Lattice Gauge Theories:

Dynamical Fermions and Parallel Computation

Thesis
submitted by

Clive Fraser Baillie

for the degree of

Doctor of Philosophy

Department of Physics
University of Edinburgh
August 1986

To Mum and Dad

Buddha *ca* 563–483 BC

All composite things decay. Strive diligently.

[His last words]

Acknowledgements

I would like to register my appreciation of the guidance and encouragement given by supervisors Ken Bowler and David Wallace throughout my research. I have benefited from numerous discussions with Simon Carson and Richard Kenway (concerning the work of Chapter 3), and Brian Pendleton. I have enjoyed collaborating with Bryan Carpenter, Ian Barbour and Philip Gibbs. It is a pleasure to thank all the staff and students in the Physics Department at Edinburgh for their help and friendship, particularly my office-mates Catherine Chalmers and Simon Hands.

The simulation reported in Chapter 3 was performed using the ICL DAPs at Edinburgh which are maintained by Edinburgh Regional Computing Centre (ERCC) and supported by SERC grants NG11849 and NG15908. The simulation reported in Chapter 4 was performed using the Gould PN9080 which is also maintained by ERCC. I am indebted to Bob Gray, Adam Hamilton, David Mercer and Jimmy Stewart for their help in using this and other Unix machines. The translator described in Chapter 5 was developed at GEC Hirst Research Centre and at ERCC. I would like to thank Steve Pass, David Scott and Bryan Stephenson, of GEC, for useful discussions.

Finally, I acknowledge the award of an SERC CASE (Co-operative Award in Science and Engineering) studentship with GEC as the industrial partner.

Declaration

The work of Chapter 2.2 was done in collaboration with Bryan Carpenter; the simulation reported in Chapter 4 was performed in collaboration with Ian Barbour and Philip Gibbs. All other work is my own, except where otherwise stated.

Published work includes the following references:

Carpenter D. B. and Baillie C. F., 1985, Nucl. Phys. B260 103

Baillie C. F., 1986, Proc. 6th Summer School on Computing Techniques in Physics (to appear in Comput. Phys. Commun.)

[redacted]

29th August 1986

Abstract

The inclusion of fermionic degrees of freedom into lattice gauge theory and aspects of parallel computation are examined.

The problem of fermion doubling and the two most popular methods for circumventing it - Wilson and Susskind fermions - are reviewed. Methods, both approximate and exact, for introducing dynamical fermions into lattice gauge theory are discussed. The chiral condensate $<\overline{\psi}\psi>$ is calculated for free Wilson and Susskind fermions with periodic and antiperiodic boundary conditions. Various "hadron" (fermion bilinear/trilinear) propagators are also calculated and finite-size effects investigated. This indicates that the propagators for free fermions are bounded above and below by periodic and antiperiodic boundary conditions in the spatial directions respectively.

The pseudofermion method is used to perform a numerical simulation of the Schwinger model (two dimensional QED) with massive Wilson fermions. This method is efficiently implemented on a highly parallel SIMD computer (the ICL DAP). The continuum Schwinger model is reviewed and the pure gauge theory, free fermions, the quenched and the dynamical model are simulated. For the quenched model the behaviour of $<\overline{\psi}\psi>/g$ as $m/g \to 0$ agrees with that predicted by Carson and Kenway; for the dynamical model $<\overline{\psi}\psi>$ varies linearly with mass for small mass.

The Lanczos algorithm is used to perform a numerical simulation of SU(2) at finite density. Finite density, or non-zero chemical potential, in lattice gauge theories is reviewed and the simulation performed in two regimes: fixed chemical potential; varying fermion mass, and fixed fermion mass; varying chemical potential. In the former, for a small chemical potential, the signal of a phase transition is observed; in the latter, at strong coupling, chiral symmetry is restored as a continuous phase transition, in agreement with a calculation by Dagotto, Moreo and Wolff.

A general FORTRAN to C translator, primarily for parallel computation, has been developed and is described in detail. This software automatically converts DAP FORTRAN programs written for the ICL Distributed Array Processor (DAP) into equivalent programs in GRID extended C which will run on the GEC Rectangular Image and Data processor (GRID). It also translates standard FORTRAN 77 into C.

Table of Contents

Table of Contents

Appendices

Chapter 1
Introduction

This chapter provides an introduction to the physics in chapters 2, 3 and 4. Chapter 5, concerning computing, has its own introduction. In Sec. 1 we describe gauge theories, which are now ubiquitous in elementary particle physics, with emphasis on QED and QCD. Then in Sec. 2 we explain how such theories are discretised on a space-time lattice, *à la* Wilson, giving lattice gauge theories. In particular, we show how quark confinement arises naturally on the lattice and investigate the renormalisation properties required in order to recover the continuum limit. Fermions are introduced in Sec. 3 and we demonstrate the fermion doubling problem with the naive lattice action before going on to explain the two most popular methods for circumventing it: Wilson and Susskind fermions. We describe what is involved in a numerical simulation of a lattice gauge theory with so-called dynamical fermions and discuss methods for doing this. These fall into two classes: approximate methods, like the hopping parameter expansion and the pseudofermion method; and exact methods, including Scalapino and Sugar's method, the block Lanczos algorithm, Weingarten and Petcher's method, and equation of motion methods. Finally, in Sec. 4 we detail the main technique used in numerical simulations of lattice gauge theories: the Monte Carlo method, outlining both the Metropolis and the heat bath versions of it.

1.1. Gauge theories

Gauge theories now dominate elementary particle physics: electromagnetic, weak and strong interactions are all based on the gauge principle. A gauge theory is a field theory whose dynamics arise from a local, or gauge, symmetry requirement.† The simplest gauge theory is quantum electrodynamics (QED) with its Abelian U(1) local symmetry. The QED Lagrangian density can actually be "derived" by requiring the free Dirac electron theory to be gauge invariant. The Lagrangian density for a free electron field $\psi(x)$ is

† For more details, see Cheng T-P. and Li L-F., 1984, Gauge theory of elementary particle physics (Clarendon Press, Oxford)

$$\mathcal{L}_0 = \bar{\psi}(x) \{ i \gamma^\mu \partial_\mu - m \} \psi(x) , \tag{1.1}$$

where γ^μ are the Dirac matrices satisfying $\{\gamma^\mu, \gamma^\nu\} = 2g^{\mu\nu}$. This clearly has a global U(1) symmetry under the phase change

$$\begin{aligned} \psi(x) &\longrightarrow \psi'(x) = e^{i\alpha} \psi(x) \\ \bar{\psi}(x) &\longrightarrow \bar{\psi}'(x) = e^{-i\alpha} \bar{\psi}(x) . \end{aligned} \tag{1.2}$$

We gauge this symmetry, that is, make it local, by introducing a space-time dependent phase change $\alpha(x)$. Then to keep (1.1) gauge invariant we must introduce into the theory a new vector, or gauge, field $A_\mu(x)$, which transforms as

$$A_\mu(x) \longrightarrow A'_\mu(x) = A_\mu(x) + \frac{1}{e} \partial_\mu \alpha(x), \tag{1.3}$$

and generalise the derivative to the so-called (gauge-)covariant derivative

$$D_\mu \equiv \partial_\mu + ie A_\mu , \tag{1.4}$$

where e is a free parameter which is identified with the charge on the electron. Thus we now have (1.1) in the form

$$\mathcal{L}'_0 = \bar{\psi} \{ i \gamma^\mu (\partial_\mu + ie A_\mu) - m \} \psi . \tag{1.5}$$

We make the gauge field a dynamical variable by adding a term involving its derivatives. The simplest such term which is gauge invariant is (with conventional normalisation)

$$-\frac{1}{4} F_{\mu\nu} F^{\mu\nu}, \tag{1.6}$$

where

$$F_{\mu\nu} \equiv \partial_\mu A_\nu - \partial_\nu A_\mu. \tag{1.7}$$

Combining (1.5) and (1.6) we obtain the QED Lagrangian density

$$\mathcal{L}_{QED} = -\frac{1}{4} F_{\mu\nu} F^{\mu\nu} + \bar{\psi} \{ i \gamma^\mu D_\mu - m \} \psi. \tag{1.8}$$

Note that the gauge field, or photon, is massless because an $A_\mu A^\mu$ term is not gauge invariant. There is no gauge field self-coupling term since the photon has zero charge (U(1) quantum number). Thus without a matter field the theory is a free field theory. This is not the case for non-Abelian gauge groups which we now turn to, leaving further discussion of QED, albeit only in two space-time dimensions, for Chap. 3.3.

Yang and Mills, 1954, extended the gauge principle to non-Abelian gauge groups, for example, SU(N). In general, for a (non-Abelian) simple Lie group $\mathcal{G}$ with generators G^a satisfying the Lie algebra

$$\left[G^a, G^b \right] = i C^{abc} G^c, \tag{1.9}$$

where C^{abc} are the totally antisymmetric structure constants, we proceed as follows. Let ψ belong to some representation of this algebra with r representation matrices T^a (a = 1,...,r), then under a group transformation

$$\psi(x) \rightarrow \psi'(x) = e^{i \underline{T} \cdot \underline{\alpha}} \psi(x) \equiv U(\alpha)\, \psi(x), \quad (1.10)$$

where the scalar product involves r-component vectors $\underline{T}$ and $\underline{\alpha}$. We make this local, as before, introducing r gauge fields $A_\mu{}^1,\ldots,A_\mu{}^r$, which transform as

$$\underline{T} \cdot \underline{A}_\mu(x) \rightarrow \underline{T} \cdot \underline{A}'_\mu(x) =$$

$$U(\alpha(x))\, \underline{T} \cdot \underline{A}_\mu(x)\, U^{-1}(\alpha(x)) - \frac{i}{g} \left[\partial_\mu U(\alpha(x)) \right] U^{-1}(\alpha(x)), \quad (1.11)$$

and defining the covariant derivative

$$D_\mu \equiv \partial_\mu - i g T^a A_\mu{}^a \quad (1.12)$$

and the second-rank tensor for the gauge fields

$$F_{\mu\nu}{}^a \equiv \partial_\mu A_\nu{}^a - \partial_\nu A_\mu{}^a + g C^{abc} A_\mu{}^b A_\nu{}^c, \quad (1.13)$$

where g is a coupling constant analogous to e in QED. Hence we obtain the Yang-Mills action

$$\mathcal{L}_{YM} = -\frac{1}{4} \underline{F}_{\mu\nu} \cdot \underline{F}^{\mu\nu} + \bar{\psi} \{ i \gamma^\mu D_\mu - m \} \psi. \quad (1.14)$$

The pure Yang-Mills term $-F_{\mu\nu}{}^a F^{\mu\nu a}/4$ contains factors that are trilinear and quadrilinear in $A_\mu{}^a$:

$$-g\,C^{abc}\,\partial_\mu A_\nu^a A^{\mu b} A^{\nu c} - \frac{g^2}{4} C^{abc} C^{ade} A_\mu^b A_\nu^c A^{\mu d} A^{\nu e}, \quad (1.15)$$

these correspond to self-couplings of the non-Abelian (massless) gauge fields. They are brought about by the non-linear terms in $F_{\mu\nu}{}^a$ (1.13), because the gauge fields $A_\mu{}^a$ themselves transform non-trivially, like the generators, as members of the adjoint representation. (Hence the number of gauge fields is equal to the number of generators of the local symmetry.) It is these non-linear terms which lead to the rich structure found in non-Abelian gauge theories.

Historically the first successful application of the Yang-Mills theory was the unified description of the weak and electromagnetic interactions in terms of the gauge theory SU(2)xU(1) (Glashow, 1961; Weinberg, 1967; Salam, 1968). This led to the prediction and subsequent discovery, at CERN, of the W (Arnison *et al*, 1983a; Banner *et al*, 1983) and Z (Arnison *et al*, 1983b; Bagnaia *et al*, 1983) intermediate vector bosons.

The strong nuclear force is also believed to be a Yang-Mills gauge theory, based on the group SU(3), known as quantum chromodynamics (QCD). QCD arose from the idea that hadrons are bound states of fundamental constituents called quarks (Gell-Mann, 1964). The notion of quarks followed from the so-called eightfold way of Gell-Mann which predicted the low energy hadron spectrum in terms of different flavours. Their existence was supported experimentally by deep inelastic lepton-nucleon scattering experiments whose cross-sections satisfied Bjorken scaling which could be interpreted using Feynman's parton model (Feynman, 1972; Bjorken and Paschos, 1969) with the partons being identified as quarks. The problem was: what binds the quarks together? The solution comes from the observation that in order to satisfy the generalised Pauli exclusion principle it is necessary to endow quarks with a hidden quantum number, known as colour, which can have three possible values. Since only colour-singlet hadrons are observed, the forces between the coloured quarks must be colour-dependent. The colour symmetry of the quark model can be gauged and we arrive at the SU(3) colour Yang-Mills theory of the strong interaction, QCD, with Langrangian density

$$\mathcal{L}_{QCD} = -\frac{1}{2}\,\mathrm{tr}\,F_{\mu\nu}F^{\mu\nu} + \sum_{k=1}^{n_f} \bar{q}_k \left(i\gamma^\mu D_\mu - m_k\right) q_k \,, \tag{1.16}$$

where

$$\begin{aligned} D_\mu &= \partial_\mu - ig A_\mu \\ F_{\mu\nu} &= \partial_\mu A_\nu - \partial_\nu A_\mu - ig\,[A_\mu, A_\nu] \\ A_\mu &\equiv \sum_{a=1}^{8} A_\mu^a \frac{\lambda^a}{2} \end{aligned} \tag{1.17}$$

with λ^a being the Gell-Mann matrices satisfying the SU(3) commutation relations

$$\left[\frac{\lambda^a}{2}, \frac{\lambda^b}{2}\right] = i f^{abc} \frac{\lambda^c}{2} \tag{1.18}$$

and the normalisation condition

$$\mathrm{tr}\left(\lambda^a \lambda^b\right) = 2\,\delta^{ab} \,. \tag{1.19}$$

The quanta associated with the 8 strongly interacting gauge fields $A_\mu{}^a$ are called gluons and q_k, $k = 1,\ldots,n_f$, are n_f flavours of quark fields. Currently it is generally thought that $n_f = 6$ with the $q_k = \{d,u,s,c,b,t\}$. QCD has the property of "asymptotic freedom" (Gross and Wilczek, 1973; Politzer, 1973), that is, its coupling strength decreases at short distances, which justifies the parton model and allows reliable calculations of the short-distance behaviour of QCD using perturbation theory. However, it is widely believed that QCD also has the property of "quark confinement", that is, at long distances the coupling strength increases keeping the quarks bound as hadrons, which should explain the hadron spectrum but means that the long-distance behaviour is non-perturbative and must be investigated using other techniques. Lattice gauge theory is such a non-perturbative technique.

1.2. Lattice gauge theory

Wilson, 1974, introduced lattice gauge theory in which the space-time continuum is discretised to provide a cut-off that regulates ultraviolet divergences by eliminating all wavelengths less than twice the lattice spacing. This formulation of field theory emphasises the deep connection with statistical mechanics: in Euclidean space the Feynman path integral formalism for a field theory is identical to the partition function of an analogous statistical mechanics system, the square of the field theoretic coupling constant being identified with the statistical mechanical temperature. The method of high temperature series expansion in statistical mechanics becomes the strong coupling expansion for field theory. There are two popular methods for introducing the lattice in field theory: the Euclidean lattice formulation (Wilson, 1974), in which both space and time are discretised and the Hamiltonian formulation (Kogut and Susskind, 1975) in which only the spatial dimensions are discretised. We shall use the Euclidean lattice formulation. The connection with ordinary Minkowski space is made via a Wick rotation ($t \rightarrow i\tau$). The simplest choice of lattice is a regular hypercubic space-time lattice of spacing a with the points, or sites, labelled by a four-vector $n = (n_1,n_2,n_3,n_4)$. Then, four-dimensional integration is replaced by a sum:

$$\int d^4x \longrightarrow a^4 \sum_n . \tag{1.20}$$

Other choices of lattice are possible - as with any cut-off prescription, the physics of the theory is independent of the details of the regulator.

Consider a field theory described by a Lagrangian density $\mathcal{L}$. Every field configuration ϕ has a corresponding lattice action

$$S(\phi) = a^4 \sum_n \mathcal{L} . \tag{1.21}$$

This is quantised using the Feynman path integral formalism (Feynman, 1948) in which the expectation value of some operator $\hat{O}$ (representing a physical observable) is given by

$$\langle \hat{O}(\phi) \rangle = \frac{1}{Z} \int \mathcal{D}\phi \, \hat{O}(\phi) \, e^{-S(\phi)}, \tag{1.22}$$

where

$$Z = \int \mathcal{D}\phi \, e^{-S(\phi)}. \tag{1.23}$$

On the lattice, there is no problem with the definition of the measure in these integrals. The functional integral is defined simply as the product of the integrals over the fields at every site of the lattice $\phi(n)$:

$$\int \mathcal{D}\phi \equiv \int \prod_n \left[d\phi(n) \right]. \tag{1.24}$$

With a finite lattice there are a finite number of integrals which means that it is possible to investigate the field theory by numerical simulations (using, for example, the Monte Carlo method - see Sec. 4) on a computer. Note also that no gauge fixing term has been included in the path integral, as this procedure (which is necessary in the continuum to control divergences resulting from integration over all gauges) is not required in numerical simulations of lattice gauge theory.

Now to construct a lattice gauge theory, we should keep the gauge symmetry explicit in the lattice formulation so that in the continuum limit we recover the Yang-Mills theory - this is what Wilson's formulation achieves. We associate elements $U_\mu(n)$ of a gauge group $\mathcal{G}$ (which we shall take as U(1) or SU(N)) with with links on the lattice joining sites n and $n+e_\mu$, where e_μ is a unit lattice vector in the μ-direction. $U_\mu(n)$ is a directed variable: in going from $n+e_\mu$ to n we use $U_\mu^\dagger(n)$. The elements for the groups we shall consider are

$$U_\mu(n) = \begin{cases} e^{i\alpha_\mu(n)}, & \text{for } U(1) \\ e^{i\frac{\sigma^i}{2}\theta^i_\mu(n)}, & \text{for } SU(2) \\ e^{i\frac{\lambda^a}{2}A_\mu{}^a(n)}, & \text{for } SU(3) \end{cases} \tag{1.25}$$

where the generators of SU(3) λ^a are defined in (1.18) and (1.19), and the generators of SU(2) σ_i are the usual Pauli matrices satisfying

$$\left[\frac{\sigma_i}{2}, \frac{\sigma_j}{2}\right] = i\,\varepsilon_{ijk}\frac{\sigma_k}{2} \quad ; \quad i,j,k = 1,2,3. \tag{1.26}$$

Local gauge symmetry corresponds to allowing an arbitrary group rotation $\Omega(n)$ at every lattice site, under which the link variables transform as

$$U_\mu(n) \longrightarrow U'_\mu(n) = \Omega(n)\,U_\mu(n)\,\Omega(n+e_\mu). \tag{1.27}$$

Thus $\Omega(n)$ defines the orientation of a local colour frame of reference at each site and $U_\mu(n)$ tells us how this orientation changes in going from one frame to the next in the direction μ. To construct an action with this local symmetry it is clear that we require products of U matrices around closed paths, for this is gauge invariant provided all SU(N) colour indices are locally contracted. The simplest such action involves the most local interaction of four U matrices around an elementary square of the lattice, called a plaquette:

$$S_G(U) = \beta \sum_{\square} \left(1 - \frac{1}{N}\,\mathrm{Re}\,\mathrm{tr}\,U_\square\right) \tag{1.28}$$

with

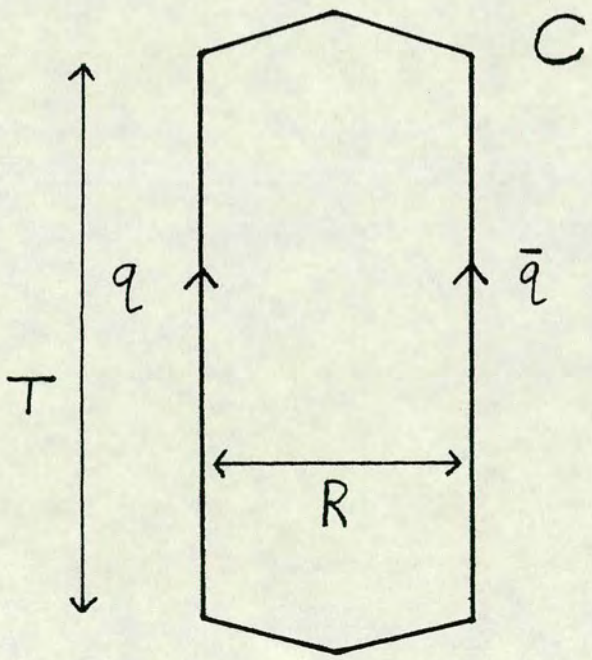
$$U_\square \equiv U_\mu(n)\,U_\nu(n+e_\mu)\,U^\dagger_\mu(n+e_\nu)\,U^\dagger_\nu(n),$$

where $\beta \equiv 2N/g^2$ for SU(N) in four dimensions, or $\beta \equiv 1/g^2a^2$ for U(1) in two dimensions (Chap. 3). The additive constant in (1.28) ensures that the action

vanishes when the group elements approach the identity. N, the dimensionality of the group matrices, is a normalisation. The trace may be performed in any representation of the group, though we will follow the usual practice and consider only the fundamental representation. By Taylor expanding the gauge field $A_\mu(n)$ and using the Baker-Campbell-Hausdorff identity, it is easily shown that this action reduces to the usual Yang-Mills action in the naive continuum limit $a \to 0$.

Equipped with a lattice gauge theory for QCD (SU(3)) we can now show that quarks are confined at long distances, that is, in the strong coupling limit. Consider the following thought experiment: i) adiabatically separate a heavy $q\text{-}\bar{q}$ pair to a distance R; ii) hold them apart there for a long time T; iii) bring them together adiabatically and annihilate. This yields the world-line C shown in Fig. 1.1.

Fig. 1.1 Illustration for the thought experiment.



