follows trivially from (A7), so we may assume that $Q_1 \neq Q_2$. Let us define

$$\Delta Q(b) \equiv Q_1(b) - Q_2(b), \tag{A12}$$

and $\mathscr{S}_\pm \equiv \{\xi \mid \Delta Q(\xi) \gtrless 0\} \subset \mathscr{S}$. From the definition of d (A6),

$$\begin{aligned} d(PQ_1, PQ_2) &= \sum_{a \in \mathscr{S}} |(PQ_1)(a) - (PQ_2)(a)| \\ &= \sum_{a \in \mathscr{S}} \left| \sum_{b \in \mathscr{S}} P(a \leftarrow b)\, Q_1(b) - \sum_{b \in \mathscr{S}} P(a \leftarrow b)\, Q_2(b) \right| \\ &= \sum_{a \in \mathscr{S}} \left| \sum_{b \in \mathscr{S}} P(a \leftarrow b)\, \Delta Q(b) \right| \\ &= \sum_{a \in \mathscr{S}} \left| \sum_{b \in \mathscr{S}_+} P(a \leftarrow b)\, \Delta Q(b) + \sum_{b \in \mathscr{S}_-} P(a \leftarrow b)\, \Delta Q(b) \right| \\ &= \sum_{a \in \mathscr{S}} \left\{ \sum_{b \in \mathscr{S}_+} P(a \leftarrow b)\, \Delta Q(b) - \sum_{b \in \mathscr{S}_-} P(a \leftarrow b)\, \Delta Q(b) \right\} \\ &\quad - 2 \sum_{a \in \mathscr{S}} \min_{\pm} \left| \sum_{b \in \mathscr{S}_\pm} P(a \leftarrow b)\, \Delta Q(b) \right| \end{aligned}$$

(using the identity $||x| - |y|| = |x| + |y| - 2\min(|x|, |y|)$)

$$\begin{aligned} &= \sum_{a \in \mathscr{S}} \sum_{b \in \mathscr{S}} P(a \leftarrow b)\, |\Delta Q(b)| - 2 \sum_{a \in \mathscr{S}} \min_{\pm} \left| \sum_{b \in \mathscr{S}_\pm} P(a \leftarrow b)\, \Delta Q(b) \right| \\ &= \sum_{b \in \mathscr{S}} |\Delta Q(b)| - 2 \sum_{a \in \mathscr{S}} \min_{\pm} \left| \sum_{b \in \mathscr{S}_\pm} P(a \leftarrow b)\, \Delta Q(b) \right| \\ &\leqslant \sum_{b \in \mathscr{S}} |\Delta Q(b)| - 2 \sum_{a \in \mathscr{S}} \min_{\pm} P_{\min}^{(a)} \left| \sum_{b \in \mathscr{S}_\pm} \Delta Q(b) \right| \end{aligned} \tag{A13}$$

with $P_{\min}^{(a)} \equiv \min_{b \in \mathscr{S}} P(a \leftarrow b) > 0$. However, using the normalization condition (A1) we have

$$\sum_{b \in \mathscr{S}_+} \Delta Q(b) + \sum_{b \in \mathscr{S}_-} \Delta Q(b) = \sum_{b \in \mathscr{S}} \Delta Q(b) = \sum_{b \in \mathscr{S}} Q_1(b) - \sum_{b \in \mathscr{S}} Q_2(b) = 1 - 1 = 0,$$

so

$$\left|\sum_{b\in\mathscr{S}_+}\Delta Q(b)\right| = \left|\sum_{b\in\mathscr{S}_-}\Delta Q(b)\right| = \tfrac{1}{2}\sum_{b\in\mathscr{S}}|\Delta Q(b)|. \tag{A14}$$