The Euclidean amplitude for this process is given by the matrix element

$$\langle f | e^{-HT} | i \rangle, \tag{1.29}$$

where H is the Hamiltonian of the SU(3) gauge theory, and i and f label, respectively, the initial and final states of the $q-\bar{q}$ pair when they are a distance R apart. This can be written as the following path integral:

$$\frac{\int \mathcal{D}A_\mu \exp\left\{-S + ig\int A_\mu^a J_\mu^a d^4x\right\}}{\int \mathcal{D}A_\mu\, e^{-S}}, \tag{1.30}$$

where $J_\mu{}^a$ is external current of the heavy quarks (and we have scaled the gauge fields A_μ by the coupling g). $J_\mu{}^a$ vanishes everywhere except on the world-line of the quarks so (1.30) becomes

$$\frac{\int \mathcal{D}A_\mu \exp\left\{-S + ig\oint_c A_\mu\, dx_\mu\right\}}{\int \mathcal{D}A_\mu\, e^{-S}}. \tag{1.31}$$

As $|i\rangle$ and $|f\rangle$ are identical, and the process is static, (1.29) reduces to

$$e^{-V(R)T} \tag{1.32}$$

with the heavy quark potential V(R) defined, from (1.31), as

$$V(R) = \lim_{T\to\infty} -\frac{1}{T} \ln \left\langle \mathrm{tr}\, P\, e^{ig\oint_c A_\mu dx_\mu} \right\rangle, \tag{1.33}$$

where P denotes a path ordering of the operators. The argument of the logarithm is the continuum analogue of the Wilson loop W(C) which is defined as the expectation of the trace of a product of link variables around any closed path C:

$$W(C) \equiv \left\langle \mathrm{tr} \prod_{\mu,n \in C} U_\mu(n) \right\rangle . \tag{1.34}$$

In the strong coupling limit ($g^2 \to \infty$), the Wilson loop can be shown to be given by

$$W(C) \simeq \left(\frac{1}{g^2}\right)^{N_c} = e^{-\ln g^2 \left(\frac{\mathrm{Area}}{a^2}\right)} , \tag{1.35}$$

where N_c is the minimum number of plaquettes required to tile the surface, of area Area/a^2, bounded by the contour C. With a rectangular contour of length T and width R we therefore find, combining (1.33), (1.34) and (1.35),

$$V(R) = \sigma R \tag{1.36}$$

which defines the string tension at strong coupling:

$$\sigma = \frac{\ln g^2}{a^2} + \ldots . \tag{1.37}$$

(Higher orders in the coupling constant may be obtained by tiling the surface in a way that is not minimal - see Creutz, 1983.) Thus the potential increases linearly with distance and confines the quarks. We note that this so-called area law criterion for confinement loses its value when quarks are introduced as dynamical variables - which is in itself a good reason why lattice gauge theories should be investigated beyond the quenched approximation in which quarks are ignored - because the widely separated sources may reduce their energy by pair-production from the vacuum (the Wilson loop then measures the potential between two mesons rather than bare quarks).

Finally, the lattice, considered as an ultraviolet cut-off, must be removed by letting the lattice spacing go to zero and so recovering the continuum limit. As when removing any cut-off, physical variables should approach their observable

values. For example, the mass of a particle, m, should be independent of the lattice spacing:

$$a \frac{d}{da} m = 0. \tag{1.38}$$

Now (in four dimensions) from dimensional analysis

$$m = \frac{1}{a} f(g), \tag{1.39}$$

where f is some dimensionless function of the gauge coupling only. Thus to obtain a sensible continuum limit, as the lattice spacing $a \to 0$, $g \to g^*$, a critical value of the coupling, such that $f(g^*) = 0$. Hence the coupling is a function of lattice spacing. Moreover, the critical point $g = g^*$ must have scaling properties, that is, once the relationship between g and a has been established by fixing one physical observable, this form for g(a) must make all other observables tend to their physical values as $a \to 0$. For non-Abelian gauge theories, perturbative arguments have shown that $g = 0$ is such a scaling critical point. By combining (1.38) and (1.39) we get

$$f(g) = -\beta(g) f'(g), \tag{1.40}$$

where

$$\beta(g) \equiv -a \frac{dg}{da}. \tag{1.41}$$

This β function, which gives the relation between coupling and lattice spacing, has been calculated in perturbation theory for small g (Politzer, 1973; Gross and Wilczek, 1973) to be

$$-\beta(g) = \beta_0 g^3 + \beta_1 g^5 + \dots , \tag{1.42}$$

where for an SU(N) gauge theory (without fermions)

$$\beta_0 = \frac{11}{3}\left(\frac{N}{16\pi^2}\right) \quad ; \quad \beta_1 = \frac{34}{3}\left(\frac{N}{16\pi^2}\right)^2 . \tag{1.43}$$

Only the first two terms of the β function are universal; higher order terms are regularisation-dependent. We can now write down the relation between g and a in the form

$$a = \frac{1}{\Lambda} f(g), \tag{1.44}$$

where, from (1.40),

$$f(g) = \exp\left\{ - \int^g \frac{dg'}{\beta(g')} \right\}. \tag{1.45}$$

Hence the physical mass Λ which sets the scale for all masses in the theory is given in terms of β_0 and β_1 by

$$\Lambda = \frac{1}{a} (\beta_0 g^2)^{-\frac{\beta_1}{2\beta_0^2}} \exp\left(-\frac{1}{2\beta_0 g^2}\right) \left[1 + O(g^2)\right]. \tag{1.46}$$

It is clear from this expression that Λ does not have a perturbative expansion and consequently mass generation is a non-perturbative effect. However, once the scale is set, ratios of masses in the theory are determined (with no free parameters) as pure numbers depending only on the gauge group. The regime in which (1.46) holds is known as the asymptotic scaling region of the theory. It is possible to relate lattice calculations to ones based on continuum regularisation schemes by relating their Λ parameters (Hasenfratz and Hasenfratz, 1980).

Having established that the pure gauge theory is confining at strong coupling and that the continuum limit is reached when g = 0, we must verify that there is no phase transition in the intermediate coupling region for the phenomenon of confinement to be present in the continuum. It is known analytically (Guth, 1980) - and numerically by Monte Carlo simulation (Creutz, Jacobs and Rebbi, 1979; Lautrup and Nauenberg, 1980) - that such a transition occurs for QED (in four dimensions) and there is a critical point separating the charge confining phase from the free charge phase. This is, of course, as it should be: continuum QED is not a confining theory in four dimensions. For QCD such an analytical proof has not been found; however, it has been demonstrated numerically by Monte Carlo simulation (Creutz, 1979, 1980) that there is no phase transition in the intermediate coupling region (neither for SU(3) nor for SU(2)) - the strong coupling phase persists into the continuum and quarks are confined.

1.3. Fermions

We now have a pure gauge theory for QCD on the lattice. The next obvious step is to introduce fermions which interact with the gauge fields to produce the strongly interacting particle spectrum we see in nature. This will prove to be rather difficult due to the problem of fermion doubling.

We start from the continuum free fermion action in Euclidean space

$$S = \int d^4x \; \bar{\psi}(x) \left(\not{\partial} + m \right) \psi(x), \tag{1.47}$$

where $\not{\partial} = \gamma^\mu(\partial_\mu + ieA_\mu)$ and the γ matrices, satisfying $\{\gamma_\mu, \gamma_\nu\} = 2\delta_{\mu\nu}$, are chosen to be Hermitian: ${\gamma_\mu}^\dagger = \gamma_\mu$. This is discretised on the lattice by associating the fermion fields ψ with sites of the lattice n and making the replacement

$$\partial_\mu \psi(n) \rightarrow \frac{1}{2a}\left[\psi(n+e_\mu) - \psi(n-e_\mu)\right], \tag{1.48}$$

where e_μ is the displacement vector by one site in the μ-direction (and therefore has length a, the lattice spacing). We choose the central difference to preserve the anti-Hermitian nature of $\not{\partial}$. Hence we obtain the so-called naive lattice action for fermions

$$S = \frac{1}{2a}\sum_{n,\mu}\bar{\psi}(n)\left[\gamma_\mu \psi(n+e_\mu) - \gamma_\mu \psi(n-e_\mu)\right] + m\sum_n \bar{\psi}(n)\psi(n). \tag{1.49}$$

From this action we can calculate the lattice momentum space propagator (in the same way as we do in Chap. 2.1):

$$\tilde{G}(q) = \left[\sum_\mu \frac{i}{a}\gamma_\mu \sin q_\mu + m\right]^{-1}. \tag{1.50}$$

For massless free fermions, $\tilde{G}(q)$ has poles for $\sum_\mu \sin q_\mu = 0$, that is, for

$$q_\mu = \begin{matrix}(0,0,0,0)\\(0,0,0,\pi)\\(0,0,\pi,\pi)\\\vdots\\(\pi,\pi,\pi,\pi)\end{matrix} \tag{1.51}$$

Thus we find that is addition to the expected excitation about zero momentum, there are 15 extra modes at the edge of the Brillouin zone. The fermions have "doubled" in each dimension - so that, in general, on a d-dimensional lattice 2^d degenerate fermionic species survive in the continuum limit. To circumvent this problem we must go beyond the naive lattice action.

Before doing this we should point out that the transition from this free fermion theory to the interacting fermion and gauge theory is straightforward: the replacement $\bar{\psi}(n)\psi(n+e_\mu) \rightarrow \bar{\psi}(n)U_\mu(n)\psi(n+e_\mu)$ induces the correct gauge-covariant coupling between fermion and gauge degrees of freedom.

1.3.1. Wilson fermions

Wilson, 1977, invented a method whereby the unwanted fermion species are given a mass of order 1/a and so decouple from the theory in the continuum limit. This is done by adding to the action a term corresponding to the lattice version of the second derivative of the fermion field, multiplied by an arbitrary parameter r. This term is allowable because it is of order the cut-off and so will disappear in the continuum limit. The Wilson action is thus

$$S_W = \frac{1}{2a} \sum_{n,\mu} \bar{\psi}(n)\left[(\gamma_\mu - r\mathbb{1})\,\psi(n+e_\mu) - (\gamma_\mu + r\mathbb{1})\,\psi(n-e_\mu)\right] + m\sum_n \bar{\psi}(n)\,\psi(n). \tag{1.52}$$

This gives the propagator (Chap. 2.1.1)

$$\tilde{G}(q) = \left[\sum_\mu \left(\frac{i}{a}\gamma_\mu \sin q_\mu - \frac{r}{a}\mathbb{1}\cos q_\mu\right) + m\right]^{-1} \tag{1.53}$$

which has the following values at the values of q_μ in (1.51):

$$\tilde{G}(q) = \begin{matrix} \left(m - \frac{4r}{a}\right)^{-1}, & q_\mu = (0,0,0,0) \\ \left(m - \frac{2r}{a}\right)^{-1} & (0,0,0,\pi) \\ \vdots & \vdots \\ \left(m + \frac{4r}{a}\right)^{-1} & (\pi,\pi,\pi,\pi). \end{matrix} \qquad (1.54)$$

If we define $m = m_o + 4r/a$, where m_o is the ground state mass, then only the state corresponding to $q = (0,0,0,0)$ retains a non-zero propagator as $a \to 0$, giving us the one fermion required. Note that there are two special cases: $r = 0$ (which reduces to naive fermions) and $r = 1$ (for which $\gamma_\mu \pm 1$ act as projection operators). In the continuum limit for $r = 1$, zero mass for the lowest (free fermion) mode is given by $m_o = 0$, that is, $m = 4/a$; this is called the critical mass and denoted m_c. Wilson, 1977, has shown that in the strong coupling limit ($g^2 \to \infty$) for $r = 1$, the critical mass becomes $m_c = 2/a$. Hence in the interacting theory ($0 < g^2 < \infty$), we assume that the critical mass, in lattice units, lies in the range $4 > m_c > 2$. The actual value it assumes must be found numerically, which is one of the disadvantages of Wilson fermions. Another disadvantage is that the r-dependent terms in the action explicitly break the chiral symmetry of the massless theory. In the continuum, chiral symmetry is spontaneously broken at $m_o = 0$ dynamically generating a Goldstone boson which is taken to be the pion. Thus the use of Wilson's action relies on the observation that at some value of m_c the mass of the lowest pseudoscalar in the theory approaches zero, suggesting that it is indeed the Goldstone boson.

1.3.2. Susskind fermions

Susskind, 1977, proposed reducing the fermion degeneracy by "thinning" the degrees of freedom, distributing them on sub-lattices. To derive this we follow Kawamoto and Smit, 1981, and spin-diagonalise the naive lattice action. Define a field $\chi(n)$ as follows:

$$\psi_{(n)} \doteq T_{(n)}\, \chi_{(n)}$$
$$\bar{\psi}_{(n)} = \bar{\chi}_{(n)}\, T^{+}_{(n)} \qquad (1.55)$$

with

$$T_{(n)} \equiv \gamma_1^{n_1}\, \gamma_2^{n_2}\, \gamma_3^{n_3}\, \gamma_4^{n_4} \quad ; \quad T_{(n)}\, T^{+}_{(n)} = 1,$$

where the four-vector labelling lattice sites, n = (n_1,n_2,n_3,n_4). Rewriting the action (1.49) in terms of χ yields

$$S = \frac{1}{2a} \sum_{n,\mu} \bar{\chi}^{\alpha}_{(n)}\, \eta_{\mu(n)} \left[\chi^{\alpha}_{(n+e_\mu)} - \chi^{\alpha}_{(n-e_\mu)} \right]$$
$$+ \; m \sum_{n} \bar{\chi}^{\alpha}_{(n)}\, \chi^{\alpha}_{(n)}, \qquad (1.56)$$

where the phase factor

$$\eta_{\mu(n)} = (-1)^{n_1 + n_2 + \ldots + n_{\mu-1}}, \qquad (1.57)$$

and the index α labels the Dirac components of the original fermion fields, running from 1 to 4. Thus the naive action has been diagonalised in spin space, that is, it has completely decoupled into 4 identical spinor copies. All but one of them is thrown away reducing the fermion degeneracy, in d dimensions, from 2^d to $2^{d/2}$. This diagonalisation may, equivalently, be carried out in momentum space (Sharatchandra, Thun and Weisz, 1981). Thus the Susskind action is

$$S_S = \frac{1}{2a} \sum_{n,\mu} \bar{\chi}(n)\, \eta_\mu(n) \left[\chi(n+e_\mu) - \chi(n-e_\mu) \right] + m \sum_n \bar{\chi}(n)\chi(n) \tag{1.58}$$

which yields the propagator (Chap. 2.1.2)

$$\hat{G}(q) = \left[\sum_\mu \frac{i}{a} \eta_\mu \sin q_\mu + m \right]^{-1} \tag{1.59}$$

with the same poles as the naive propagator, the difference now being that there are only $1/2^{d/2}$ times the number of degenerate fermions. We see from (1.59) that translational invariance by one lattice spacing is lost but that translational invariance by two lattice spacings in a given direction is retained. This is a reflection of the fact that the physical fermion fields should now be identified with combinations of the Susskind fields around a 2^d hypercube (Kluberg-Stern, Morel, Napoly and Petersson, 1983). Although having the disadvantage of more than one fermion, Susskind fermions have the advantage of preserving some chiral symmetry at finite lattice spacing.

Neither Wilson nor Susskind fermions fulfil our hope of obtaining a lattice gauge theory with just one fermion and with the required chiral symmetry. This is a consequence of the Nielsen-Ninomiya, 1981, no-go theorem which essentially says that chiral symmetry must be (at least partly) broken if one wants to avoid fermion doubling with a lattice action which is bilinear in the fermion fields, has exact gauge invariance and has only finite range interactions. Hence the only reasonable way to achieve our goal is to choose a non-local action - this has been done by Drell, Weinstein and Yankielowicz, 1976, using the so-called SLAC derivative, but being highly non-local is of no use in numerical simulations and moreover appears to fail to recover locality, and Lorentz invariance, in the continuum limit (Karsten and Smit, 1978, 1979). Hence in the following we use only Wilson or Susskind fermions. We shall also henceforth take the lattice spacing a = 1.

1.3.3. Numerical simulations

To summarise, we have derived the standard Euclidean action for a lattice gauge theory with fermions:

$$S(U,\bar{\Psi},\Psi) = S_G(U) + S_F(U,\bar{\Psi},\Psi), \tag{1.60}$$

where S_G is given by (1.28), and S_F is taken to be either S_W (1.52) or S_S (1.58) (including gauge fields) - written generically as

$$S_F(U,\bar{\Psi},\Psi) = \sum_{n,m} \bar{\Psi}(n)\, M(U)(n,m)\, \Psi(m), \tag{1.61}$$

where $M(U)(n,m) \equiv \not{\partial}(U)(n,m) + m\delta(n,m)$, and $\not{\partial}$ is the Dirac operator appropriate to Wilson or Susskind fermions. Physical observables are obtained as before from (1.22) and (1.23) but now with the full action (1.60). We wish to calculate these observables from numerical simulations and must therefore eliminate the fermionic variables $\bar{\psi}$, ψ which are anticommuting elements of a Grassmann algebra rather than numbers. This can be done analytically using the standard Matthews-Salam, 1954, 1955, formulae

$$\int \mathcal{D}\bar{\Psi}\mathcal{D}\Psi\, e^{-S(U,\bar{\Psi},\Psi)} = \det[M(U)]\, e^{-S_G(U)}$$

$$\int \mathcal{D}\bar{\Psi}\mathcal{D}\Psi\, \bar{\Psi}(n)\Psi(m)\, e^{-S(U,\bar{\Psi},\Psi)} = M^{-1}(U)(m,n)\, \det[M(U)]\, e^{-S_G(U)} \tag{1.62}$$

Hence

$$\langle \hat{O} \rangle = \frac{1}{Z} \int \mathcal{D}U \; \hat{O}|_{\{U\}} \det[M(U)] e^{-S_G(U)}, \quad (1.63)$$

where

$$Z = \int \mathcal{D}U \, e^{-S_{eff}(U)} \quad (1.64)$$

with the effective action

$$S_{eff}(U) \equiv S_G(U) - tr \ln[M(U)]. \quad (1.65)$$

$\hat{O}|_{\{U\}}$ represents the expectation value of the operator $\hat{O}$ in the background of the fixed gauge field configuration {U}. Unfortunately, this purely bosonic action is still no good for numerical simulations because it is highly non-local due to the determinant of the Dirac operator. This determinant represents the contribution to the action coming from closed fermion loops. The simplest way to proceed is to ignore fermion loops and work in the so-called quenched approximation. However, we wish to investigate the effects of fermions in lattice gauge theories so we must retain the determinant and use the unquenched theory with dynamical fermions. To deal with this non-local determinant many methods have been developed, most of which use of the sparse nature of M (the Dirac operator couples only to nearest neighbours so that M is essentially tridiagonal, although periodic or antiperiodic boundary conditions on the fermion fields introduce non-zero elements in the corners of the matrix). We shall briefly review some of these methods before going on to describe how numerical simulations are performed using the Monte Carlo method.

1.3.4. Dynamical fermions

In Monte Carlo numerical simulations what one requires to calculate is the change in effective action

$$\Delta S_{eff} \equiv S_{eff}(U+\delta U) - S_{eff}(U) \qquad (1.66)$$

which, from (1.65), is

$$e^{-\Delta S_{eff}} = e^{-\Delta S_G} \det\left[1 + M^{-1}(U)\,\delta M(U)\right]. \qquad (1.67)$$

We shall firstly discuss two approximate methods for calculating this and then go on to describe some exact methods.

In the hopping parameter expansion for Wilson fermions (Hasenfratz and Hasenfratz, 1981; Lang and Nicolai, 1982; Stamatescu, 1982; Montvay, 1984) we write M(U) = 1 − KB(U) so that

$$S_{eff} = S_G + \sum_j \frac{K^j}{j} \operatorname{tr} B^j, \qquad (1.68)$$

where the hopping parameter

$$K \equiv \frac{1}{2m} = \frac{1}{2m_0 + 8}, \qquad (1.69)$$

is small. The trace over Dirac and colour of B^j, giving the contribution from all closed fermion paths of order j, is calculated on the lattice. This expansion is analogous to the high temperature series expansion is statistical mechanics, in many respects. The hopping parameter is proportional to the amplitude for moving a fermion by one lattice spacing, and the order of the expansion is the length of the fermion paths considered. As long as the maximum order of the expansion is comparable with the size of a hadron in lattice units, the change in the effective action should be fairly accurate. This method has been used to calculate, to 32nd order, ground state meson and baryon masses in QCD on an 8^4 lattice with Wilson fermions (Langguth and Montvay, 1984). Kuti, 1982, modified the hopping parameter expansion so that instead of summing all the

closed fermion paths order by order, they are generated stochastically. This means that the fermions perform random walks on the lattice. The advantage of this is that, for a given statistical accuracy, the number of walks required does not depend on the size of the lattice. The main problem is to correctly choose the transition and stop probabilities of the walk - if they are not chosen correctly then most of the time is spent generating irrelevant paths and the convergence will be slow.

Another approximate method is the pseudofermion method (Fucito, Marinari, Parisi and Rebbi, 1981) which has been widely used. This is described in detail in Chap. 3.1. Essentially it involves using a Monte Carlo technique to calculate M^{-1} appearing in (1.67), the approximation being that δU is taken to be small, the effective action linearised and terms of order δU^2 neglected. The advantage of this method is that the computer time required is independent of lattice size, being proportional to the number of pseudofermion iterations, or sweeps, needed to achieve a desired statistical accuracy. Moreover, as we shall see in Chap. 3.2, the technique is ideally suited to implementation on a parallel computer. The main problem is that the systematic error introduced by throwing away terms of order δU^2 must be minimised by keeping δU small thus reducing the convergence rate.

We now turn to the first exact method for calculating (1.67) which was derived, and tested on a simple one-dimensional model, by Scalapino and Sugar, 1981. (This method was also obtained, and used for the massless Schwinger model, by Duncan and Furman, 1981.) This requires an initial knowledge of the entire fermion Green's function M^{-1} and then makes use of the fact that a change in the gauge variable on a single link induces changes in M only for those elements near the link. This means that $\delta M(U)(i,j)$ is non-zero only for a small number L of values of i and j; hence the determinant in (1.67) is effectively that of an LxL matrix only. The entire matrix M^{-1} is stored (which is a big problem for large lattices as the size of this matrix is proportional to the square of the lattice size) between iterations and updated according to the identity (rank annihilation)

$$(M+\delta M)^{-1}(i,j) = M^{-1}(i,j) - \frac{M^{-1}(i,k)\,\delta M(k,l)\,M^{-1}(l,j)}{1+\delta M(k,l)\,M^{-1}(l,k)}, \quad (1.70)$$

where the indices k and l are summed over the L non-vanishing values of δM_{kl}. (Rounding errors, which will cause M^{-1} to stray from its true value after many iterations may be reduced by periodically carrying out the correction procedure

$$M^{-1} \longrightarrow 2M^{-1} - M^{-1}MM^{-1} \quad (1.71)$$

which effectively renormalises the product $M^{-1}M$ to unity.) Scalapino and Sugar, 1981, admit that their method is too slow to be used for large, multi-dimensional lattices - it takes too long to update M^{-1}, even using (1.70) - but go on to point out that by dividing the lattice of N sites into P blocks with N/P sites per block, only a (N/P)x(N/P) sub-matrix of M^{-1} need be calculated within the block and this would be quick to update. Moreover, the sub-matrix could be calculated using an efficient method - for example, the Lanczos (or, equivalently, conjugate gradient) algorithm.

Combining these two ideas leads to the block Lanczos algorithm (Barbour *et al*, 1985b), discussed in detail in Chap. 4.1, in which the blocks correspond to hypercubes of 2^4 sites. We perform the Monte Carlo simulation by visiting hypercubes of the lattice in turn, iterating on each one a few times to bring it into local equilibrium and then moving on to the next one. The main advantage of the Lanczos algorithm is that it works well at small fermion mass, unlike the pseudofermion method, for example, which has poor convergence for this. The disadvantage of this algorithm is that its computation time increases dramatically with lattice size.