Combining (A13) and (A14) we find

$$d(PQ_1, PQ_2) \leqslant \sum_{b\in\mathscr{S}}|\Delta Q(b)| - \sum_{a\in\mathscr{S}} P_{\min}^{(a)} \sum_{b\in\mathscr{S}}|\Delta Q(b)|,$$

or $d(PQ_1, PQ_2) \leqslant \alpha d(Q_1, Q_2)$ as required, where

$$0 < \alpha \leqslant 1 - \sum_{a\in\mathscr{S}} P_{\min}^{(a)} < 1. \tag{A15}$$

Starting from an arbitrary probability distribution Q we may generate the sequence of distributions $(Q, PQ, P^2Q, P^3Q,\ldots)$ by successive application of the Markov matrix P. If P is strongly ergodic then this sequence converges to a unique limit $P^\infty Q$, because the space of probability distributions is complete in the topology generated by d, and by the following result:

LEMMA. *The sequence* $(Q,\ldots, P^nQ,\ldots, P^{n'}Q,\ldots)$ *is Cauchy, that is,* $\forall\varepsilon\ \exists N$ *such that* $\forall n, n' \geqslant N$,

$$0 \leqslant d(P^nQ, P^{n'}Q) < \varepsilon. \tag{A16}$$

If Q is a fixed point of P then this is obvious, otherwise by transitivity (A9)

$$\begin{aligned}
d(P^nQ, P^{n'}Q) &\leqslant d(P^nQ, P^NQ) + d(P^{n'}Q, P^NQ)\\
&\leqslant \sum_{k=0}^{n-N-1} d(P^{N+k}Q, P^{N+k+1}Q)\\
&\quad + \sum_{k'=0}^{n'-N-1} d(P^{N+k'}Q, P^{N+k'+1}Q)\\
&\leqslant \sum_{k=0}^{n-N-1} \alpha^{N+k} d(Q, PQ) + \sum_{k'=0}^{n'-N-1} \alpha^{N+k'} d(Q, PQ)\\
&= \alpha^N d(Q, PQ)\left\{\sum_{k=0}^{n-N-1}\alpha^k + \sum_{k'=0}^{n'-N-1}\alpha^{k'}\right\}\\
&= \alpha^N d(Q, PQ)\left\{\frac{2-\alpha^{n-N}-\alpha^{n'-N}}{1-\alpha}\right\}\\
&< \frac{2\alpha^N}{1-\alpha} d(Q, PQ),
\end{aligned}$$

and therefore (A16) is satisfied $\forall \varepsilon > 0$ with $N \geqslant \ln(\varepsilon(1-\alpha)/2d(Q, PQ))/\ln \alpha$, where α satisfies (A15).

Quasi-heatbath Algorithm

When performing stochastic computations to solve lattice gauge theories we need to generate field configurations according to the probability distribution

$$P(U) = \frac{1}{Z} e^{-\beta S(U)}, \tag{A17}$$

where $Z = \int (dU)\, e^{-\beta S(U)}$, $\mathcal{G}$ is the gauge group, U is the $\mathcal{G}$-valued link variable on the link under consideration, $S(U)$ is the lattice action viewed as a function of U (it also depends upon the neighbouring link variables of course), and (dU) is the $\mathcal{G}$-invariant Haar measure.

Whereas a true "heatbath" method would produce new link variables according to this distribution, the QH method merely ensures that given an initial U distributed according to (A17) it will generate a new U' with the same distribution: i.e., $P(U)$ is a fixed point of the QH transformation. If we further ensure that any possible $U' \in \mathcal{G}$ can be generated by a sequence of such transformations ($P^n(U \to U') > 0$ for large enough n) then applying the method stepwise to each link on the lattice is a weakly ergodic procedure. These two facts together ensure convergence of the Markov process to the desired equilibrium distribution (A17).

The QH method consists of applying a sequence of transformations restricted to some subgroup $\mathcal{H} \subset \mathcal{G}$. We select an element $h \in \mathcal{H}$ and define $U' \equiv Uh$, where we choose h such that the U' value is distributed according to (A17) restricted to the image of U, namely $U\mathcal{H}$. In other words, we generate U' with a heatbath distribution over the subset of $\mathcal{G}$ which can be reached by a transformation in $\mathcal{H}$. A single "hit" of this sort is not even weakly ergodic in general, because $\mathcal{H}$ does not act on $\mathcal{G}$ without fixed points; however, if we use a sequence of hits using different subgroups ($\mathcal{H}$, $\mathcal{H}'$,...) such that $\mathcal{G}$ has no fixed points under this action, then weak ergodicity ensues.