Another exact method for evaluating M^{-1} is that due to Weingarten and Petcher, 1981. They write M(U) = 1 - KB(U), where K is the hopping parameter, and consider a system with two identical fermion flavours so that the fermion contribution to the effective action may be written as

$$\left[\det(1-KB)\right]^2 = \int \mathcal{D}Q\, e^{-\frac{1}{2}\left|(1-KB)^{-1}Q\right|^2} \qquad (1.72)$$

The usefulness of this depends upon an efficient algorithm for calculating $(1 - KB)^{-1}Q$. Weingarten and Petcher, 1981, use Gauss-Seidel iteration: if χ is defined as $(1 - KB)^{-1}Q$ then by rearranging we have

$$\chi = KB\chi + Q \qquad (1.73)$$

which may be iterated until a satisfactory value for χ is obtained. Hamber, 1981, similarly solves this equation for χ but by using Gaussian iteration. In both versions the natural initial vector for any iteration is the vector χ that resulted from the previous iteration. However, these iterations must be carried out many times, in principle for every updating step - this rapidly becomes prohibitive for larger lattices.

Finally, we shall mention, for completeness, the recent development of so-called equation of motion methods which can be used for simulating lattice gauge theories with dynamical fermions. In these methods the average over the fields in (1.22) is replaced by an average over a fictitious time evolution. This evolution can be stochastic or deterministic. In the stochastic method (Parisi and Wu, 1981; Ukawa and Fukugita, 1985; Batrouni *et al*, 1985), one introduces a Gaussian white noise function $\eta(\tau)$, normalised by $\langle\eta(\tau)\eta(\tau')\rangle = 2\delta(\tau-\tau')$, and defines the time dependence of ϕ by the Langevin equation

$$\dot{\phi}(\tau) = -S'(\phi) + \eta(\tau) \qquad (1.74)$$

so that

$$\langle \hat{O}(\phi) \rangle = \overline{\hat{O}(\phi)} = \lim_{T \to \infty} \frac{1}{T} \int_0^T d\tau \, \hat{O}(\phi(\tau)). \tag{1.75}$$

In the deterministic method, $S(\phi)$ is interpreted as the potential energy (per unit mass) for a classical dynamics governed by Newton's law:

$$\ddot{\phi}(t) = - S'(\phi). \tag{1.76}$$

The conjugate momentum $\pi = \dot{\phi}$ and Hamiltonian $H(\pi,\phi) = \pi^2/2 + S(\phi)$ are then introduced so that for the microcanonical method (Callaway and Rahman, 1982, 1983; Polonyi and Wyld, 1983), the average (1.75) becomes

$$\overline{\hat{O}(\phi)} = \frac{\int \mathcal{D}\phi \, \mathcal{D}\pi \, \delta(E - H(\pi,\phi)) \, \hat{O}(\pi,\phi)}{\int \mathcal{D}\phi \, \mathcal{D}\pi \, \delta(E - H(\pi,\phi))}, \tag{1.77}$$

where the integrals are over the (2N−1)-dimensional hypersurface of constant energy defined by $H(\pi,\phi) = E$. The microcanonical and Langevin methods complement each other in that the former has a smooth trajectory through phase space and therefore moves quickly but may be non-ergodic, whereas the latter is ergodic but jumps around in phase space (following a random walk) advancing slowly. This observation lead Duane, 1985, to construct a hybrid (canonical) method which is essentially microcanonical most of the time but every now and then has a Langevin "kick" to some other part of phase space. Thus we have the advantage of microcanonical's speed in exploring phase space, made ergodic by Langevin's "kicks". To conclude, the advantage of these equation of motion methods is that one update, updates the whole system – thus avoiding the slowing down with increasing system size found in other methods.

1.4. Monte Carlo

As we have already seen (in Sec. 2), the expectation value of an operator representing a physical observable in a field theory is given by the functional integral

$$\langle \hat{O} \rangle = \frac{\int \mathcal{D}\phi \; \hat{O}(\phi) \; e^{-S(\phi)}}{\int \mathcal{D}\phi \; e^{-S(\phi)}} , \tag{1.78}$$

where ϕ denotes generically the dynamical field variables in the theory. The idea of the Monte Carlo method is to replace this integral by an average over field configurations C_i:

$$\langle \hat{O} \rangle = \frac{\sum_i \hat{O}(C_i)}{\sum_i 1} = \frac{1}{N} \sum_i \hat{O}(C_i) . \tag{1.79}$$

The mean value converges to $\langle \hat{O} \rangle$ as $N \to \infty$ with statistical errors which fall as $N^{-1/2}$. These field configurations should be configurations which significantly contribute to the average, that is, they should be typical of thermal equilibrium in the statistical analogy, distributed with the Boltzmann factor e^{-S}. The Monte Carlo method is designed to generate such a set of configurations. It begins with some arbitrary initial configuration and from this generates a sequence of configurations, such that, once statistical equilibrium is reached, the probability of finding any configuration C, $p_{eq}(C)$, is proportional to $e^{-S(C)}$. The passage from one configuration to the next is determined by the transition matrix $P(C \to C')$ satisfying the constraints of a probability:

$$P(C \to C') \geq 0$$

$$\sum_{C'} P(C \to C') = 1. \tag{1.80}$$

If after n steps we have a configuration C with probability $p_n(C)$ then

$$p_{n+1}(C') = \sum_{C} P(C \to C')\, p_n(C) \tag{1.81}$$

so we may write

$$p_{n+1}(C') - p_n(C') = \sum_{C} P(C \to C')\, p_n(C) - \sum_{C} P(C' \to C)\, p_n(C'). \tag{1.82}$$

An obvious condition on P is that it leaves an equilibrium configuration in equilibrium, hence from (1.82) we have

$$\sum_{C} P(C \to C')\, p_{eq}(C) = \sum_{C} P(C' \to C)\, p_{eq}(C'). \tag{1.83}$$

A sufficient (but not necessary) condition for (1.83) to hold is equality term by term, that is, each step of the transition matrix satisfies detailed balance:

$$P(C \to C')\, e^{-S(C)} = P(C' \to C)\, e^{-S(C')}. \tag{1.84}$$

Then $p_n(C) \to p_{eq}(C)$ as $n \to \infty$. The detailed balance condition does not uniquely determine the transition probabilities; the two most popular choices lead to the Metropolis and the heat bath algorithms.

1.4.1. Metropolis algorithm

The Metropolis algorithm (Metropolis *et al*, 1953) is often used for updating the gauge fields in the Monte Carlo simulation of a lattice gauge theory. Consider

a gauge field configuration {U}. From this we wish to generate a new configuration {U'} by updating a single link $U_\mu(n)$. This is done by selecting arbitrarily a new variable $\tilde{U}_\mu(n)$ giving a new configuration $\{\tilde{U}\}$, and calculating the change in action

$$\Delta S = S(\{\tilde{U}\}) - S(\{U\}). \tag{1.85}$$

If $\Delta S \leq 0$, the change is accepted and we have $U'_\mu(n) = \tilde{U}_\mu(n)$; $\{U'\} = \{\tilde{U}\}$. If $\Delta S > 0$, the new configuration is accepted with the probability $e^{-\Delta S}$. In practice this is done by generating a pseudo-random number r in the interval [0,1] with uniform probability distribution. If $r \leq e^{-\Delta S}$ the change is accepted: $\{U'\} = \{\tilde{U}\}$; otherwise it is rejected: $\{U'\} = \{U\}$. This means that

$$P(\{U\} \to \{U'\}) = \begin{cases} 1, & \text{if } S(\{U\}) \geq S(\{U'\}); \\ e^{-[S(\{U'\}) - S(\{U\})]}, & \text{if } S(\{U\}) < S(\{U'\}) \end{cases} \tag{1.86}$$

so detailed balance (1.84) is satisfied:

$$\frac{P(\{U'\} \to \{U\})}{P(\{U\} \to \{U'\})} = e^{S(\{U'\}) - S(\{U\})}. \tag{1.87}$$

1.4.2. Heat bath algorithm

The heat bath algorithm (Yang, 1963) is also used for updating gauge fields (Creutz, 1980b; Cabibbo and Marinari, 1982) but we shall use it for updating the pseudofermion variables in the pseudofermion method. It simply replaces each variable with a new one selected randomly with a probability given by the exponential of minus the resulting action. Thus P(C → C') is independent of C, being proportional to the Boltzmann factor for C', so that detailed balance is

automatically satisfied.

Explicitly, for the pseudofermion method, this works as follows. The pseudofermion action, which is discussed in detail in Chap. 3 - see (3.12) and (3.14), is quadratic in both the real part $\phi_R(n)$ and the imaginary part $\phi_I(n)$ of the complex pseudofermion variable $\phi(n)$, that is,

$$S_{pf} = \begin{aligned} & a(n)\,\phi_R^2(n) + 2\,b_R(n)\,\phi_R(n) \\ & + a(n)\,\phi_I^2(n) + 2\,b_I(n)\,\phi_I(n)\,, \end{aligned} \tag{1.88}$$

where a(n) comes from the part of the action coupling $\phi(n)$ to itself and $b(n) = b_R(n) + ib_I(n)$ comes from the coupling of $\phi(n)$ to its nearest and next nearest neighbours. Now at equilibrium the pseudofermion variables are distributed with the Boltzmann factor $\exp(-S_{pf})$ which means that real and imaginary parts of $\phi(n)$ are separately distributed according to the Gaussian distribution

$$e^{-\frac{1}{2\sigma^2(n)}\left[\phi(n) - \bar{\phi}(n)\right]^2} \tag{1.89}$$

with

$$\sigma(n) = \frac{1}{\sqrt{2a(n)}}\,; \quad \bar{\phi}_{R,I}(n) = -\frac{b_{R,I}(n)}{a(n)}\,. \tag{1.90}$$

The heat bath algorithm for the pseudofermions thus consists of generating two pseudo-random numbers r_1 and r_2 with Gaussian distribution $N(\sigma^2=1, \bar{\phi}=0)$, that is, $\exp(-r_{1,2}^2/2)$, and rescaling them to obtain new pseudofermion variables with correct Boltzmann distribution:

$$\phi'_{R,I}(n) = \frac{1}{\sqrt{2a(n)}}\,r_{1,2} - \frac{b_{R,I}(n)}{a(n)}\,. \tag{1.91}$$

Chapter 2
Free Fermions

When performing numerical simulations of lattice gauge theories, it is worthwhile looking at free fermions as a check on algorithms so in this chapter we shall investigate some aspects of free fermions on a lattice. In Sec. 1 we calculate $<\bar{\psi}\psi>$ for free fermions - both Wilson and Susskind with periodic and antiperiodic boundary conditions - and see how it changes with lattice size. In Sec. 2 we calculate various "hadron" (fermion bilinear/trilinear) propagators for free fermions and investigate finite-size effects.

2.1. $<\bar{\psi}\psi>$

When calculating $<\bar{\psi}\psi>$ in numerical simulations of lattice gauge theories, one usually subtracts out the free fermion chiral condensate, which we denote $<\bar{\psi}\psi>_0$. We shall be investigating the Schwinger model with Wilson fermions (in Chap. 3.3) and SU(2) with Susskind fermions (in Chap. 4.3) and will therefore require $<\bar{\psi}\psi>_0$ for both Wilson fermions in two dimensions and Susskind fermions in four dimensions. In this section we detail the calculation of this on a lattice.

2.1.1. Wilson fermions

The lattice Green's function G(n;m) for free Wilson fermions (all gauge fields set to unity) satisfies

$$\sum_{\mu} \frac{1}{2} \left\{ (\gamma_\mu - r\mathbb{1})\, G(n+e_\mu;0) - (\gamma_\mu + r\mathbb{1})\, G(n-e_\mu;0) \right\} + m\, G(n;0) = \delta(n;0), \tag{2.1}$$

where n denotes a site, μ denotes a direction, e_μ is a unit vector in the μ-direction and γ_μ are the Dirac matrices. We shall work in d dimensions on a lattice with N^d sites. If we define

$$G(n;0) = \frac{1}{N^d} \sum_{q} e^{iq.n}\, \tilde{G}(q) \tag{2.2}$$

with

$$q_\mu = \frac{2\pi k}{N} \quad ; \quad k = 0, 1, \ldots, N-1$$

then, the Fourier transform,

$$\tilde{G}(q) = \sum_{n} e^{-iq.n}\, G(n;0) \tag{2.3}$$

and

$$\delta(n;0) = \frac{1}{N^d} \sum_{q} e^{iq.n} . \tag{2.4}$$

By substituting (2.2) and (2.4) into (2.1) we obtain

$$\tilde{G}(q) = \frac{\sum_{\mu} (-i\gamma_\mu \sin q_\mu - r\mathbb{1} \cos q_\mu) + m\mathbb{1}}{\sum_{\mu} \sin^2 q_\mu + \left(m - \sum_{\mu} r \cos q_\mu\right)^2} , \tag{2.5}$$

where $q_\mu = q.e_\mu$. Now $\langle \bar{\psi}\psi \rangle_0$ is the quark propagator at zero space-time separation:

$$\langle \bar{\psi}\psi \rangle_0 = Tr\ G(0;0), \tag{2.6}$$

where Tr is trace over the Dirac and the colour indices, which from (2.2) with (2.5) becomes (for N_c colours of quark)

$$\langle \bar{\psi}\psi \rangle_0 = N_c \frac{d}{N^d} \sum_q \frac{\sum_\mu -r\cos q_\mu + m}{\sum_\mu \sin^2 q_\mu + (m - \sum_\mu r\cos q_\mu)^2}, \tag{2.7}$$

since $Tr\gamma_\mu = 0$. This corresponds to periodic boundary conditions; for antiperiodic boundary conditions we replace q_μ with $q_\mu + \pi/N$.

For r = 1, single colour (N_c = 1) Wilson fermions in two dimensions on a lattice with N = 64, we obtain Fig. 2.1 showing the behaviour of $<\bar{\psi}\psi>_0$ with Wilson mass parameter. The zero mode in the propagator for the periodic case gives the expected divergence in $<\bar{\psi}\psi>_0$ at the critical mass

$$m_c = \sum_\mu r = dr = 2. \tag{2.8}$$

$<\bar{\psi}\psi>_0$ obtained with antiperiodic boundary conditions at various values of the mass are listed in Table 2.1.

Fig. 2.1 $\langle\bar{\psi}\psi\rangle_0$ for r = 1 Wilson fermions on 64^2 lattice with periodic (solid line) and antiperiodic (dotted line) boundary conditions.

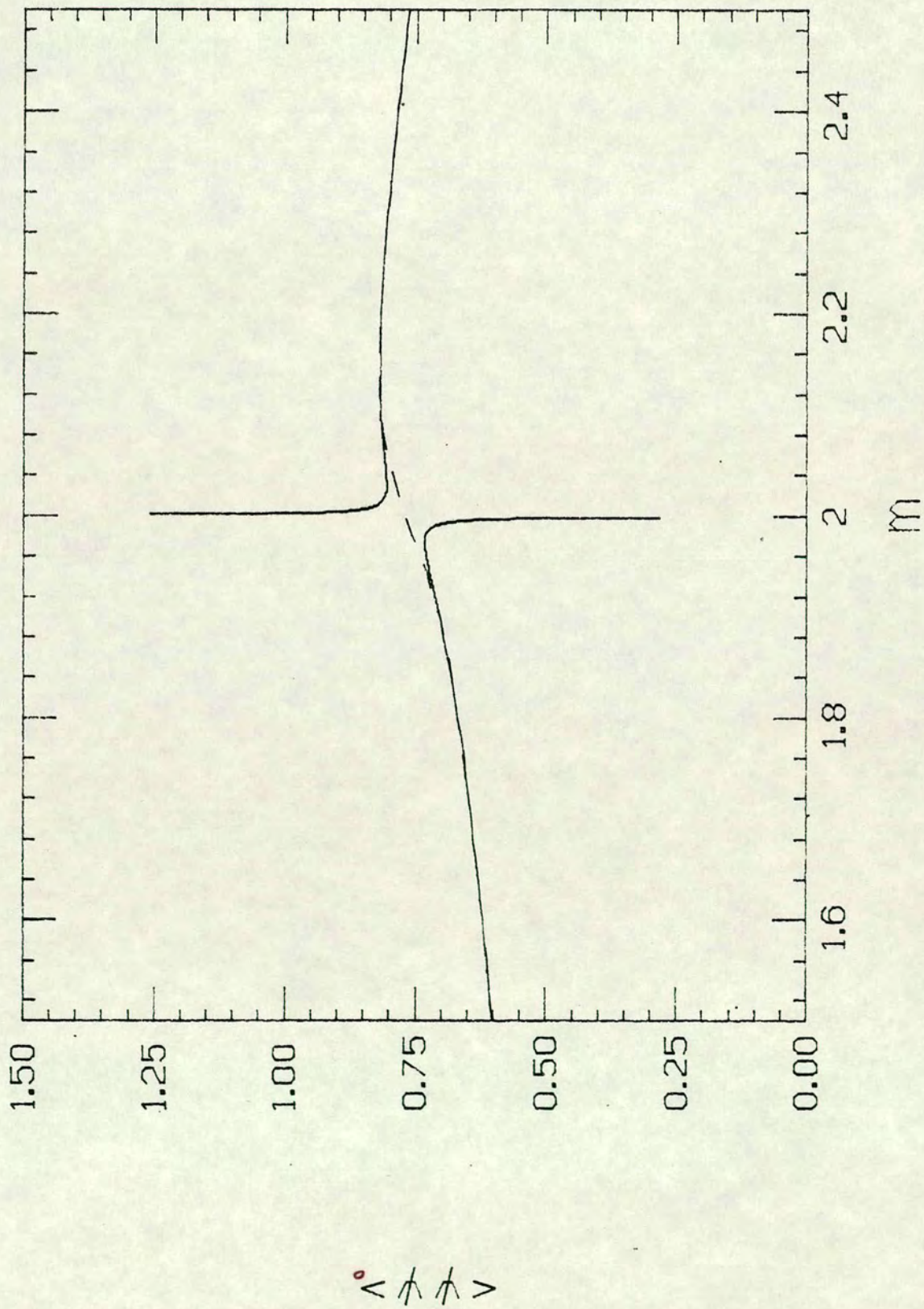


Table 2.1

$<\bar{\psi}\psi>_0$ at various masses for r = 1 Wilson fermions on 64^2 lattice with antiperiodic boundary conditions.

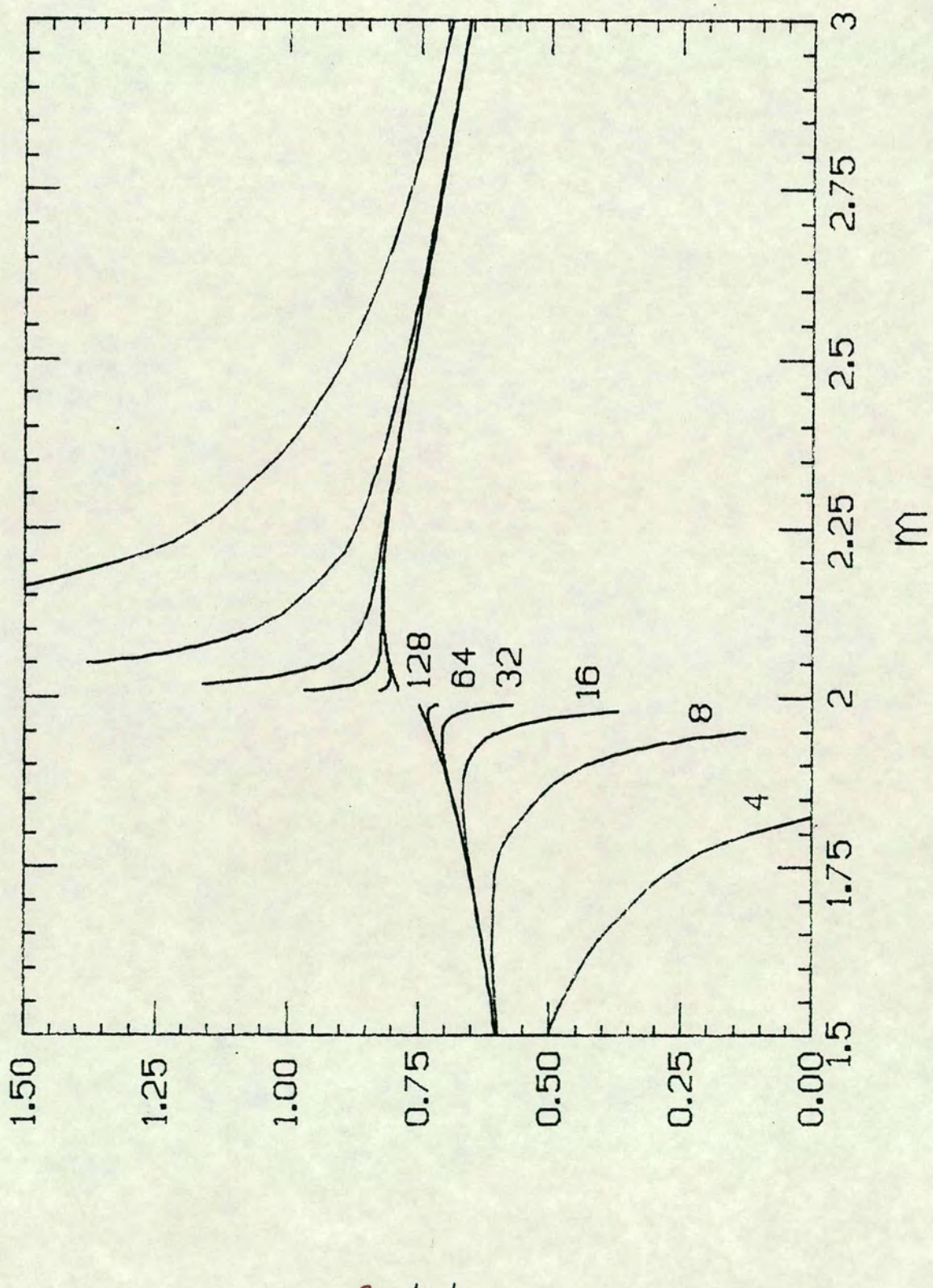
m	$<\psi\psi>_0$
1.0	0.5173
1.1	0.5341
1.2	0.5505
1.3	0.5667
1.4	0.5834
1.5	0.6009
1.6	0.6201
1.7	0.6420
1.8	0.6683
1.9	0.7032
2.0	0.7698
2.1	0.8167
2.2	0.8167
2.3	0.8033
2.4	0.7842
2.5	0.7626
2.6	0.7402
2.7	0.7179
2.8	0.6960
2.9	0.6749
3.0	0.6546

They will be used in Chap. 3.3 for the Schwinger model. We can vary N to see how $<\bar{\psi}\psi>_0$ changes with lattice size; the result is shown in Fig. 2.2 for the more dramatic case of periodic boundary conditions. It appears that lattice sizes of 64^2 or more are close to the continuum, for free fermions at least.

2.1.2. Susskind fermions

The lattice Green's function for free Susskind fermions satisfies

Fig. 2.2 $\langle\bar{\psi}\psi\rangle_0$ for r = 1 Wilson fermions on N^2 lattices with periodic boundary conditions.



$$\sum_{\mu} \frac{1}{2} \eta_{\mu} \{ G(n+e_{\mu};0) - G(n-e_{\mu};0) \} + m\, G(n;0) = \delta(n;0), \tag{2.8}$$

where the phase factors

$$\eta_1 = 1, \quad \eta_2 = (-1)^{n_1}, \ldots, \eta_d = (-1)^{n_1+n_2+\ldots+n_{d-1}}.$$

Fourier transforming this in exactly the same way as for Wilson fermions yields

$$\tilde{G}(q) = \frac{\sum_{\mu} -i\eta_{\mu} \sin q_{\mu} + m}{\sum_{\mu} \sin^2 q_{\mu} + m^2} \tag{2.9}$$

so that

$$\langle \bar{\psi}\psi \rangle_0 = N_c \frac{1}{N^d} \sum_{q} \frac{m}{\sum_{\mu} \sin^2 q_{\mu} + m^2}, \tag{2.10}$$

since $\sum_q \sin q_\mu = 0$. Again for antiperiodic boundary conditions we replace q_μ with $q_\mu + \pi/N$. We notice that $\langle\bar{\psi}\psi\rangle_0$ for Susskind fermions is precisely 1/d times $\langle\bar{\psi}\psi\rangle_0$ for Wilson fermions with r = 0, as expected.

For single colour Susskind fermions in four dimensions on a lattice with N = 4 we obtain Fig. 2.3. Now the zero mode in the propagator occurs at m = 0. $\langle\bar{\psi}\psi\rangle_0$ obtained with antiperiodic boundary conditions at various values of the mass are listed in Table 2.2.

Fig. 2.3 $<\bar{\psi}\psi>_0$ for Susskind fermions on 4^4 lattice with periodic (solid line) and antiperiodic (dotted line) boundary conditions.

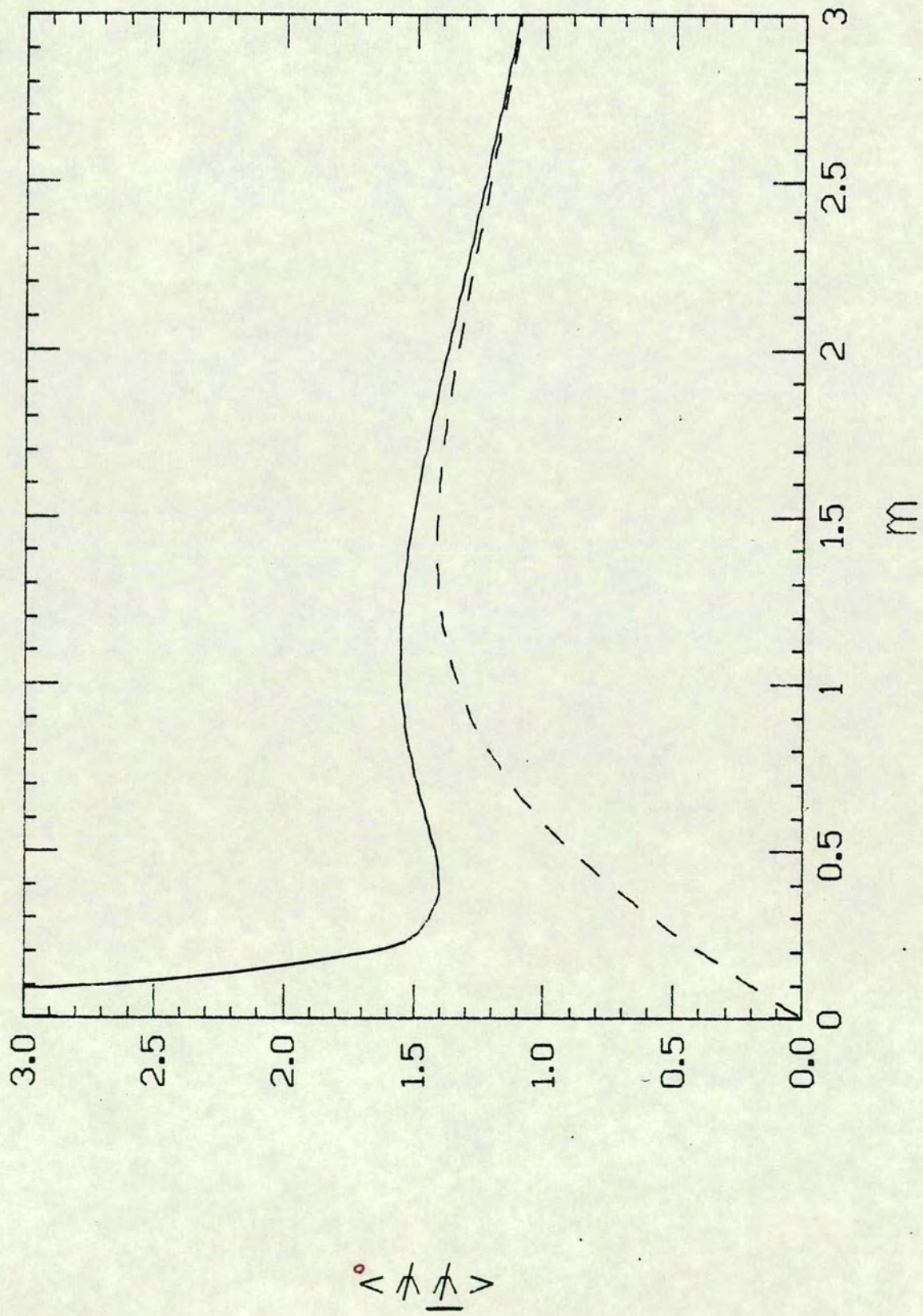


Table 2.2

$<\bar{\psi}\psi>_0$ at various masses for Susskind fermions on 4^4 lattice with antiperiodic boundary conditions.

m	$<\psi\psi>_0$
0.1	0.1990
0.2	0.3922
0.3	0.5742
0.4	0.7407
0.5	0.8889
0.6	1.0169
0.7	1.1245
0.8	1.2121
0.9	1.2811
1.0	1.3333
1.1	1.3707
1.2	1.3953
1.3	1.4092
1.4	1.4141
1.5	1.4118
1.6	1.4035
1.7	1.3906
1.8	1.3740
1.9	1.3547
2.0	1.3333

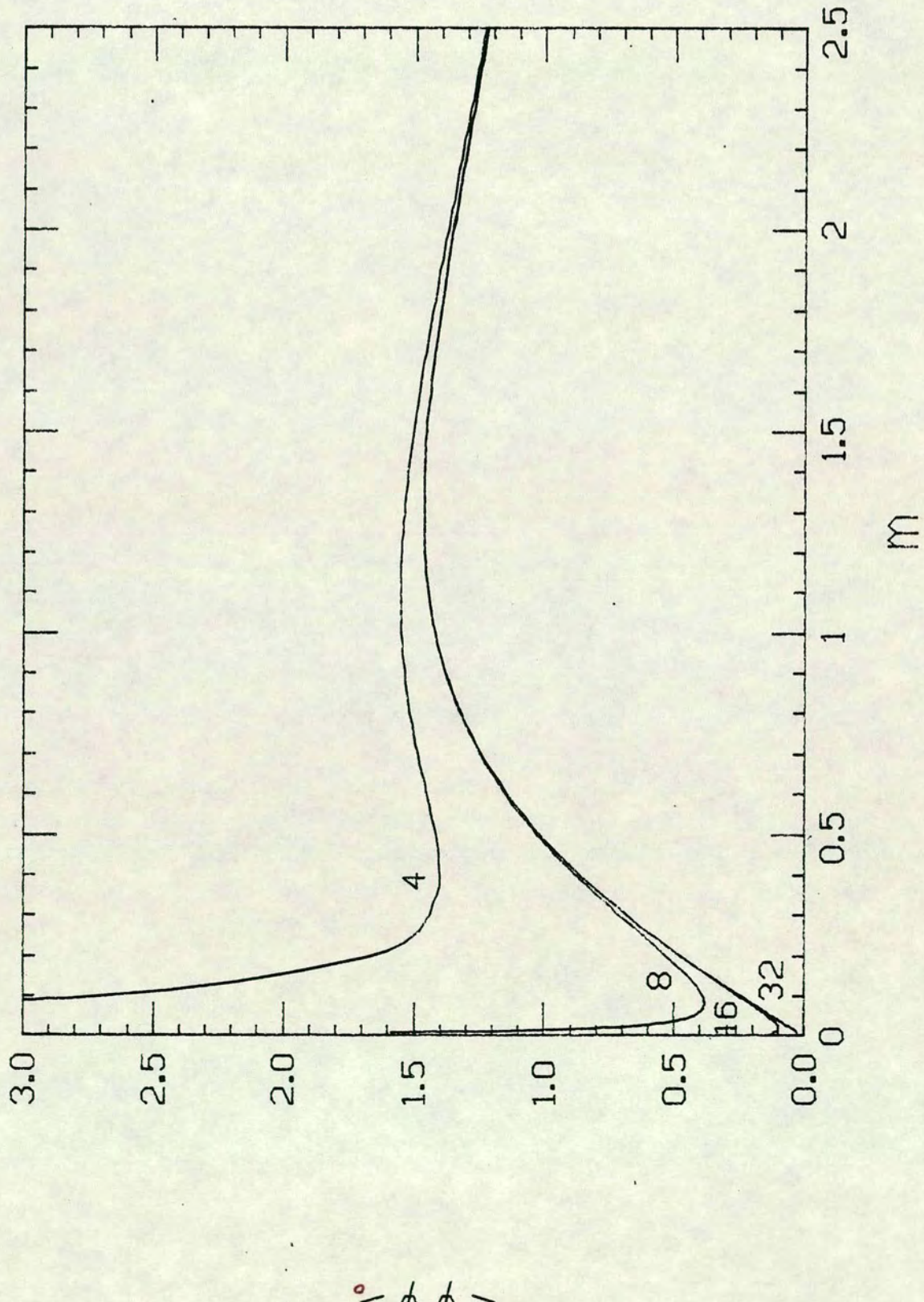
Varying N to see how $<\bar{\psi}\psi>_0$ changes with lattice size for periodic boundary conditions, results in Fig. 2.4. Again a lattice with N = 64 is close to the continuum, but with N = 4 there are large finite-size effects.

2.2. Propagators

In this section we shall calculate various "hadron" (fermion bilinear/trilinear) propagators for free lattice fermions and investigate finite-size effects. Periodic and antiperiodic boundary conditions in the spatial directions appear to yield upper and lower bounds respectively for both the meson-like and baryon-like propagators.

We shall consider Wilson fermions in four dimensions on a Euclidean L^3xL_4 lattice (L_4 is the time direction). By setting r = 0 we can obtain (four copies of) Susskind fermions, via a Kawamoto-Smit transformation (Kawamoto and Smit, 1981). We write the free fermion propagator (Green's function) as follows:

Fig. 2.4 $\langle\bar{\psi}\psi\rangle_0$ for Susskind fermions on N^4 lattices with periodic boundary conditions.



$$G(n;0) \equiv G(n) = \frac{1}{L^3 L_4} \sum_{\vec{q}} \sum_{q_4} e^{iq.n} \, G(q) \tag{2.11}$$

with

$$G(q) = \frac{-i\gamma_4 \sin q_4 - iQ + r(1-\cos q_4) + M}{\sin^2 q_4 + Q^2 + [r(1-\cos q_4) + M]^2} , \tag{2.12}$$

where

$$Q \equiv \sum_{i=1}^{3} \gamma_i \sin q_i$$

and

$$M \equiv \sum_{i=1}^{3} r(1-\cos q_i) + m .$$

The momentum sum is over $q_\mu = 2\pi(n_\mu + \delta_\mu)/L_\mu$, $n_\mu = 0,1, \ldots, L_\mu - 1$, where $\delta_\mu = 0$ for periodic boundary conditions and $\delta_\mu = 1/2$ for antiperiodic boundary conditions in the μ-direction. Note that we have used a different notation for G(q) in (2.12) since it differs, by irrelevant terms, from $\tilde{G}(q)$ in (2.5).

2.2.1. Calculation of fermion propagator

G(n) can be evaluated on a computer from (2.12) as it stands. However, some insight can be gained and computer time saved by performing the q_4 sum analytically. We shall consider the case $L_4 \to \infty$ for which

$$G_{L_4}(t,\vec{q}) \equiv \frac{1}{L_4} \sum_{q_4} e^{iq_4 t} \, G(q)$$

$$\longrightarrow \int_{-\pi}^{\pi} \frac{dq_4}{2\pi} e^{iq_4 t} \, G(q) \equiv G_\infty(t,\vec{q}). \tag{2.13}$$

(The case of finite L_4 is treated in Carpenter and Baillie, 1985.) This integral can be evaluated as shown below; there are two cases.

1) $\underline{0 \leq r < 1}$

The denominator in (2.12) can be written

$$(r^2-1)\cos^2 q_4 - 2r(r+M)\cos q_4 + 1 + Q^2 + (r+M)^2$$

which shows that there are poles at

$$\cos q_4 = \frac{\pm u - r(r+M)}{1-r^2}, \tag{2.14}$$

where

$$u^2 \equiv (1+rM)^2 + (1-r^2)(Q^2+M^2).$$

The right-hand side of (2.14) is ≥ 1 with the positive sign and ≤ -1 with the negative sign so the poles occur at $q_4 = iE_1$ and $q_4 = \pm\pi + iE_2$, where

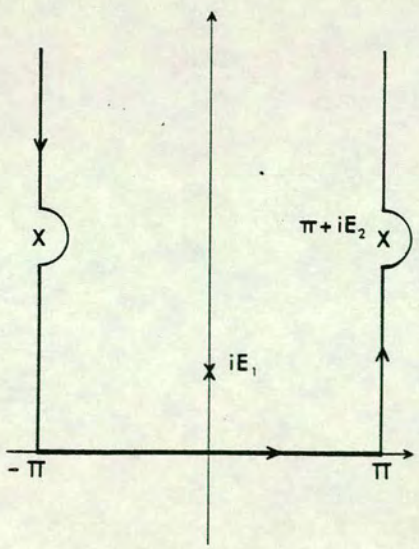
$$\cosh E_1 = \frac{u - r(r+M)}{1-r^2} \tag{2.15a}$$

and

$$\cosh E_2 = \frac{u + r(r+M)}{1-r^2}. \tag{2.15b}$$

For $t > 0$ we employ the contour in Fig. 2.5.

Fig. 2.5 Contour in the complex q_4 plane for the integral of eq. (2.12).



The contributions from the two legs of the contour parallel to the imaginary axis cancel due to the periodicity of the integrand (t takes integer values) and the piece at infinity parallel to the real axis is killed by the $e^{iq_4 t}$. Thus $G_\infty(t,\underline{q})$ is given simply by the residues from the two poles inside the contour:

$$G_\infty(t,\underline{q}) = \frac{1}{2U\sinh E_1}\left\{\gamma_4 \sinh E_1 - iQ + r(1-\cosh E_1) + M\right\} e^{-E_1 t}$$

$$+ \frac{(-1)^t}{2U\sinh E_2}\left\{-\gamma_4 \sinh E_2 - iQ + r(1+\cosh E_2) + M\right\} e^{-E_2 t} \quad (2.16)$$

For $t < 0$ the poles and the contour lie in the negative half-plane which results in a change in sign of the γ_4 terms in (2.16). Finally, for $t = 0$ we can do the integral analytically and find that the γ_4 terms in (2.16) disappear. Hence, for all t,

$$G_\infty(t,\underline{q}) = \frac{1}{2U\sinh E_1}\left\{\gamma_4\, \mathrm{sgn}(t) \sinh E_1 - iQ + r(1-\cosh E_1) + M\right\} e^{-E_1|t|}$$

$$+ \frac{(-1)^t}{2U\sinh E_2}\left\{-\gamma_4\, \mathrm{sgn}(t) \sinh E_2 - iQ + r(1+\cosh E_2) + M\right\} e^{-E_2|t|} \quad (2.17)$$

with the convention that sgn(0) = 0.

2) <u>r = 1</u>

The denominator in (2.12) reduces to

$$-2(1+M)\cos q_4 + 1 + Q^2 + (1+M)^2$$

which means that there is a pole at $q_4 = iE_1$ with

$$\cosh E_1 = 1 + \frac{Q^2 + M^2}{2(1+M)} \tag{2.18}$$

and now $U = (1 + M)^2$. For $t > 0$ we use the same contour as before (Fig. 2.5), but without the poles at $\pm\pi + iE_2$, to obtain

$$G_\infty(t,q) = \frac{1}{2U\sinh E_1}\left\{\gamma_4 \sinh E_1 - iQ + 1 - \cosh E_1 + M\right\} e^{-E_1 t} \tag{2.19}$$

Again $t < 0$ gives a sign change for the γ_4 term in (2.19). But now, for $t = 0$, when we analytically evaluate the integral, we find that the γ_4 term in (2.19) has been replaced by $1/2(1 + M)$. Hence, for all t,

$$G_\infty(t,q) = \frac{1}{2U\sinh E_1}\left\{\gamma_4\, \mathrm{sgn}(t) \sinh E_1 - iQ + 1 - \cosh E_1 + M\right\} e^{-E_1|t|} + \delta(t;0)\frac{1}{2(1+M)} \tag{2.20}$$

2.2.2. Meson-like propagators

We will now consider the "meson" propagators in the free theory. If we decompose

$$G_\infty(t,q) = \sum_{\mu=1}^{4} \gamma_\mu G_\mu(t,q) + \mathbb{I}\, G_u(t,q) \tag{2.21}$$

then the time-slice propagator for a typical fermion bilinear $\bar{\psi}\Gamma\psi$ is given by

$$\sum_{\underline{n}} \langle \bar{\psi} \Gamma \psi(t, \underline{n}) \; \bar{\psi} \Gamma \psi(0) \rangle$$

$$= \sum_{\underline{n}} \mathrm{Tr}\left[\Gamma G(t, \underline{n}) \Gamma \gamma_5 G^{+}(t, \underline{n}) \gamma_5 \right] \qquad (2.22)$$

$$= \frac{1}{L^3} \sum_{\underline{q}} \left\{ \sum_{\mu=1}^{4} T_\mu \left| G_\mu(t, \underline{q}) \right|^2 + T_u \left| G_u(t, \underline{q}) \right|^2 \right\},$$

where

$$T_\mu \equiv \mathrm{Tr}\left[\Gamma \gamma_\mu \Gamma \gamma_5 \gamma_\mu \gamma_5 \right]$$

and

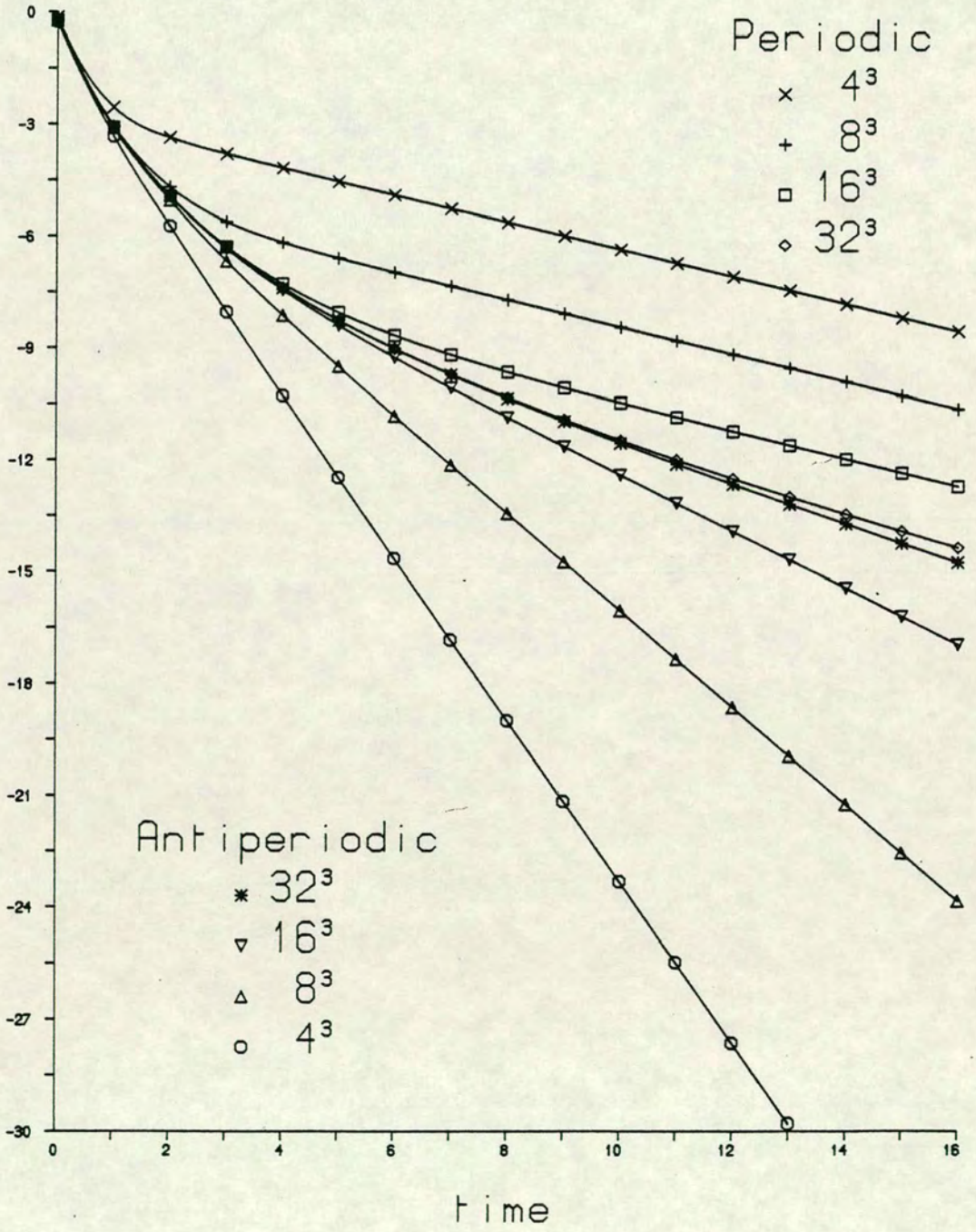
$$T_u \equiv \mathrm{Tr}\left[\Gamma \Gamma \right].$$

For example, in the case of the pion-type propagator ($\Gamma = \gamma_5$), $T_1 = T_2 = T_3 = T_4 = T_u = 4$. For purposes of numerical evaluation we note that the number of terms in the momentum sum (2.22) can be greatly reduced by exploiting the reflection and permutation symmetries of the summand. In this way we can reduce the range of the momentum sum to $0 \leq q_3 \leq q_2 \leq q_1 \leq \pi$, with a little care about the counting of terms on the edge of this domain.

The large-time behaviour of the propagator (2.22) is governed by the lowest-lying intermediate quark-antiquark state. For periodic boundary conditions this is the $\underline{q} = \underline{0}$ state with energy $2E_1$, irrespective of the lattice size. However, for antiperiodic boundary conditions the lowest-lying quark momentum state is $\underline{q} = (\pi/L, \pi/L, \pi/L)$ and the corresponding energy is, to a first approximation (the exact result is given by (2.15a) or (2.18)), $2E_1 \sim 2\sqrt{(m^2 + 3\pi^2/L^2)}$. The L-dependent correction is quite large even for fairly large lattices.