Note that

(i) the QH procedure is not a heatbath,

(ii) it satisfies detailed balance for a single hit (because it generates a heatbath distribution over accessible states), and

(iii) it does not satisfy detailed balance for multiple hits.[4]

Suppose we generate h with distribution $Q_U(h)$, then U' will be distributed according to

$$P'(U') = \int (dh)(dU)\, P(U)\, Q_U(h)\, \delta(U' - Uh). \tag{A18}$$

[4] If $h \in \mathcal{H}$, $h' \in \mathcal{H}'$, then $U' = Uhh' \Rightarrow U = U'h'^{-1}h^{-1} \neq U'\tilde{h}\tilde{h}'$, because in general there are no $\tilde{h} \in \mathcal{H}$, $\tilde{h}' \in \mathcal{H}'$ satisfying $h'^{-1}h^{-1} = \tilde{h}\tilde{h}'$.

Our claim is that if $Q_U(h)$ is a heatbath distribution restricted to $U\mathscr{H}$, or to be precise

$$Q_U(h) = \frac{1}{Z_U} e^{-\beta S(Uh)} \chi(h), \tag{A19}$$

where χ is the characteristic function for $\mathscr{H}$, and $Z_U = \int (dh)\, e^{-\beta S(Uh)} \chi(h)$, then P' will equal P. This is easily proved, for

$$\begin{aligned} P'(U') &= \int (dh)(dU)\, P(U)\, Q_U(h)\, \delta(U' - Uh) \\ &= \int (dU)\, P(U)\, Q_U(U^{-1}U') \qquad \text{(A20)} \\ &= \int (dU)\, Z^{-1} e^{-\beta S(U)} Z_U^{-1} e^{-\beta S(U')} \chi(U^{-1}U') \\ &= Z^{-1} e^{-\beta S(U')} \int (dU) \frac{e^{-\beta S(U)} \chi(U^{-1}U')}{\int (dh')\, e^{-\beta S(Uh')} \chi(h')} \\ &= P(U') \int (dh) \frac{e^{-\beta S(U'h^{-1})} \chi(h)}{\int (dh'')\, e^{-\beta S(U'h''^{-1})} \chi(hh''^{-1})}, \qquad \text{(A21)} \end{aligned}$$

where $h = U^{-1}U'$ as before, and $h'' = h'^{-1}h$. Finally,

$$P'(U') = P(U') \frac{\int (dh)\, e^{-\beta S(U'h^{-1})} \chi(h)}{\int (dh'')\, e^{-\beta S(U'h''^{-1})} \chi(h'')} = P(U'), \tag{A22}$$

where in Eqs. (A20), (A21), (A22) we have made use of the invariance of the Haar measure and of the characteristic function χ,

$$\begin{gathered} (dh) = (d(Uh)), \\ (dh'') = (d(h'^{-1}h)) = (dh'^{-1}) = (dh'), \\ \chi(hh''^{-1}) = \chi(h''^{-1}) = \chi(h'') \qquad \text{for} \quad \chi(h) = 1 \Leftrightarrow h \in \mathscr{H}. \end{gathered}$$