In Fig. 2.6 we plot the "pion" propagator ($\Gamma = \gamma_5$) for various lattice sizes, with periodic and antiperiodic boundary conditions, and with $r = 1$, $m = 0.2$ and $L_4 = \infty$. We observe that the finite-size effects can be at least as large for antiperiodic boundary conditions as for periodic boundary conditions, contrary to some expectations (Barbour *et al*, 1983). Antiperiodic boundary conditions usually win out at small t, but at large t they give larger finite-size effects than periodic boundary conditions. These two types of boundary conditions appear to bound the propagator from below and above respectively.

Fig. 2.6 Natural log. of the "pion" time-slice propagator, for r = 1, m = 0.2, $L_4 = \infty$, various L^3 and periodic/antiperiodic boundary conditions.



The other "meson" propagators behave in a very similar way to the "pion" propagator. In fact the "rho" propagator is essentially degenerate with the "pion" propagator at large t, because large-t behaviour for mesons is dictated by the G_4, G_u parts of the quark propagator ($G_i = 0$ for the $\underline{q} = \underline{0}$ intermediate quark states), and the T_4, T_u traces are identical for the pseudoscalar and vector meson bilinears.

2.2.3. Baryon-like propagators

The "baryon" propagators also display similar behaviour to the "pion" propagator. As an example we look at the "proton" propagator in the free theory. If a proton field is defined in terms of the quark field as

$$p_\alpha = (\psi^T C^{-1} \gamma_5 \psi) \psi_\alpha ,$$

where C is the Dirac charge conjugation matrix, then the free-field expression for the time-slice "proton" propagator is

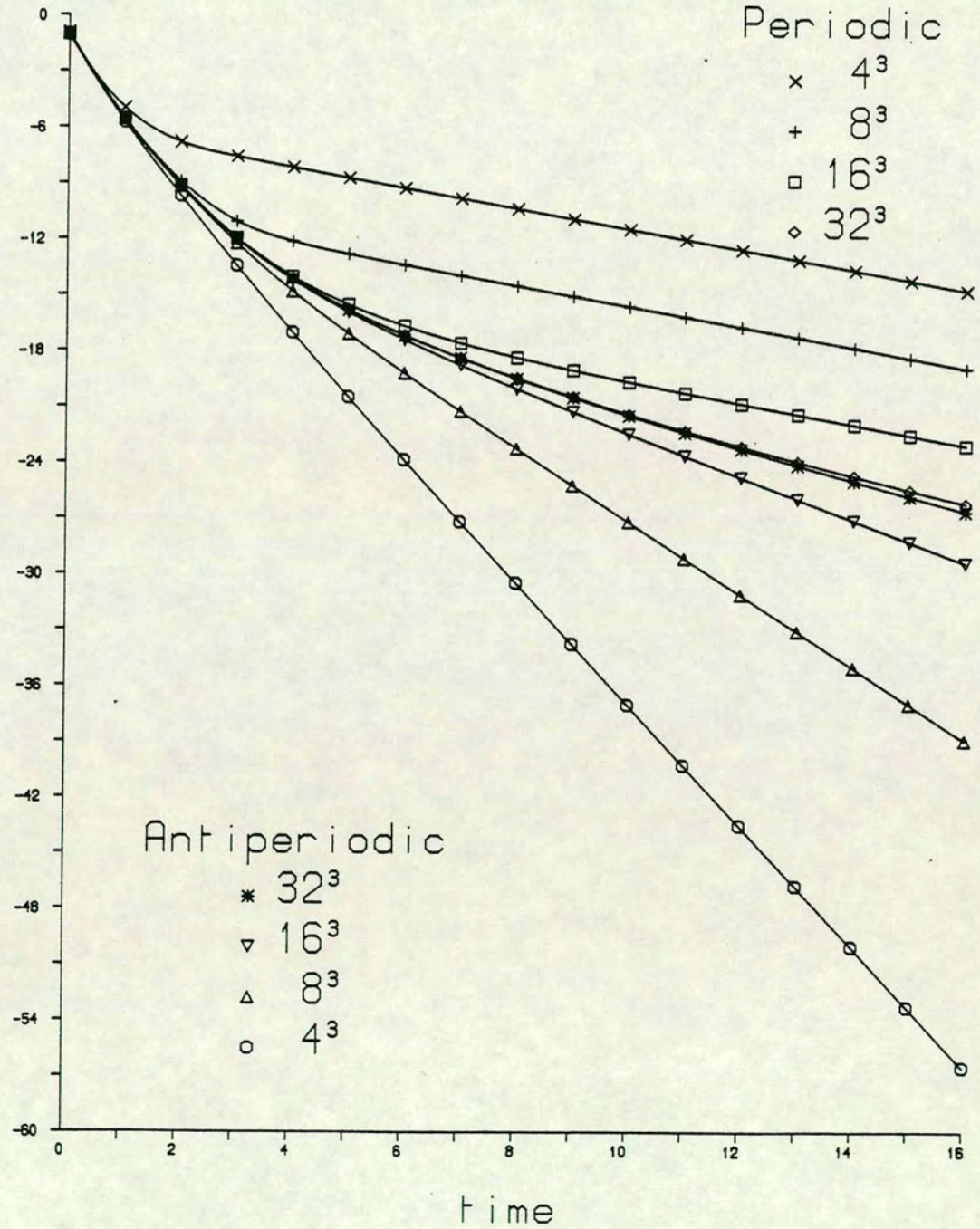
$$\sum_{\underline{n}} \langle p \bar{p} \rangle = \sum_{\underline{n}} \{ G_u (5 G_u^2 + 7 \sum_{\mu=1}^{4} G_\mu^2) \mathbb{1} + G_4 (7 G_u^2 + 5 \sum_{\mu=1}^{4} G_\mu^2) \gamma_4 \} , \qquad (2.23)$$

where

$$G_M(t, \underline{n}) = \frac{1}{L^3} \sum_{\underline{q}} e^{i \underline{q} \cdot \underline{n}} G_M(t, \underline{q}) \; ; \quad M = 1, 2, 3, 4, u .$$

Again computer time can be saved by exploiting the symmetries of the propagators to reduce the range of summation in (2.23) to $0 \leqslant n_3 \leqslant n_2 \leqslant n_1 \leqslant L/2$. The resulting "proton" propagator (actually the value of the upper component of the diagonal matrix (2.23)) is plotted in Fig. 2.7, for the same parameters as in Fig. 2.6. As before, we see lower and upper bounds on the propagator from antiperiodic and periodic boundary conditions respectively.

Fig. 2.7 Natural log. of the "proton" time-slice propagator, parameters as for Fig. 2.6.



2.2.4. Concluding remarks

The free-fermion case corresponds to infinite inverse coupling constant, β, so we expect this analysis to apply to QCD calculations at large β. This conclusion is supported by Monte Carlo work on small lattices at $\beta = 6.0$ (Gupta and Patel, 1983; Bernard, Draper, Olynyk and Rushton, 1983; Bowler *et al*, 1984) which yields degenerate pion and rho mass as we found in Sec. 2.2.

Chapter 3
Pseudo-Fermions

In this chapter we shall use the method of pseudofermions in a numerical simulation of the Schwinger model with Wilson fermions. In Sec. 1 we describe the method of pseudofermions and in Sec. 2 go on to discuss the details of using it in performing numerical simulations on a highly parallel computer. In Sec. 3 we firstly review the continuum Schwinger model and then turn to the numerical simulation, describing the pure gauge theory, free fermions, the quenched model and the dynamical model. We also outline an effective Lagrangian calculation of the meson propagators in U(N) and SU(N) lattice gauge theories at strong coupling.

3.1. The method

We begin by rewriting the effective action (Chap. 1.3.3) in terms of the Hermitian operator $K = (\not{\partial} + m)^{\dagger}(\not{\partial} + m)$:

$$S_{eff}(U) = S_G(U) - \frac{1}{2} tr\, ln\left[K(U)\right]. \quad (3.1)$$

For n_f flavours of Wilson fermions the effective action becomes

$$S_{eff}(U) = S_G(U) - \frac{n_f}{2} tr\, ln\left[K(U)\right]. \quad (3.2)$$

Setting $n_f = 0$ (that is, ignoring the fermionic determinant) yields the quenched approximation, whereas $n_f = 1$ gives the fully interacting unquenched or dynamical theory with one flavour of fermion.

The problem with using this action directly for Monte Carlo calculations is the non-local nature of the fermionic determinant. Consider updating the gauge field

variable on one link $U_\mu(l)$, say, leading to the change in effective action

$$e^{S_{eff}(U') - S_{eff}(U)} = e^{S_G(U') - S_G(U)} \frac{\det [\phi(U) + m]}{\det [\phi(U') + m]}. \quad (3.3)$$

If the new link variable is chosen close to the old one, which is usually the case when the Metropolis algorithm (discussed in Chap. 1.4.1) is used, then the change in the effective action can be linearised

$$S_{eff}(U') - S_{eff}(U) = S_G(U') - S_G(U)$$
$$- \frac{1}{2} \sum_{n,m} K^{-1}(n,m) \frac{\delta K(m,n)}{\delta U_\mu(l)} \delta U_\mu(l) + O\left(\delta U_\mu^2(l)\right). \quad (3.4)$$

As only one gauge link is being changed, $\delta K/\delta U_\mu(l)$ is non-zero only for sites neighbouring this link. Hence we only require the elements of K^{-1} for these sites, rather than the entire Green function. These elements can be calculated in a variety of ways, one of which is the pseudofermion method of Fucito, Marinari, Parisi and Rebbi, 1981.

This uses the fact that the inverse of a Hermitian operator K can be written

$$K^{-1}(m,n) = \frac{\int \mathcal{D}\phi^* \mathcal{D}\phi \; \phi^*(n) \phi(m) e^{-S_{pf}}}{\int \mathcal{D}\phi^* \mathcal{D}\phi \; e^{-S_{pf}}} \quad (3.5)$$

where $\equiv \langle \phi^*(n) \phi(m) \rangle$,

$$S_{pf} = \sum_{n,m} \phi^*(n) K(n,m) \phi(m). \quad (3.6)$$

S_{pf} is the action for the so-called pseudofermions ϕ which are complex bosonic fields. Now we can approximate $K^{-1}(m,n)$ for a given gauge configuration $\{U\}$ by performing another Monte Carlo calculation, this time using the heat bath algorithm (discussed in Chap. 1.4.2) to improve convergence. If an ensemble of N_{pf} pseudofermion configurations $\{\phi\}$ is generated then

$$K^{-1}(m,n)\Big|_{\{U\}} = \frac{1}{N_{pf}} \sum_{\{\phi\}} \phi^*(n)\,\phi(m)\Big|_{\{U\}}. \qquad (3.7)$$

This yields the exact result (apart from errors of order δU^2) in the limit $N_{pf} \to \infty$ (in practice one takes N_{pf} as large as is required to produce reliable results) which is then fed back into the effective action for the gauge fields as these are updated, according to (3.4). Thus simulation of the dynamical theory using the pseudofermion method requires two Monte Carlo simulations: the usual Metropolis one for the gauge fields and, within this, a heat bath calculation for the pseudofermions.

The above discussion has been for 1-component, i.e. Susskind, fermions which we shall use in the simulation of SU(2) (Chap. 4.3). For the simulation of the Schwinger model (Sec. 3) we shall require Wilson fermions in two space-time dimensions, i.e. 2-component fermions. In this case the pseudofermion variables will also have two components and K will be a 2x2 matrix. Hence (3.5) and (3.6) respectively become

$$K^{-1}(m,n) = \langle \phi^*(n)\, \phi^T(m) \rangle \qquad (3.8)$$

and

$$S_{pf} = \sum_{n,m} \phi^+(n)\, K(n,m)\, \phi(m). \qquad (3.9)$$

3.2. Computational details

The numerical simulation of the Schwinger model was carried out entirely on the ICL Distributed Array Processor (DAP). This computer (described more fully in Appendix I) has a highly parallel Single Instruction stream, Multiple Data stream (SIMD) architecture consisting of a 64x64 array of bit-serial processing elements (PEs), each with connections to the four nearest neighbours. (We note that the

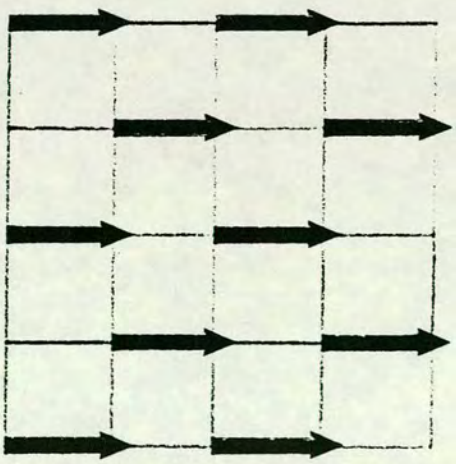
simulation could also have been performed on the GRID (Appendix II), which has a similar architecture to the DAP, in exactly the same manner. In fact, software which will automatically translate a DAP program into an equivalent one for the GRID is described in Chap. 5.) Therefore the lattice size chosen for the simulation was 64x64 so that one site of the lattice occupies one PE of the DAP. Hence at each PE a pseudofermion variable and two gauge field variables (one for each link in the positive coordinate directions) are stored. The SIMD architecture of the DAP means that all the PEs can be updated simultaneously. However, this would violate detailed balance - variables which interact must not be modified simultaneously - therefore some of the PEs must be "masked off". Different masks are required for the gauge field and the pseudofermion variables.

The action for the gauge fields is

$$S_G(U) = \beta \sum_{\square} \left(1 - \mathrm{Re}\ \mathrm{tr}\ U_{\square} \right), \qquad (3.10)$$

thus the gauge fields interact via plaquettes. This means that we can only update half of the links in each direction simultaneously, as shown in Fig. 3.1.

Fig. 3.1 Mask for updating gauge fields: links shown as arrows may be updated simultaneously.



This holds in any dimension.

The pseudofermion actions for Susskind and Wilson fermions are as follows.

For Susskind fermions (Chap. 1.3.2)

$$\not{\partial}(n,m) = \frac{1}{2} \sum_{\mu} \eta_\mu \left\{ U_\mu(n)\, \delta(m, n+e_\mu) - U_\mu^\dagger(n-e_\mu)\, \delta(m, n-e_\mu) \right\}, \tag{3.11}$$

which is anti-Hermitian:

$$\not{\partial}^\dagger(n,m) \equiv \not{\partial}^*(m,n) = -\not{\partial}(n,m).$$

Hence

$$S_{pf} = \sum_{n,m,\ell} \phi^*(n) \left[\not{\partial}+m\right]^\dagger(n,\ell) \left[\not{\partial}+m\right](\ell,m)\, \phi(m)$$

$$= \left(\frac{d}{2} + m^2\right) \sum_n \phi^*(n)\, \phi(n) \qquad \text{self interaction}$$

$$- \frac{1}{4} \sum_{n,\mu} \phi^*(n) \left[U_\mu(n)\, U_\mu(n+e_\mu)\, \phi(n+2e_\mu) + U_\mu^\dagger(n-e_\mu)\, U_\mu^\dagger(n-2e_\mu)\, \phi(n-2e_\mu) \right]$$

'straight' next nearest neighbour interaction

$$+ \frac{1}{4} \sum_{\substack{n,\mu,\nu \\ \mu \neq \nu}} \eta_\mu \eta_\nu\, \phi^*(n) \Big[U_\mu(n) U_\nu(n+e_\mu)\, \phi(n+e_\mu+e_\nu) - U_\mu(n) U_\nu^\dagger(n+e_\mu-e_\nu)\, \phi(n-e_\mu+e_\nu)$$

$$- U_\mu^\dagger(n-e_\mu) U_\nu(n-e_\mu)\, \phi(n-e_\mu+e_\nu) + U_\mu^\dagger(n-e_\mu) U_\nu^\dagger(n-e_\mu-e_\nu)\, \phi(n-e_\mu-e_\nu) \Big].$$

'bent' next nearest neighbour interaction. (3.12)

In the last term, $\eta_\mu \eta_\nu = (-1)^{n_1}$ for $\mu \neq \nu$, where $n = (n_1, n_2)$. We notice that there are no nearest neighbour interactions, that is, there is no term in the action involving $\phi(n)$ and $\phi(n \pm e_\mu)$.

For Wilson fermions (Chap. 1.3.1)

$$\not{D}(n,m) = \frac{1}{2}\sum_{\mu}\left\{(\gamma_\mu - r)\,U_\mu(n)\,\delta(m,n+e_\mu) - (\gamma_\mu + r)\,U_\mu^{+}(n-e_\mu)\,\delta(m,n-e_\mu)\right\}. \tag{3.13}$$

If we write $\not{D} = \not{D}_1 + r\not{D}_2$ then $\not{D}_1$ is anti-Hermitian and $\not{D}_2$ is Hermitian. Hence

$$S_{pf} = \left[\frac{d}{2}(1+r^2) + m^2\right]\sum_n \phi^*(n)\,\phi(n)$$

self interaction

$$-\frac{1}{4}(1-r^2)\sum_{n,\mu}\phi^*(n)\left[U_\mu(n)\,U_\mu(n+e_\mu)\,\phi(n+2e_\mu) + U_\mu^{+}(n-e_\mu)\,U_\mu^{+}(n-2e_\mu)\,\phi(n-2e_\mu)\right]$$

'straight' next nearest neighbour interaction

$$+\frac{1}{4}\sum_{\substack{n,\mu,\nu\\ \mu\neq\nu}}\phi^*(n)\Big[-(\gamma_\mu+r)(\gamma_\nu-r)\,U_\mu(n)\,U_\nu(n+e_\mu)\,\phi(n+e_\mu+e_\nu)$$
$$+(\gamma_\mu+r)(\gamma_\nu+r)\,U_\mu(n)\,U_\nu^{+}(n+e_\mu-e_\nu)\,\phi(n+e_\mu-e_\nu)$$
$$+(\gamma_\mu-r)(\gamma_\nu-r)\,U_\mu^{+}(n-e_\mu)\,U_\nu(n-e_\mu)\,\phi(n-e_\mu+e_\nu)$$
$$-(\gamma_\mu-r)(\gamma_\nu+r)\,U_\mu^{+}(n-e_\mu)\,U_\nu^{+}(n-e_\mu-e_\nu)\,\phi(n-e_\mu-e_\nu)\Big]$$

'bent' next nearest neighbour interaction

$$-mr\sum_{n,\mu}\phi^*(n)\left[U_\mu(n)\,\phi(n+e_\mu) + U_\mu^{+}(n-e_\mu)\,\phi(n-e_\mu)\right]. \tag{3.14}$$

nearest neighbour interaction

For computational convenience, a representation of the γ matrices is chosen in which γ_2 is diagonal:

$$\gamma_1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad ; \quad \gamma_2 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

We notice firstly that for r = 0 the Susskind case is recovered and secondly that for r = 1 the straight next nearest neighbour interactions cancel (that is, there is no $\phi(n)$ and $\phi(n\pm 2e_\mu)$ term).

Thus there are three possible cases depending on the Wilson r parameter. Moreover, the masks also depend on the number of dimensions.

For two dimensions we have:

r = 0 (Wilson fermions reduce to two copies of Susskind fermions)

The nearest neighbour interactions vanish so we can update 1 in 4 sites, see Fig. 3.2a.

r = 1 (Wilson fermions)

The straight next nearest neighbour interactions vanish so we can update 1 in 4 sites, see Fig. 3.2b.

0 < r < 1

All the interactions are present so the best we can do is to update 1 in 5 sites, see Fig. 3.2c, although because we are using a 64x64 lattice a 1 in 8 update pattern such as Fig. 3.2d is easier to implement.

For numerical simulations of QCD the following update ratios (with appropriate four-dimensional masks) are possible: for r = 0, only 1 in 16; for r = 1, 1 in 8; and for 0 < r < 1, 1 in 9 or more practically 1 in 16.

We shall be concerned with calculating the chiral condensate $\langle\bar{\psi}\psi\rangle$ which is the quark propagator at zero space-time spacing and is given by

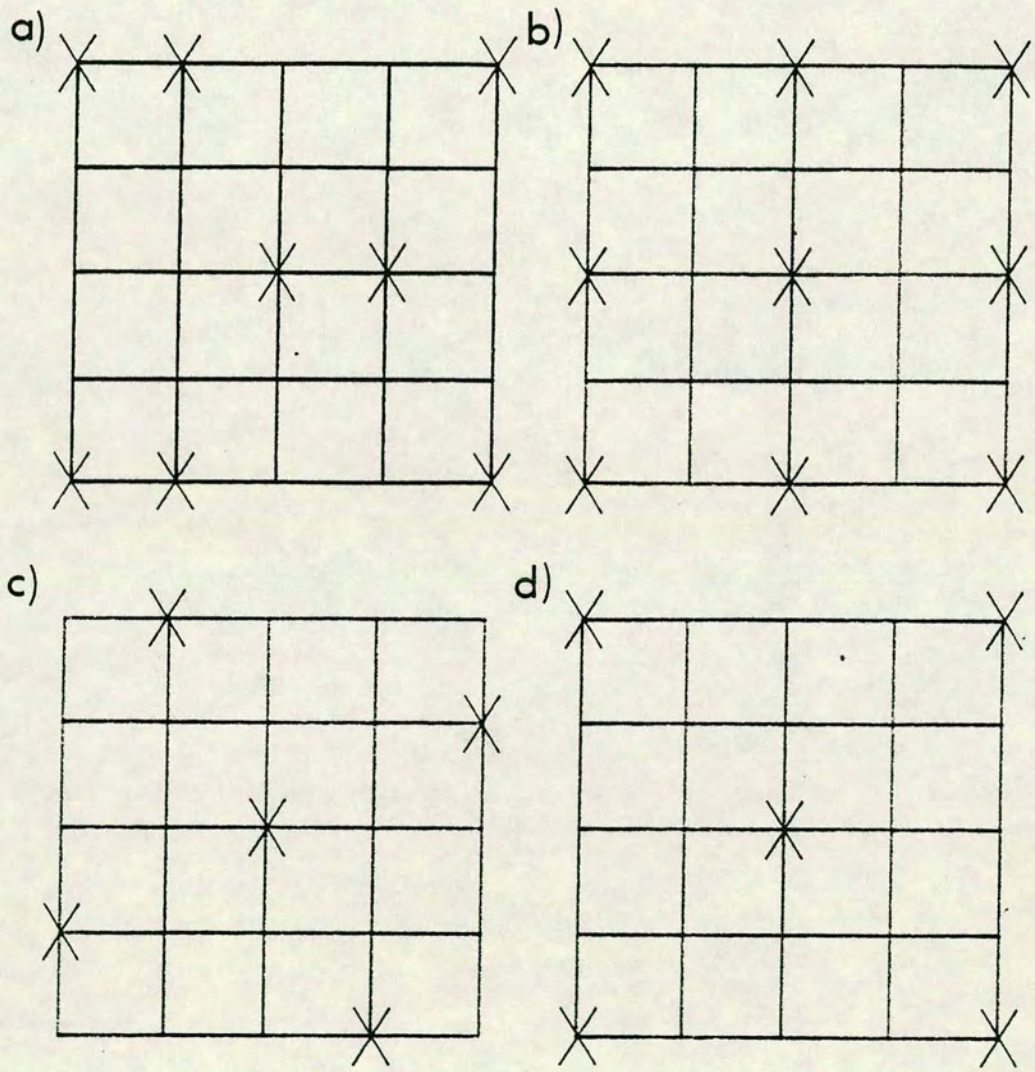
$$\langle \bar{\psi}\psi \rangle = tr\,(\not{\partial}+m)^{-1}. \qquad (3.15)$$

For the Wilson fermions in the Schwinger model this can be written

$$\langle \bar{\psi}\psi \rangle = Tr\,[(\not{\partial}+m)^{-1}](n,n), \qquad (3.16)$$

where Tr denotes trace over the two Dirac indices only (for single colour quarks). In terms of Wilson pseudofermions this is, from (3.8),

Fig. 3.2 Masks for updating pseudofermions: sites shown with crosses may be updated simultaneously. a) $r = 0$; b) $r = 1$; c) & d) $0 < r < 1$.



$$\langle \bar{\psi}\psi \rangle = Tr \langle [(\not{\partial}+m)\phi]^*_{(n)} \; \phi^T_{(n)} \rangle . \qquad (3.17)$$

For Susskind fermions, the $\not{\partial}$ in (3.17) disappears because it connects nearest neighbour sites which do not interact in (3.12), leaving

$$\langle \bar{\psi}\psi \rangle = m \langle \phi^*_{(n)} \phi_{(n)} \rangle . \qquad (3.18)$$

Finally we note that, computationally, Wilson fermions require twice as much work as Susskind in two space-time dimensions: a Wilson fermion has 2 components at each site of a 2Nx2N lattice; a Susskind fermion begins with 1 component which species-doubles to 4 (on a NxN lattice) and is spread out over the 2Nx2N lattice giving 1 component per site.