Creutz Algorithm

The main problem is thus reduced to generating the distribution (A19). This may be done using Creutz's $SU(2)$ heatbath algorithm for the interesting case when $\mathscr{H} = SU(2)$, and when the action has the form Re Tr ΣU, where Σ is the sum over all six "staples" in the coboundary of link U. Note that $\Sigma \notin \mathscr{G}$ in general. According to (A19) we want to generate h such that its distribution is

$$Q_U(h) \propto e^{(\beta/3)\,\mathrm{Re\ Tr}\,\Sigma U h} \chi(h).$$

For the cases of interest h may be represented as a 2×2 $SU(2)$ matrix $\bar{h}$ embedded in a $\mathscr{G}$ matrix, and $\operatorname{Tr}\Sigma U h = \operatorname{Tr}\Xi\bar{h} + \text{const.}$, where Ξ is also a 2×2 matrix: In practice Ξ is a trivial submatrix of ΣU. We must, therefore, generate the distribution

$$Q_\Xi(\bar{h}) = Z_\Xi^{-1} e^{(\beta/3)\operatorname{Re}\operatorname{Tr}\Xi\bar{h}} \tag{A23}$$

for $\bar{h}\in SU(2)$ and Ξ an arbitrary 2×2 complex matrix:

$$\bar{h} = \overline{h_0}\mathbb{1} + i\bar{\mathbf{h}}\cdot\sigma \qquad (\overline{h_\alpha}\in\mathbb{R},\ \overline{h_0}^2 + \bar{\mathbf{h}}\cdot\bar{\mathbf{h}} = 1),$$
$$\Xi = \Xi_0\mathbb{1} + i\boldsymbol{\Xi}\cdot\sigma \qquad (\Xi_\alpha\in\mathbb{C}).$$

Now,

$$\begin{aligned}\operatorname{Re}\operatorname{Tr}\Xi\bar{h} &= \operatorname{Re}\operatorname{Tr}[(\Xi_0\mathbb{1} + i\boldsymbol{\Xi}\cdot\boldsymbol{\sigma})(\overline{h_0}\mathbb{1} + i\bar{\mathbf{h}}\cdot\boldsymbol{\sigma})]\\ &= \operatorname{Re}\operatorname{Tr}[(\Xi_0\overline{h_0} - \boldsymbol{\Xi}\cdot\bar{\mathbf{h}})\,\mathbb{1} + (\cdots)\cdot\boldsymbol{\sigma}]\\ &= \xi\operatorname{Tr}u\bar{h},\end{aligned}$$

where $u\in SU(2)$, and $\xi u_0 = \operatorname{Re}\Xi_0$, $\xi\mathbf{u} = \operatorname{Re}\boldsymbol{\Xi}$. Consider $\Xi - \Xi^\dagger = (\Xi_0 - \Xi_0^*)\,\mathbb{1} + i(\boldsymbol{\Xi} + \boldsymbol{\Xi}^*)\cdot\boldsymbol{\sigma}$ and $\operatorname{Tr}\Xi^\dagger = 2\Xi_0^*$, hence

$$\begin{aligned}\Xi - \Xi^\dagger + \mathbb{1}\operatorname{Tr}\Xi^\dagger &= 2\operatorname{Re}(\Xi_0)\,\mathbb{1} + 2i\operatorname{Re}(\boldsymbol{\Xi})\cdot\boldsymbol{\sigma}\\ &= 2\xi u_0\mathbb{1} + 2i\xi\mathbf{u}\cdot\boldsymbol{\sigma} = 2\xi u,\end{aligned}$$

so we have

$$4\xi^2 = \operatorname{Det}[\Xi - \Xi^\dagger + \mathbb{1}\operatorname{Tr}\Xi^\dagger],$$
$$u = \frac{1}{2\xi}(\Xi - \Xi^\dagger + \mathbb{1}\operatorname{Tr}\Xi^\dagger),$$

and from (A23) $Q_\Xi(\bar{h}) = Z_\Xi^{-1} e^{(\beta/3)\xi\operatorname{Tr}u\bar{h}}$.