3.3. Schwinger model

Recently, the massive Schwinger model has been investigated numerically (Carson and Kenway, 1986) using Susskind fermions (Susskind, 1977). The order parameter for chiral symmetry breaking $<\bar{\psi}\psi>$ and the low-lying meson masses were calculated for both the model with two flavours and the model with one flavour for the quenched case and for the unquenched or dynamical case. Here we shall also investigate the massive Schwinger model numerically, but we will calculate only $<\bar{\psi}\psi>$ and will use Wilson fermions (Wilson, 1977). The motivation for doing this is as follows. We wish to determine the behaviour of $<\bar{\psi}\psi>$ in both the quenched and the dynamical massive Schwinger model. In particular, we would like to decide, on the basis of our results, which of two analytical calculations of the behaviour of $<\bar{\psi}\psi>/g$ as $m/g \to 0$ for the quenched case is correct. In addition, for the dynamical case there is an exact value of $<\psi\psi>$ for the massless model but how does $<\bar{\psi}\psi>$ vary with mass? We use Wilson fermions to avoid the problem of fermion-doubling which for Susskind fermions must be overcome by introducing a one-link mass term which gives the

unwanted flavours a mass of order of the cut-off to decouple them (Burkitt, Kenway and Kenway, 1983). Unfortunately, Wilson fermions change the mass scale of the theory. We can find the critical mass from the exact massless value of $<\overline{\psi}\psi>$ for the dynamical case but for the quenched case we must find it numerically.

3.3.1. Continuum Schwinger model

The Schwinger model (Schwinger, 1962) is quantum electrodynamics of a massless fermion with charge g in one space and one time dimension; it is exactly soluble. The massive theory (Coleman, Jackiw and Susskind, 1975; Coleman, 1976) is not exactly soluble but can be analysed by perturbation theory at both strong and weak coupling. This model is used primarily for testing ideas in quantum field theory and for checking algorithms in lattice gauge theory since it contains many of the interesting features found in more physical models. In particular it displays both the properties of asymptotic freedom and confinement of the fundamental charges found in QCD. Hence the methods developed to investigate the Schwinger model may also be of use for QCD.

The massive Schwinger model is described by the Lagrangian density (derived in Chap. 1.1)

$$\mathcal{L}_{QED} = -\frac{1}{4} F_{\mu\nu} F^{\mu\nu} + \overline{\Psi}(i\not{\partial} - g\not{A} - m)\Psi, \qquad (3.19)$$

where

$$\Psi(x) = \begin{pmatrix} \Psi_1(x) \\ \Psi_2(x) \end{pmatrix}, \quad \overline{\Psi} = \Psi^{+}\gamma_0$$

and

$$x = (x^0, x^1).$$

The equations of motion are

$$(i\not{\partial} - g\not{A})\,\psi = 0$$

$$j_\mu \equiv g\,\bar{\psi}\gamma_\mu\psi = \partial^\nu F_{\mu\nu} \,. \tag{3.20}$$

The coupling constant g has dimensions of mass; consequently the model is super-renormalisable, and both g and m are finite (though bare) parameters. The dimensionless parameter that measures the interaction strength is m/g. The limit $m/g \to 0$ is the exactly soluble massless Schwinger model and the limit $m/g \to \infty$ is the exactly soluble free theory. Since the model is exactly soluble in both limits it is possible to do perturbative calculations. We shall discuss these calculations after looking at the massless model.

Following Schwinger, alternative solutions of the massless model have been given by Lowenstein and Swieca, 1971; Casher, Kogut and Susskind, 1974; and Bander, 1976, amongst others. Schwinger, 1962, solves the model by computing the Green's function (in the Lorentz gauge)

$$G_F(p) = \frac{1}{p^2 + \frac{g^2}{\pi} - i\varepsilon} \,. \tag{3.21}$$

Lowenstein and Swieca, 1971, solve the model in terms of explicit operator solutions and obtain the covariant solution

$$A^\mu(x) = -\frac{\sqrt{\pi}}{g}\left(\varepsilon^{\mu\nu}\partial_\nu\tilde{\sigma}(x) + \varepsilon^{\mu\nu}\partial_\nu\tilde{\eta}(x)\right), \tag{3.22}$$

where $\tilde{\sigma}$ is a massive free scalar field and $\tilde{\eta}$ is a massless field quantised with indefinite metric. Casher, Kogut and Susskind, 1974, solve the model in terms of the degrees of freedom of the Lagrangian density (3.19) and show that

$$T^{\mu\nu}(x) \equiv -i \langle 0 | T\, j^{\mu}(x)\, j^{\nu}(0) | 0 \rangle$$

$$= \frac{g^2}{\pi} \left(g^{\mu\nu} \Box - \partial^{\mu}\partial^{\nu} \right) \Delta_F(m^2; x^2), \qquad (3.23)$$

where Δ_F is the Feynman propagator. Bander, 1976, solves the model by making the following identification with a boson theory:

$$\psi_{1,2}(x) = \left(\frac{\Lambda}{2\pi\gamma} \right)^{\frac{1}{2}} \exp\left\{ -i\sqrt{\pi} \int_{-\infty}^{x_1} dz\, e^{-z/R} \left[\Pi(x_0, z) \pm \partial_1 \phi(x_0, z) \right] \right\}, \qquad (3.24)$$

where R is a spatial cut-off (introduced to keep the integrals finite and set to infinity at the end of the calculation), Λ is a momentum cut-off (also allowed to go to infinity), γ is Euler's constant and $\phi(x)$ is a boson field with canonical momentum $\Pi(x)$. Then, in the Coulomb gauge $A_1 = 0$,

$$A_0 = -\frac{g}{\partial_1^2} \bar{\psi} \gamma_0 \psi, \qquad (3.25)$$

so that the effective Lagrangian density for fermions is

$$\mathcal{L}_{eff} = i \bar{\psi} \not{\partial} \psi - \frac{g^2}{2} \bar{\psi} \gamma_0 \psi \frac{1}{\partial_1^2} \bar{\psi} \gamma_0 \psi, \qquad (3.26)$$

which expressed in terms of the corresponding bosons yields the action

$$\Pi \partial_0 \phi - \frac{\Pi^2}{2} - \frac{(\partial_1 \phi)^2}{2} - \frac{\mu^2}{2} \phi^2, \qquad (3.27)$$

where $\mu^2 = g^2/\pi$. This correspondence between the fermion and boson theory demonstrates explicitly that the fundamental fermion of the theory, ψ, is absent from the physical space of states; all that is present is a free neutral pseudoscalar meson ϕ with mass $g/\sqrt{\pi}$ which can be thought of as a fermion-antifermion bound state. Physically this fermion confinement is caused by charge screening. If we attempt to separate a fermion-antifermion pair, when

the separation is sufficiently large it is energetically favourable for a new pair to materialise from the vacuum. The new fermion is attracted to the original antifermion and the new antifermion is attracted to the original fermion. This both screens the long range Coulomb force and ensures that what we are separating is not a fermion and an antifermion but two fermion-antifermion bound states. (The same mechanism is believed to be responsible for quark confinement in QCD.)

Finally we note that global chiral symmetry is broken and the vacuum is infinitely degenerate. Different vacua may be labelled by an angle $\theta \in [-\pi,\pi]$; global chiral transformations rotate one vacuum into another. Again no Goldstone boson appears, this time because the axial current is afflicted with an anomaly. The parameter θ may be identified with a constant background electric field (Coleman, 1976). This field could be introduced into four-dimensional QED but there the vacuum would suffer dielectric breakdown since it is energetically favourable for the vacuum to emit pairs until the background field is brought down to zero. In one spatial dimension, however, the energetics of pair production are different. It is not energetically favourable for the vacuum to produce a pair if the background field F is such that $|F| \leq e/2$; if $|F| > e/2$, pairs will be produced until $|F| \leq e/2$. Thus physics is a periodic function of F with period e, and θ may be identified as

$$\theta \equiv \frac{2\pi F}{e} . \tag{3.28}$$

We now resume our discussion of the massive Schwinger model. Giving the fermions a mass changes the Lagrangian of the boson field to

$$\mathcal{L} = \frac{(\partial\phi)^2}{2} - \frac{\mu^2}{2}\phi^2 - \frac{m\Lambda}{\pi\gamma}\cos\left(2\sqrt{\pi}\,\phi - \theta\right) \tag{3.29}$$

The massive model is still dependent upon the parameter θ, labelling different vacua. The mass term of course explicitly breaks the chiral invariance so that the vacua are no longer degenerate. However, contrary to naive expectations, all the vacua remain stable because of the absence of Goldstone bosons. We will

restrict ourselves to the case $\theta = 0$.

For $m << g$, the Lagrangian describes a heavy pseudoscalar meson with weak self interactions. Thus the model always contains at least one particle: the original pseudoscalar meson of mass

$$M^- = \frac{g}{\sqrt{\pi}} + me^{\gamma} + O(m^2), \tag{3.30}$$

where γ is Euler's constant. If any other particles are present, they will be weakly bound n-mesons of mass nM^- plus small corrections. In particular, the next particle is a scalar meson of mass

$$M^+ = 2M^- - \pi^2 e^{2\gamma} \frac{m^2}{M^-} + O(m^3). \tag{3.31}$$

As $m \to \infty$ the fermion decouples and the model reduces to a pure U(1) gauge theory (which may be solved by transfer matrix methods).

3.3.2. Pure gauge theory

In order to perform our simulations of the quenched and dynamical massive Schwinger model we shall require equilibrated U(1) gauge configurations at various values of the inverse coupling, β. These will be used as fixed background configurations for the quenched case and as starting configurations for the dynamical case. We use six β values: ∞ (free fermions), 8, 3, 2.5, 0.25 and 0 (strong coupling limit). The gauge configuration for $\beta = \infty$ corresponds to an ordered start i.e. all the gauge fields being set equal to 1; the gauge configuration for $\beta = 0$ corresponds to a disordered start i.e. all the gauge fields being set equal to e^{ir}, with r a pseudo-random number in $[0,2\pi]$. Gauge configurations for the other β values are generated by the standard quenched Monte Carlo Metropolis algorithm (Chap. 1.4.1) beginning from an ordered start and doing 75,000 sweeps, with an update angle $\delta U = 0.2 \times 2\pi$ (giving an acceptance rate of 73%), to attain equilibrium. (On the ICL DAP, one sweep of the 64x64 lattice takes approximately 0.04 seconds.) The resulting plaquette energies

$$E_\square = 1 - Re\, U_\square \qquad (3.32)$$

averaged over the last 1000 sweeps for each value of β are listed in Table 3.1.

Table 3.1

Average plaquette energies of gauge configurations at each β value.

β	$<E_\square>$
0.25	0.875 ± .039
2.5	0.235 ± .010
3.0	0.190 ± .009
8.0	0.065 ± .004

3.3.3. Free fermions

Before investigating the interacting theory, it is worthwhile looking at free fermions, for which we know $<\bar{\psi}\psi>$ analytically (Chap. 2.1), in order to see how the pseudofermion method performs.

We choose N_{pf} = 100 in the pseudofermion method so that $<\bar{\psi}\psi>$ is obtained by averaging over 100 pseudofermion configurations and pseudofermion sweeps are carried out in sets of 100. The update angle, δU, is chosen as $0.1 \times 2\pi$. We run two simulations - one from a disordered start (that is, all the pseudofermion variables set to random numbers in [0,1]) and one from an ordered start (all the pseudofermions set to 0) - at each of four masses 2.1, 2.05, 2.025 and 2.01 with periodic and antiperiodic boundary conditions. (On the ICL DAP, one set of 100 pseudofermion sweeps through the 64x64 lattice takes approximately 1.2 minutes.)

With antiperiodic boundary conditions $<\bar{\psi}\psi>$ converges to the analytical answer, at 0.3% level of accuracy, within the first set of 100 pseudofermion sweeps at all four masses.

With periodic boundary conditions $<\bar{\psi}\psi>$ converges over the first few sets at

the highest mass, m = 2.1, as shown in Fig. 3.3 but at the lowest mass, m = 2.01, metastable states are encountered and even after 1000 sets $<\bar{\psi}\psi>$ has not fully converged, Fig. 3.4. If we estimate the error in $<\bar{\psi}\psi>$ by binning the data in time and take the bin size which yields the maximum error as an indication of the correlation time in the measurement of $<\bar{\psi}\psi>$, we obtain correlations of 5, 10, 20 and 90 sets of 100 sweeps respectively for the four masses in descending order. Averaging the last 100 sets out of the 1000 for m = 2.01 and the last 180 out of the 200 for the other masses for each start with periodic boundary conditions yields the values in Table 3.2.

Table 3.2

Average $\bar{\psi}\psi$ for free fermions with periodic boundary conditions obtained from ordered and disordered starts compared with the value obtained analytically, at different masses.

m	$<\bar{\psi}\psi>_o$	$<\bar{\psi}\psi>$ ordered	$<\bar{\psi}\psi>$ disordered
2.1	.817	.817 ± .005	.817 ± .005
2.05	.808	.809 ± .008	.804 ± .005
2.025	.805	.805 ± .016	.803 ± .008
2.01	.825	.823 ± .033	.860 ± .041

Hence the pseudofermion method performs well for free fermions, except near the zero mode at $m_c = 2$ caused by periodic boundary conditions (see Chap. 2.1).

3.3.4. The quenched model

First we review the two analytical calculations which have been performed for the quenched massive Schwinger model. Carson and Kenway, 1986, use the replica trick, which consists of generalising the model to one containing N identical fermion species and taking the limit $N \to 0$ at the end of the calculation, in the strong coupling regime. This removes the fermionic determinant that arises from the fermion integration in the partition function and works regardless of whether the fermion has a mass. The result is that

Fig. 3.3 $\langle\bar{\psi}\psi\rangle$ for free fermions at highest mass from ordered and disordered starts.

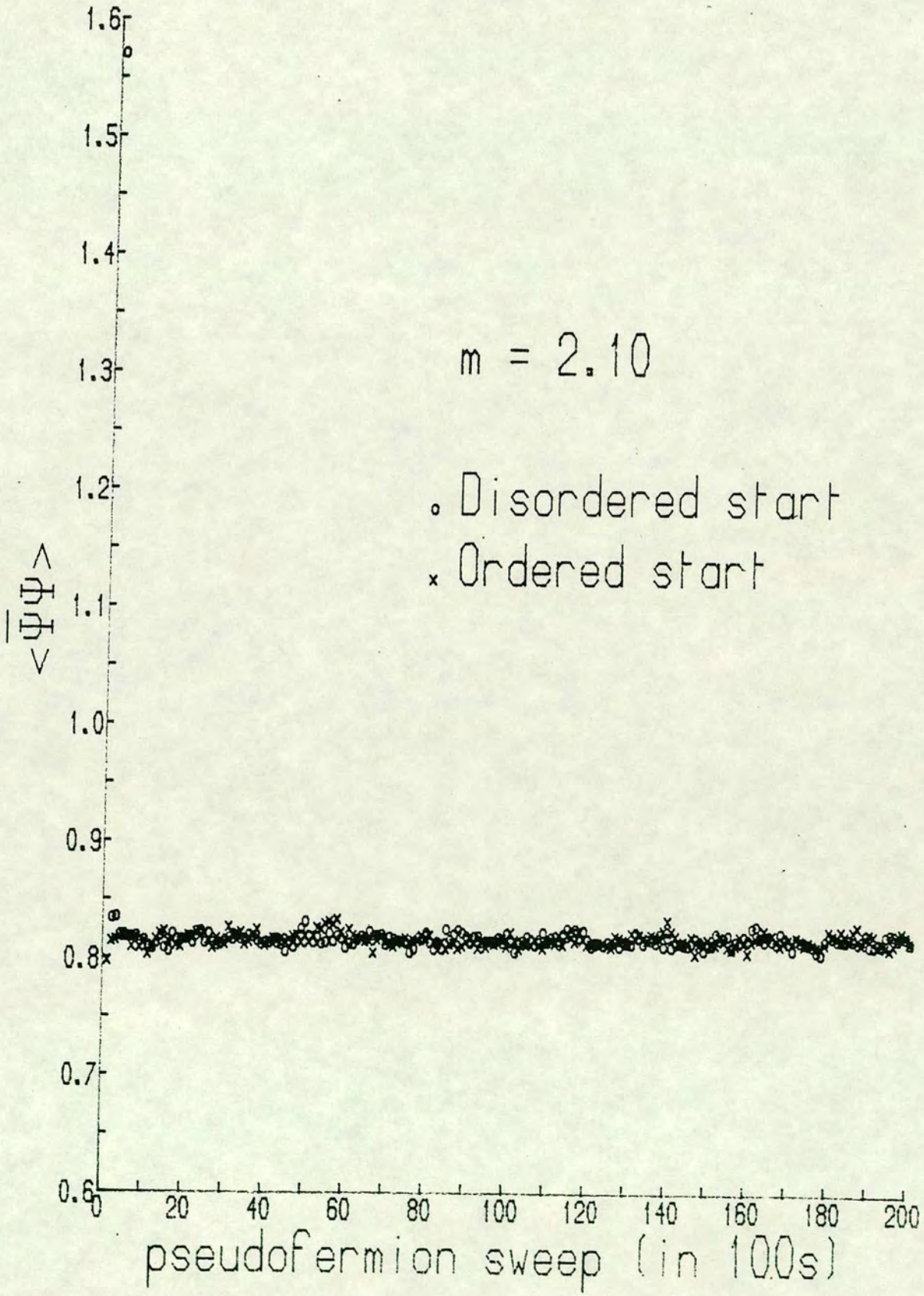
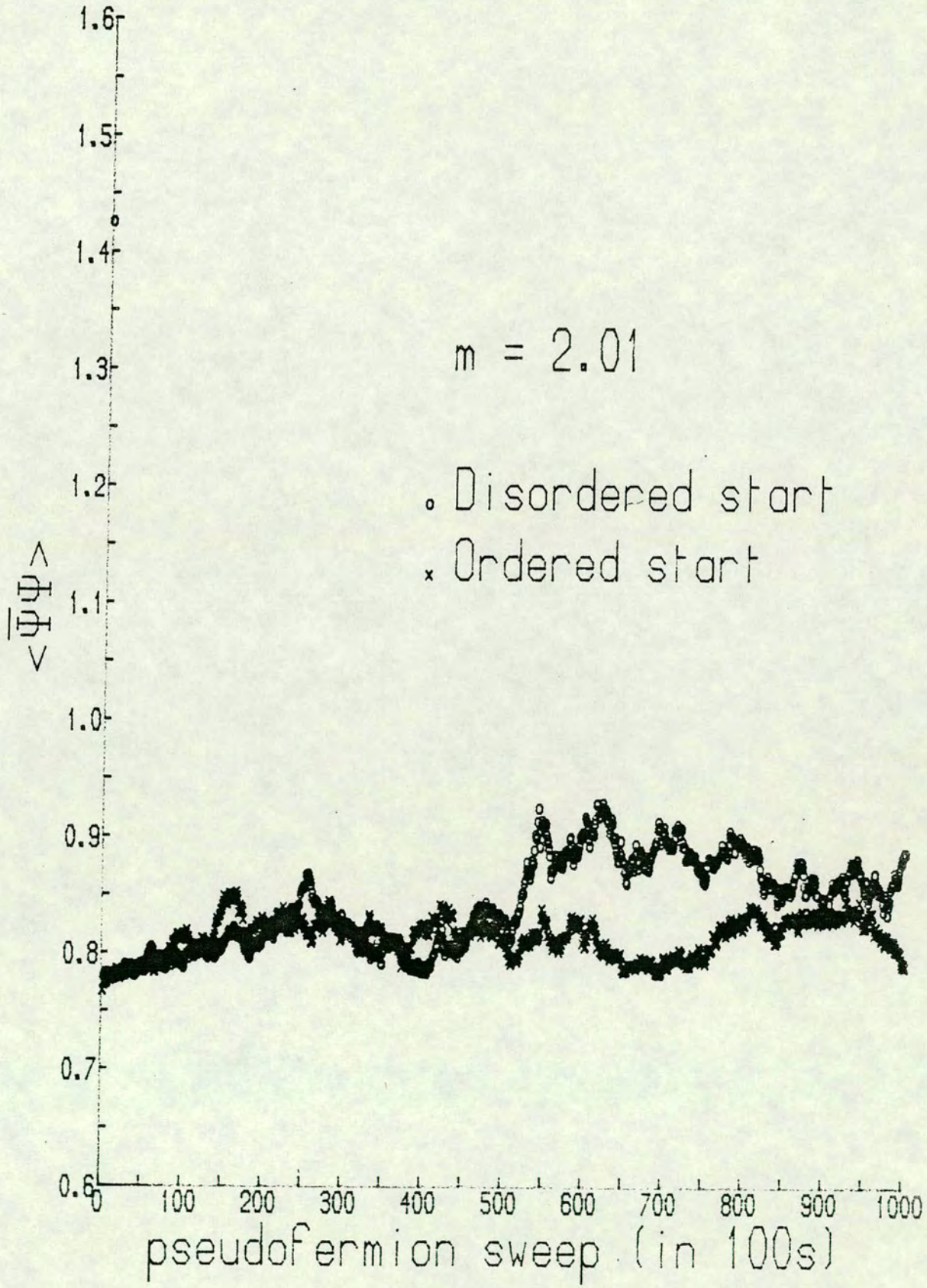


Fig. 3.4 $\langle\bar{\psi}\psi\rangle$ for free fermions at lowest mass from ordered and disordered starts.



$$\langle \bar{\Psi}\Psi \rangle - \langle \bar{\Psi}\Psi \rangle_0 = c\lambda e^{g^2/2\pi\lambda^2} - c\lambda_0 , \qquad (3.33)$$

where

$$\lambda = 4\pi cme^{g^2/2\pi\lambda^2},$$
$$\lambda_0 = 4\pi cm, \qquad (3.34)$$
$$c \equiv \frac{e^{\gamma}}{2\pi}$$

and γ is Euler's constant. Guerin and Fried, 1984, perform a gauge-invariant summation over soft photons exchanged across a fermion loop, by the method of infrared extraction, to obtain for $m/g << 1$

$$\langle \bar{\Psi}\Psi \rangle - \langle \bar{\Psi}\Psi \rangle_0 = \frac{gC}{\sqrt{8\pi^2}} + \frac{m}{2\pi} \ln \frac{m}{gC} + O(m), \qquad (3.35)$$

where C is a real positive constant of order unity. Thus Carson and Kenway predict a logarithmic divergence in $(\langle\bar{\psi}\psi\rangle - \langle\bar{\psi}\psi\rangle_0)/g$ as $m/g \to 0$, whereas Guerin and Fried expect $(\langle\bar{\psi}\psi\rangle - \langle\bar{\psi}\psi\rangle_0)/g$ to have a finite, non-zero value in this strong coupling limit.