Writing $a = u\bar{h}\in SU(2)$ we may parameterize a as $a = a_0\mathbb{1} + i\boldsymbol{a}\cdot\boldsymbol{\sigma}$ with $(a_\alpha\in\mathbb{R}$, $a_0^2 + \boldsymbol{a}\cdot\boldsymbol{a} = 1)$, and Haar measure on $SU(2)$ is $(da) = da_0 d^3a_i\delta(1 - a_0^2 - \boldsymbol{a}\cdot\boldsymbol{a})$. In polar coordinates

$$\begin{aligned}(da) &= da_0\,dr\,d\theta\,d\varphi\,\frac{\partial(a_1, a_2, a_3)}{\partial(r, \theta, \varphi)}\,\delta(1 - a_0^2 - r^2)\\ &= \tfrac{1}{2}\sqrt{1 - a_0^2}\,da_0\,dr\,d\theta\,d\varphi\sin\theta\,\delta(r - \sqrt{1 - a_0^2}),\end{aligned}$$

which means we should generate a_0 with the distribution

$$P_\xi(a_0)\propto\sqrt{1 - a_0^2}\,e^{(2/3)\beta\xi a_0}, \tag{A24}$$

and then generate a_i uniformly on a 2-sphere of radius $\sqrt{1 - a_0^2}$. The algorithm to

evaluating a logarithm, and the square root factor is imposed by a simple accept–reject test. As the details have been specified already in the main text, we will not repeat them here.

ACKNOWLEDGMENTS

The generous support by the Rechenzentrum der Universität Karlsruhe was essential for this research to be carried out. We thank A. Schreiner and the staff of the Rechenzentrum for their assistance and encouragement. We have also benefitted from the advice of D. Sandee (CDC). We would like to thank the Deutsche Forschungsgemeinschaft for their generous support. This work is based upon research supported in part by the National Science Foundation under Grants PHY77-27084 and PHY83-13324, NASA, DOE, and the United Kingdom SERC.

REFERENCES

1. K. G. WILSON, *Phys. Rev. D* **10** (1974), 2445.
2. K. G. WILSON AND J. KOGUT, *Phys. Rep. C* **12** (1974), 75.
3. D. J. GROSS, R. D. PISARSKI, AND L. G. YAFFE, *Rev. Mod. Phys.* **53** (1981), 43.
4. R. W. B. ARDILL, M. CREUTZ, AND K. J. M. MORIARTY, *Phys. Rev. D* **27** (1983), 1956.
5. F. GUTBROD, P. HASENFRATZ, Z. KUSZT, AND I. MONTVAY, *Phys. Lett. B* **128** (1983), 415.
6. G. PARISI, R. PETRONZIO, AND F. RAPUANO, *Phys. Lett. B* **128** (1983), 418.
7. J. D. STACK, *Phys. Rev. D* **29** (1984), 1213.
8. D. BARKAI, K. J. M. MORIARTY, AND C. REBBI, *Phys. Rev. D* **30** (1984), 2201.
9. K. SYMANZIK, *in* "Mathematical Problems in Theoretical Physics" (R. Schrader *et al.*, Eds.), Springer, Berlin, 1982.
10. P. WEISS, *Nucl. Phys. B* **212** (1983), 1.
11. K. BINDER, *in* "Phase Transitions and Critical Phenomena" (C. Domb and S. Green, Eds.), Academic Press, New York, 1976.
12. E. PIETARINEN, *Nucl. Phys. B [FS3]* **190** (1981), 349.
13. N. METROPOLIS, A. W. ROSENBLUTH, M. N. ROSENBLUTH, A. H. TELLER, AND E. TELLER, *J. Chem. Phys.* **21** (1953), 1087.
14. N. CABIBBO AND E. MARINARI, *Phys. Lett. B* **119** (1982), 387.
15. M. CREUTZ, *Phys. Rev. D* **21** (1980), 2308.
16. K. C. BOWLER AND B. J. PENDLETON, *Nucl. Phys. B [FS10]* **230** (1984), 109.
17. D. BARKAI, K. J. M. MORIARTY, AND C. REBBI, *Comput. Phys. Commun.* **32** (1984), 1.
18. A. D. KENNEDY, J. KUTI, S. MEYER, AND B. J. PENDLETON, *Phys. Rev. Lett.* **54** (1985), 87.
19. A. D. KENNEDY, J. KUTI, S. MEYER, AND B. J. PENDLETON, *Phys. Lett. B* **155** (1985), 414.