To simulate the quenched model we equilibrate the pseudofermions in the fixed background quenched gauge configuration at each value of β and then use (3.17) to calculate $\langle\bar{\psi}\psi\rangle$. The average over the last set out of a total of 15 sets of 100 pseudofermion sweeps, with the free fermion part $\langle\bar{\psi}\psi\rangle_0$ subtracted out, is plotted against the Wilson mass parameter m_W in Fig. 3.5 for β = 0.25, 2.5, 3 and 8. Scaling both axes by g yields Fig. 3.6 which strongly suggests that $(\langle\bar{\psi}\psi\rangle - \langle\bar{\psi}\psi\rangle_0)/g$ is diverging as the Wilson mass parameter m_W approaches a (β-dependent) critical mass m_c, that is, as the physical mass $m = m_W - m_c \to 0$. Fitting the data with Carson and Kenway's prediction yields the critical masses listed in Table 3.3.

Fig. 3.5 $\langle\bar{\psi}\psi\rangle - \langle\bar{\psi}\psi\rangle_0$ against m_W at each β value for quenched model.

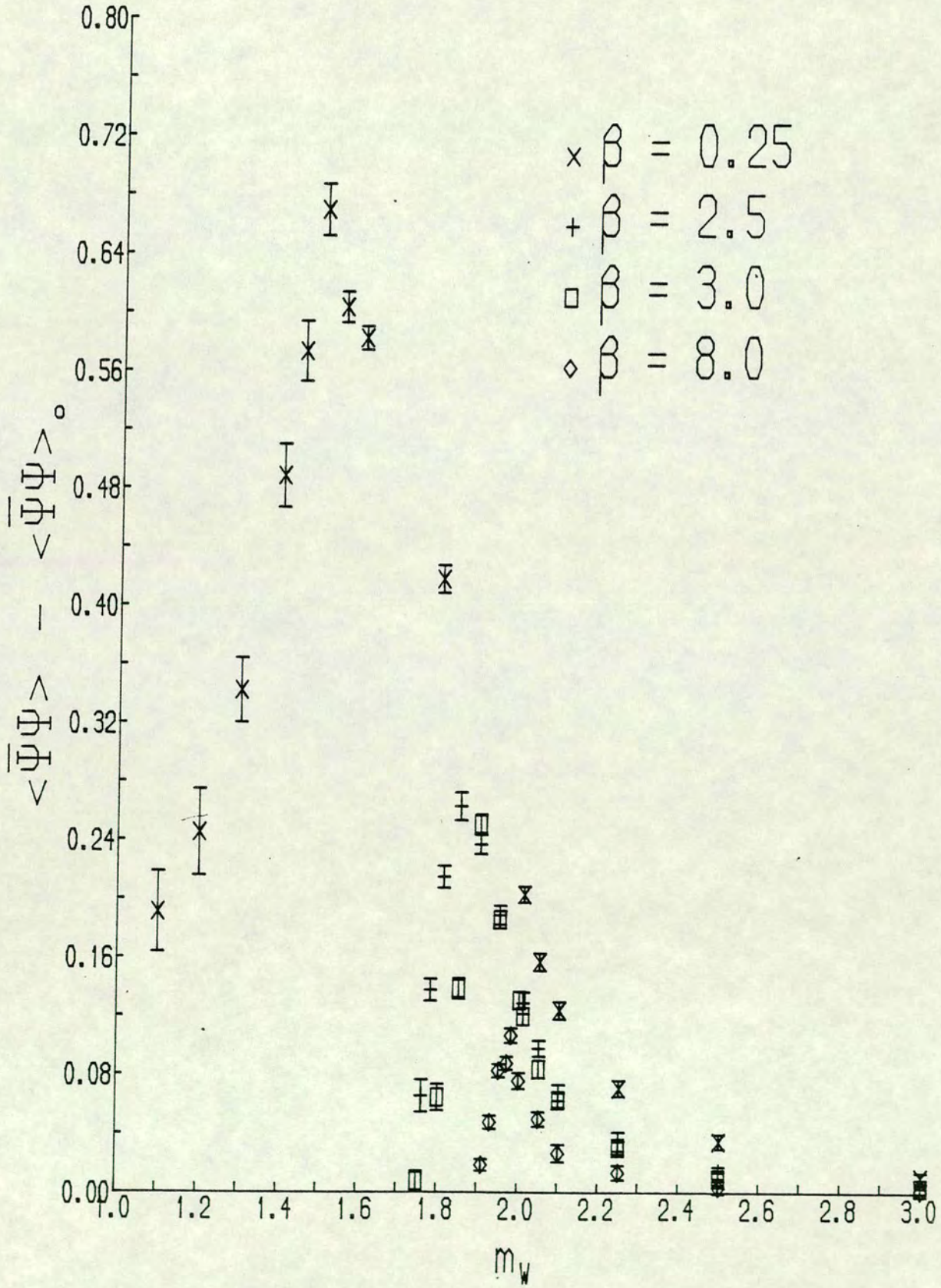


Fig. 3.6 As Fig. 3.5 with axes scaled by g.

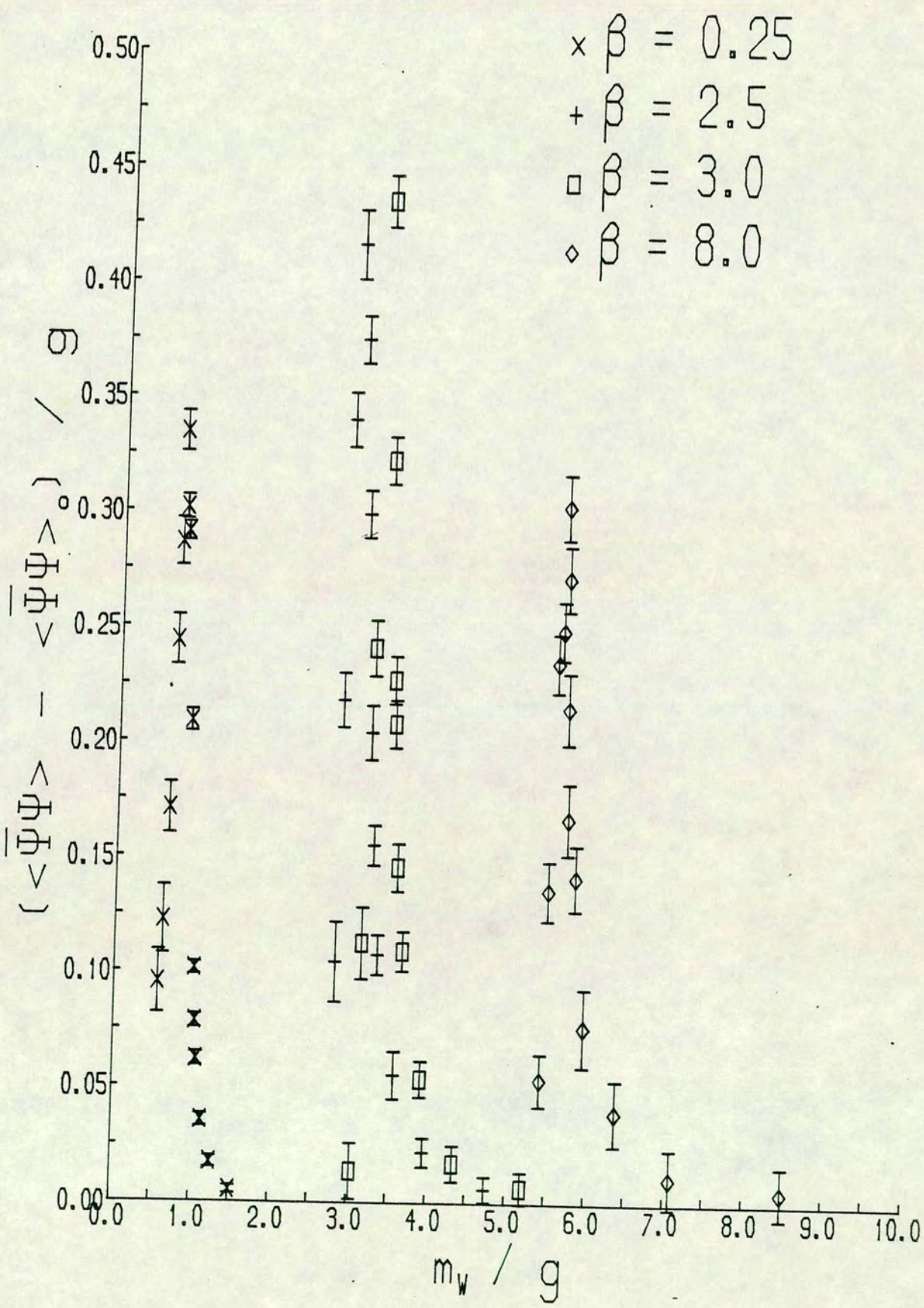


Table 3.3

Critical masses for the quenched model at each β value.

β	m_c
0.25	1.4968 ± .0025
2.5	1.8870 ± .0011
3.0	1.9348 ± .0015
8.0	1.9792 ± .0045

By using these critical masses to shift the data in Fig. 3.6 we obtain Fig. 3.7 which also shows Carson and Kenway's prediction as a solid line. We see good overall agreement with their prediction. It is best for the β = 8 data which corresponds to the smallest lattice spacing and so is nearest the continuum (but also suffers from the largest finite-size effects - since the correlation length ≃ $\sqrt{\pi\beta}$ times the lattice spacing - which causes the more rounded peak). The agreement is not perfect for a number of reasons: there is an error in determining m_c which could shift the data horizontally; $<\bar{\psi}\psi>_0$ calculated by Carson and Kenway is not exactly equal to $<\bar{\psi}\psi>_0$ calculated above since the former is for an infinite system and the latter is for a 64x64 lattice - this could shift the data vertically; and finally the calculation by Carson and Kenway is perturbative to one-loop order, whereas the lattice simulation is non-perturbative and includes all loops.

Finally, the data obtained in the strong coupling limit β = 0 is shown in Fig. 3.8. Despite being far from the continuum due to the large lattice spacing, there is still a peak in $<\bar{\psi}\psi>-<\bar{\psi}\psi>_0$ at a critical mass of about $\sqrt{2}$. This value is predicted by Kawamoto and Smit, 1981, from an effective Lagrangian calculation of meson propagators in U(N), as well as SU(N), lattice gauge theories, for any N with Wilson fermions, in the strong coupling limit. This calculation, which is outlined in the next section, applies here because there is a pseudoscalar in the massive Schwinger model at strong coupling (Sec. 3.1 above).

3.3.5. Effective Lagrangian calculation

Kawamoto and Smit, 1981, derive an effective Lagrangian which describes mesonic bound states in U(N), as well as SU(N), lattice gauge theory at strong coupling. This is then expanded in terms of these bound states about the

Fig. 3.7 $(\langle\bar{\psi}\psi\rangle - \langle\bar{\psi}\psi\rangle_0) / g$ against m / g at each β value for quenched model with Carson and Kenway's prediction as a solid line.

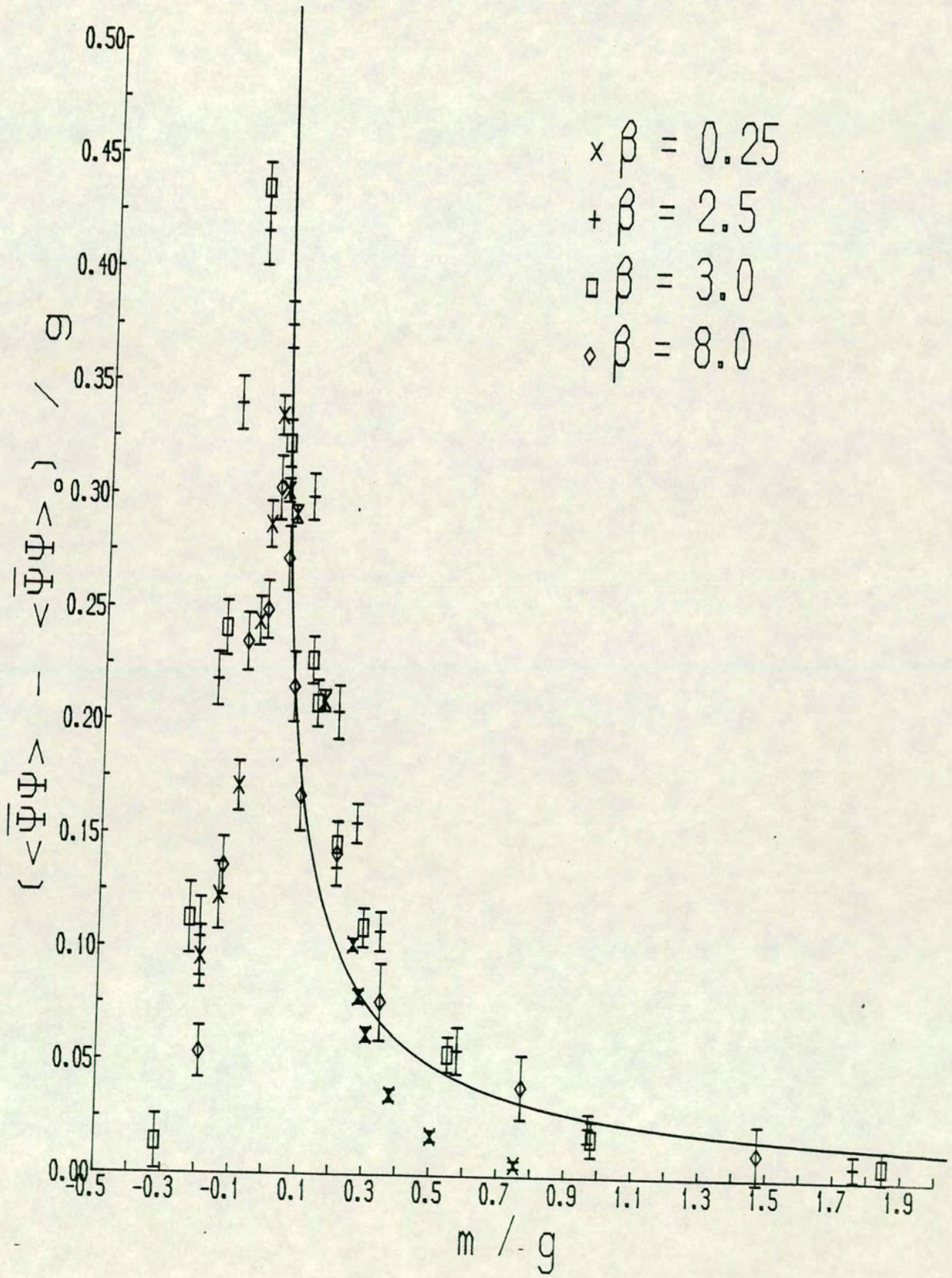
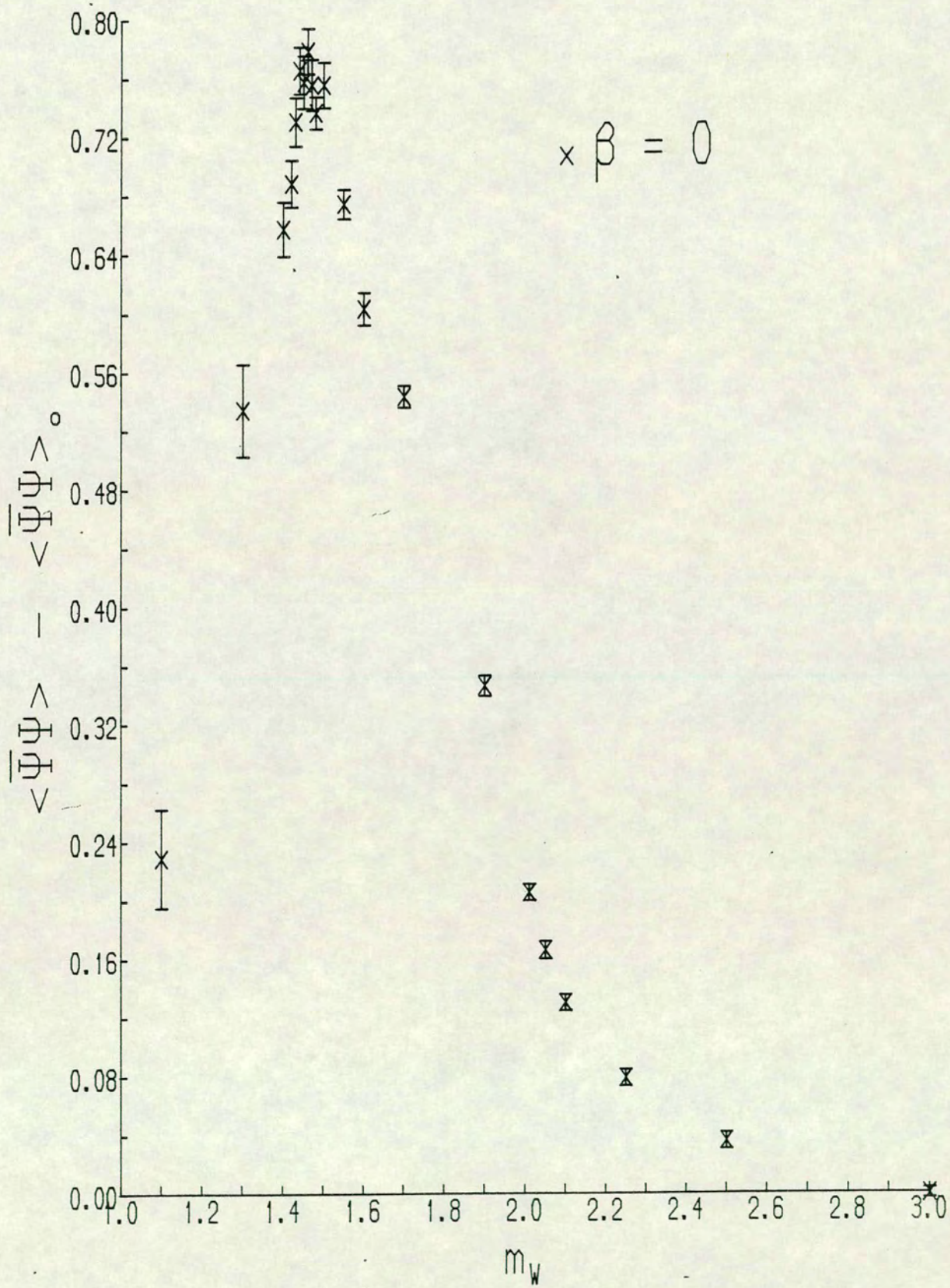


Fig. 3.8 $\langle\bar{\psi}\psi\rangle - \langle\bar{\psi}\psi\rangle_0$ against m_W at strong coupling for quenched model.



vacuum to yield their propagators and m_c is found from the pole in the pion (lightest pseudoscalar bound state or stable particle) propagator. Kawamoto and Smit only look at d = 4 dimensions; we shall keep d explicit so that it can be set equal to 2 at the end of the calculation, which then goes as follows:

Firstly, the source term

$$S_J = \sum_n N J^{\alpha}_{\beta} M^{\beta}_{\alpha}(n), \tag{3.36}$$

where $N M^{\beta}_{\alpha}(n) = \psi^{a\beta}(n) \overline{\psi}_{a\alpha}(n)$ is the elementary Bose field representation of a mesonic bound state (a is colour index and α, β are Dirac indices), is added to the action in (3.1). The resulting partition function is evaluated by first integrating over U and then integrating over $\overline{\psi}$, ψ. (The U integration is only possible analytically for U(N) with N large so the gauge fields must be generalised to $U_\mu(n) \in U(N)$ and the fermions must be given N colours: ψ^a, a = 1,...,N. In our case, for Wilson fermions with r = 1, the final result is independent of N.) This yields the effective action

$$S_{eff} = N \sum_n \mathrm{tr}\, m M(n) + N \sum_{n,\mu} \mathrm{tr} \left[-\frac{1}{2d} \ln \lambda_\mu(n) + F(\lambda_\mu(n)) \right], \tag{3.37}$$

where for Wilson fermions

$$F(\lambda) = 1 - (1-\lambda)^{\frac{1}{2}} + \ln \tfrac{1}{2}\left[1 + (1-\lambda)^{\frac{1}{2}}\right]$$

and

$$\lambda_\mu(n) = (1-r^2) M(n) M(n+e_\mu).$$

Now, parameterising

$$M(n) = v + i N^{-\frac{1}{2}} \phi(n) \tag{3.38}$$

with ϕ containing scalars, pseudoscalars and axial vectors:

$$2\phi = \mathbb{1} S + \gamma_5 P + i \gamma_\alpha \gamma_5 A_\alpha \tag{3.39}$$

leads to

$$S_{eff} = \text{constant} - A \sum_n \text{tr}\, \phi^2(n)$$
$$+ B \sum_{n,\mu} \text{tr}\, \phi(n) P_\mu^- \phi(n+e_\mu) P_\mu^+ + O(\phi^3), \tag{3.40}$$

where

$$A = \frac{m}{2v} + \frac{d}{4}(1-r^2) B \quad ; \quad B = \left[1 - (1-r^2) v^2\right]^{-\frac{1}{2}}$$

and

$$P_\mu^\pm \equiv \frac{1}{2}(r \pm \gamma_\mu),$$

at the stationary value of v which is given by

$$1 = mv + \frac{d(1-r^2)v^2}{1 + \left[1 - (1-r^2)v^2\right]^{1/2}} . \tag{3.41}$$

From this one easily obtains the pion propagator (from the pseudoscalar - axial vector channel) and the following equation for its pole (at $m = m_c$):

$$(1-r^2)^2 - 2(1+r^2)\left[4e - (d-1)r^2\right] + 4r^2$$

$$+ 4\left[2e - \tfrac{1}{2}(d-1)(1+r^2)\right]\left[2e + \tfrac{1}{2}(d-1)(1-r^2)\right] = 0, \quad (3.42)$$

where

$$e \equiv \frac{A}{B}. \quad (3.43)$$

Solving (3.42) gives

$$e = \frac{d}{4}(1+r^2) \quad (3.44)$$

which, with (3.43) and (3.41), yields

$$v^2 = \frac{\left[2d^2 - (1-d)^2(1-r^2)\right] - (d-1)\left[(1-d)^2(1-r^2)^2 + 4d^2r^2\right]^{\frac{1}{2}}}{2d^2}. \quad (3.45)$$

For Wilson fermions with r = 1 this reduces to

$$v^2 = \frac{1}{d} \quad (3.46)$$

and (3.41) becomes

$$m_c = \frac{1}{v} \quad (3.47)$$

hence the result that $m_c = \sqrt{2}$ for d = 2.

3.3.6. The dynamical model

In order to perform a fully interacting dynamical simulation we start from some gauge configuration and some pseudofermion configuration and run the complete pseudofermion method in which the pseudofermions act back on the gauge fields as the system evolves. To check convergence we start from both ordered (gauge variables set to 1 and pseudofermions set to 0) and disordered (random gauge variables and pseudofermions) plotting the plaquette energy (3.32) and the pseudofermion energy against gauge sweep in Fig. 3.9, for $\beta = 3$ and $m = 2.1$. The pseudofermion energy has been defined in terms of (3.14) as

$$E_{pf} \equiv \frac{S_{pf}}{N} , \qquad (3.48)$$

where N is the number of lattice sites. We see that the energies have settled down by 100 gauge sweeps and that it makes no difference whether we start ordered or disordered. This is for $N_{pf} = 1$, that is, only 1 pseudofermion sweep between each gauge sweep. To show the effect of varying N_{pf}, we present Fig. 3.10 in which we do runs with $N_{pf} = 1$, 10 and 100 from an ordered start for $\beta = 3$ and $m = 2$. We see clearly that $N_{pf} = 1$ gives a systematic error, whereas $N_{pf} = 10$ and 100 agree – in fact we use $N_{pf} = 100$ in the dynamical simulation to be on the safe side. We also start from the corresponding quenched gauge configuration instead of an ordered start since this shortens the number of gauge sweeps required to reach equilibrium to less than 10. The other parameter in the pseudofermion method, the update angle δU, is chosen as $0.1 \times 2\pi$. We run at $\beta = 0.25$, 2.5, 3 and 8 for several mass values averaging $<\bar{\psi}\psi>$ over the last 50 sets out of 100 sets of 100 pseudofermion sweeps to obtain Fig. 3.11, the behaviour of $<\bar{\psi}\psi>-<\bar{\psi}\psi>_0$ with Wilson mass parameter in the dynamical model. From the exact massless result of Marinari, Parisi and Rebbi, 1981, that

Fig. 3.9 Plaquette and pseudofermion energies for dynamical model from ordered and disordered starts.

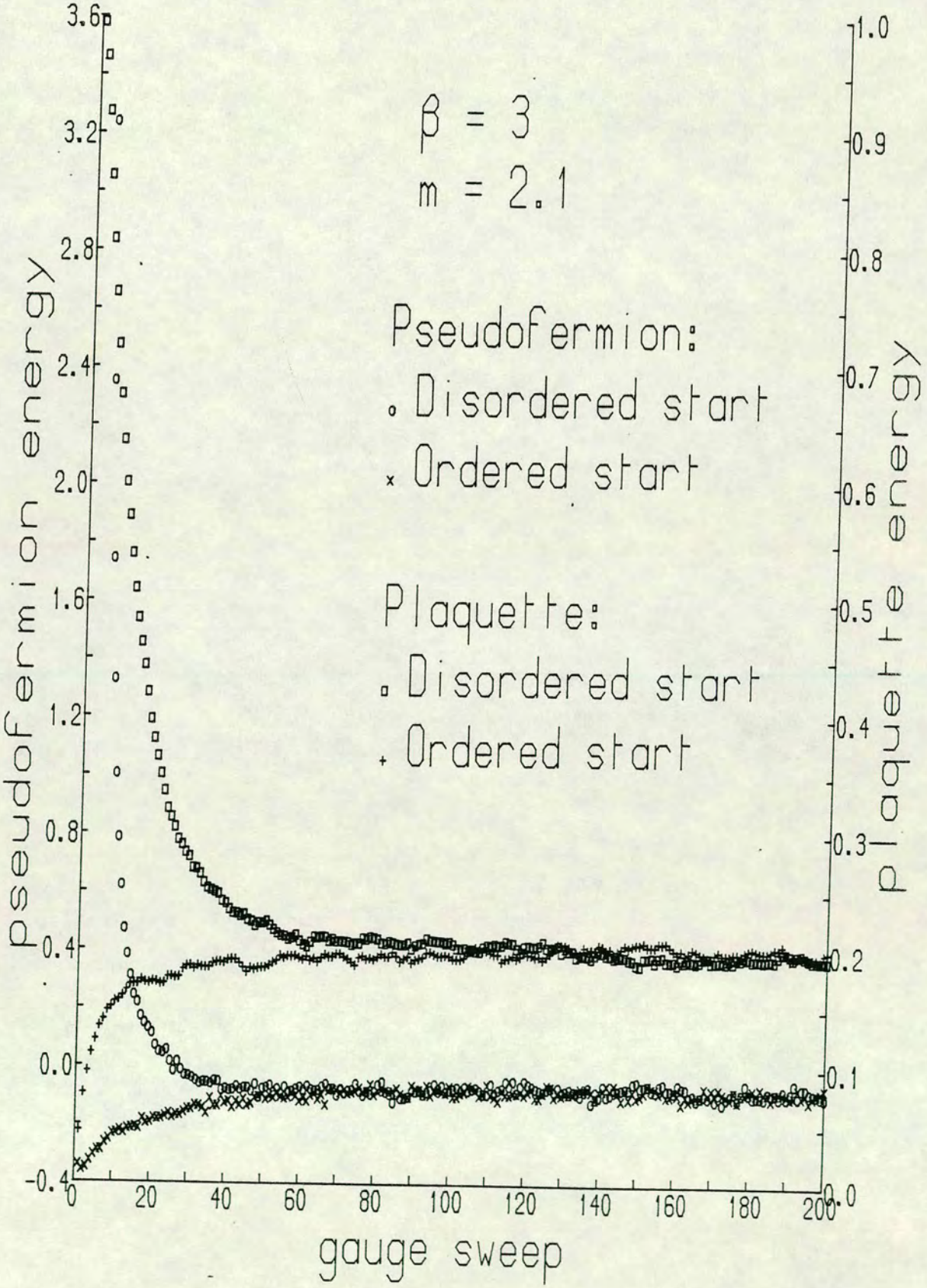


Fig. 3.10 $\langle\bar{\psi}\psi\rangle$ for dynamical model with N_{pf} = 1, 10 and 100 from an ordered start.

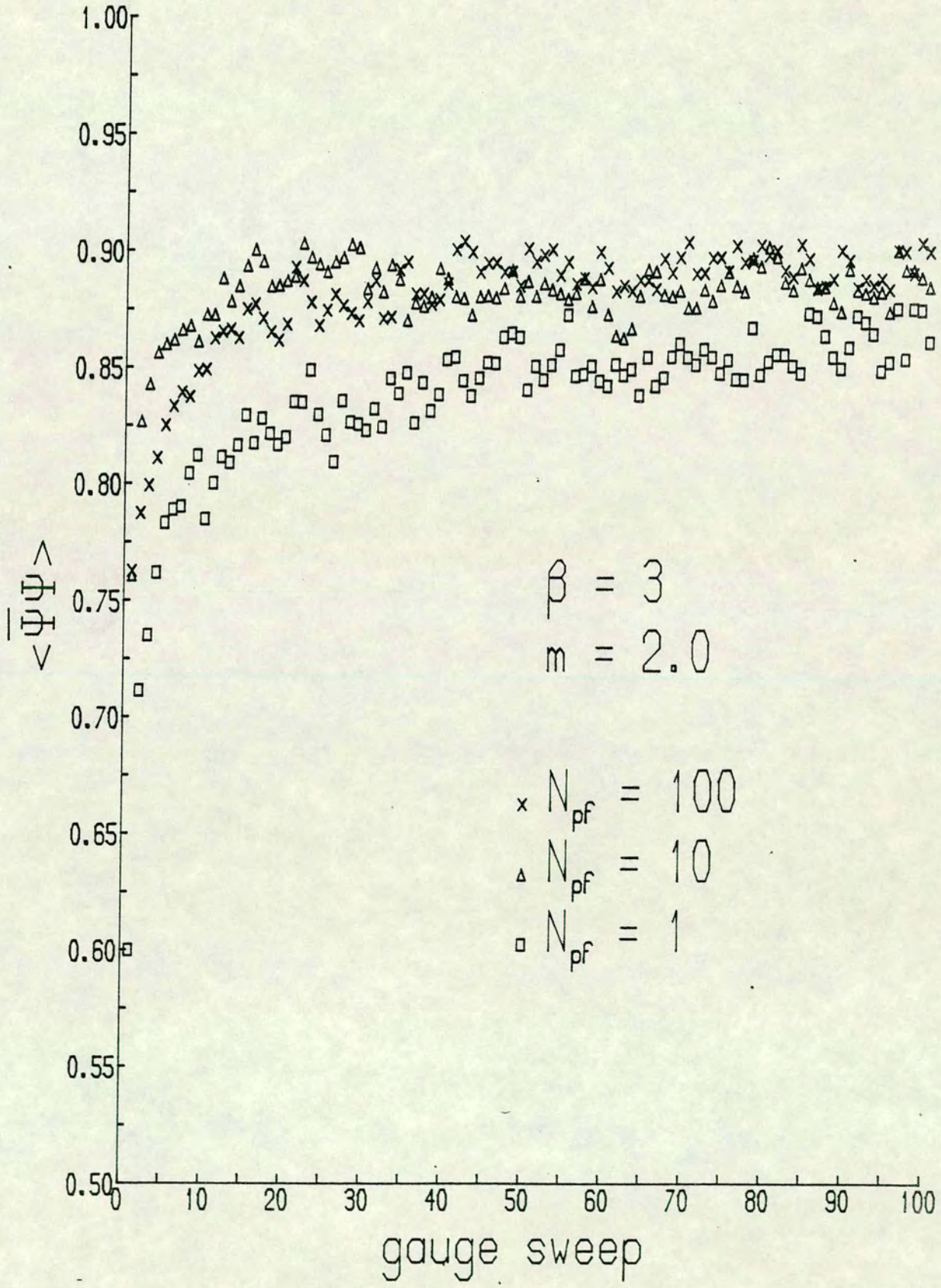
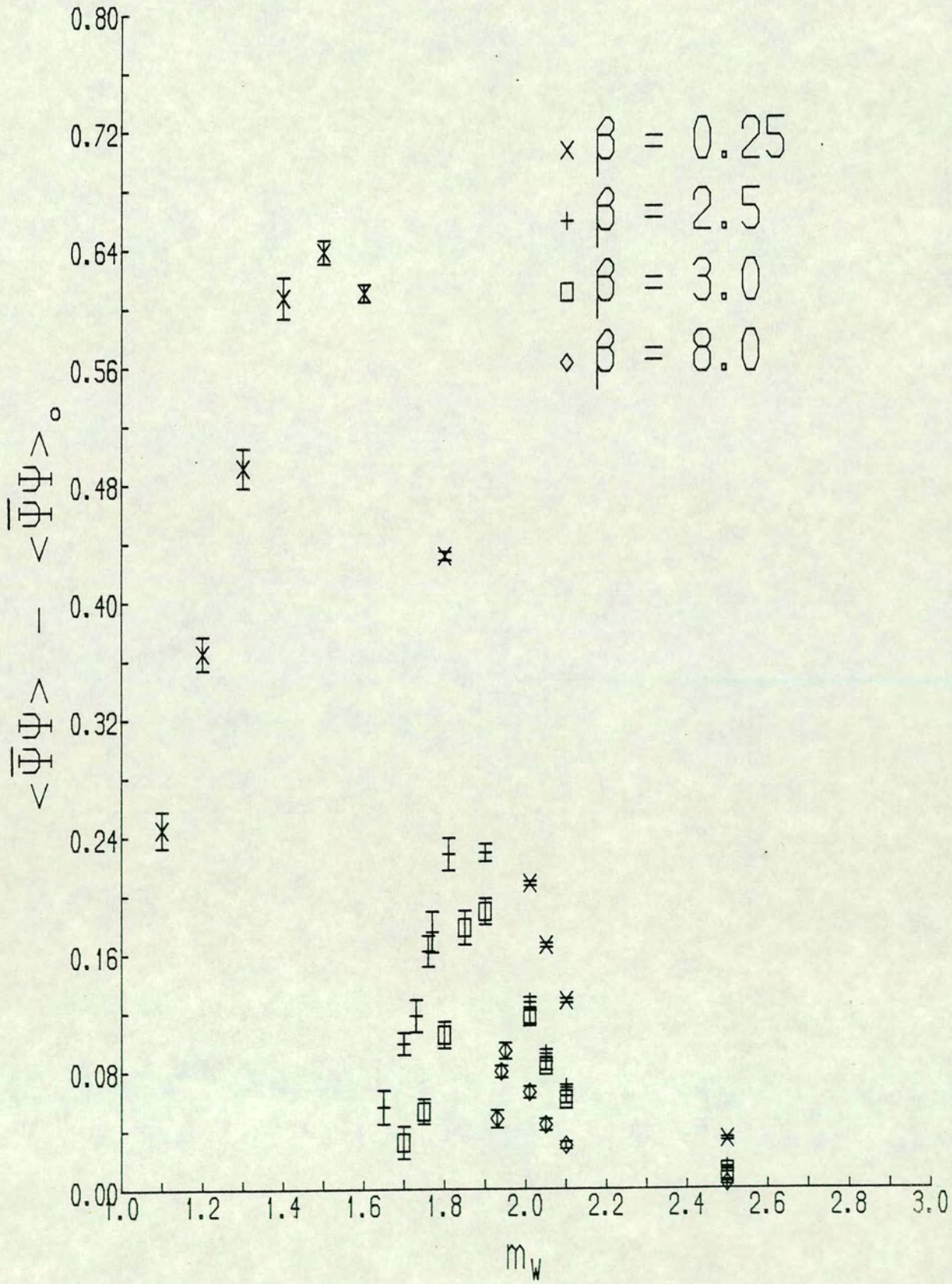


Fig. 3.11 $\langle\bar{\psi}\psi\rangle - \langle\bar{\psi}\psi\rangle_0$ against m_W at each β value for dynamical model.



$$\lim_{m \to 0} \langle \bar{\psi}\psi \rangle - \langle \bar{\psi}\psi \rangle_0 = \frac{g e^{\gamma}}{2\pi^{3/2}}, \qquad (3.49)$$

we obtain the critical masses listed in Table 3.4.

Table 3.4

Critical masses for the dynamical model at each β value.

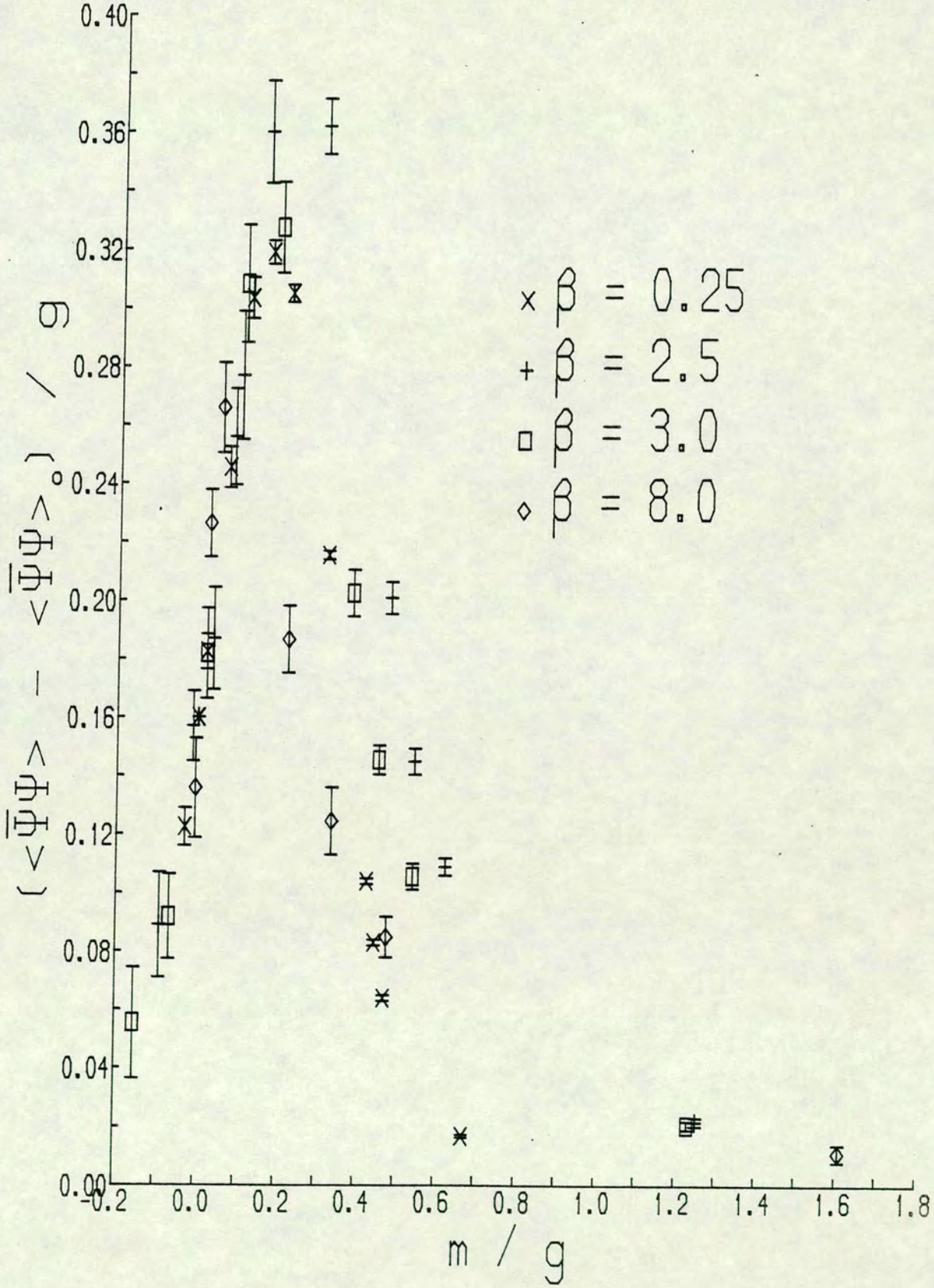
β	m_c
0.25	1.162 ± .018
2.5	1.708 ± .021
3.0	1.789 ± .015
8.0	1.932 ± .006

(Note that there are two possible values for each critical mass but we know that the actual value must be less than that for free fermions (β = ∞) i.e. 2 and greater than that for strong coupling (β = 0) i.e. $\sqrt{2}$, hence it is the smaller of the two.) If we subtract these critical masses from the Wilson mass parameter to yield the physical masses, normalise these by g and plot $\langle\bar{\psi}\psi\rangle - \langle\bar{\psi}\psi\rangle_0$ normalised by g against them, we obtain Fig. 3.12 (which also contains the exact massless result marked as an asterisk). The fact that the data appears to lie on a universal curve at low mass indicates that we have indeed found the correct critical and hence physical masses. For m > 0 $\langle\bar{\psi}\psi\rangle$ appears to increase linearly with m and at large m we know that there is no difference between the dynamical and the quenched model where $\langle\bar{\psi}\psi\rangle$ decreases, so the turnover in $\langle\bar{\psi}\psi\rangle$ around m = 0.2 comes as no surprise. The discrepancies in the data for different β values at m > 0.2 may be due to lattice artifacts since the lattice approximation is only valid for ma << 1, that is, m/g << $\sqrt{\beta}$.

3.3.7. Concluding remarks

We have numerically simulated the massive Schwinger model on a lattice with Wilson fermions and calculated the chiral condensate $\langle\bar{\psi}\psi\rangle$. The pseudofermion method performs well for both the quenched and dynamical

Fig. 3.12 $(\langle\bar{\psi}\psi\rangle - \langle\bar{\psi}\psi\rangle_0)$ / g against m / g at each β value for dynamical model with exact massless result marked as an asterisk.



theory as well as for free fermions, except near the zero mode caused by periodic boundary conditions. For the quenched model, we found that the behaviour for $<\overline{\psi}\psi>/g$ as $m/g \to 0$ agrees with that predicted by Carson and Kenway, namely a logarithmic divergence. For the dynamical model we discovered that $<\overline{\psi}\psi>$ varies linearly with mass for small mass, after using the exact massless result for $<\overline{\psi}\psi>$ to determine the critical mass. We notice that despite the similar appearance of $<\overline{\psi}\psi>$ with Wilson mass parameter for the quenched (Fig. 3.5) and the dynamical (Fig. 3.11) models, the actual behaviour with physical mass is very different (Figs. 3.7 and 3.12): in the quenched model $<\overline{\psi}\psi>$ diverges as the physical mass tends to zero, whereas in the dynamical model it decreases.

Chapter 4
Lanczos fermions

In this chapter we shall use the Lanczos algorithm in a numerical simulation of SU(2) at finite density. In Sec. 1 we describe the Lanczos algorithm for tridiagonalising a Hermitian matrix (in order to obtain its eigenvalues) and we describe the block Lanczos algorithm for inverting a matrix. In Sec. 2 we go on to discuss application of this block Lanczos matrix inversion algorithm to the fermion matrix (enabling dynamical fermion simulations). In Sec. 3 we firstly review finite density in lattice gauge theories (showing how it is introduced as a non-zero chemical potential μ and indicating the significance of the eigenvalue distribution of the fermion matrix) and then turn to the numerical simulation which is performed in two regimes: fixed μ; varying m, and fixed m; varying μ.

4.1. The method

The Lanczos algorithm (Lanczos, 1950) reduces a Hermitian matrix H (of size NxN) to tridiagonal form. It can be derived by seeking the unitary transformation X such that

$$X^{+} H X = T \quad ; \quad X^{+} X = I, \tag{4.1}$$

where T is tridiagonal, real and symmetric:

$$T = \begin{pmatrix} \alpha_1 & \beta_1 & & & \\ \beta_1 & \alpha_2 & \beta_2 & & \\ & \beta_2 & \alpha_3 & \ddots & \\ & & \ddots & & \beta_{N-1} \\ & & & \beta_{N-1} & \alpha_N \end{pmatrix} \tag{4.2}$$

We write X as a set of column vectors x_i, called the Lanczos vectors,

$$X = (x_1, x_2, \ldots, x_N) \tag{4.3}$$

which are orthonormal: $x_i^\dagger x_j = \delta_{ij}$. Hence (4.1) becomes

$$\begin{aligned} H x_1 &= \alpha_1 x_1 + \beta_1 x_2 \\ H x_i &= \beta_{i-1} x_{i-1} + \alpha_i x_i + \beta_i x_{i+1} \; ; \quad 2 \leq i \leq N-1 \\ H x_N &= \beta_{N-1} x_{N-1} + \alpha_N x_N \, . \end{aligned} \tag{4.4}$$

These are the Lanczos equations; they are used recursively to calculate all the α_i, β_i and x_i as follows. Choose x_1 to be any unit vector. Take the scalar product of x_1 with the first Lanczos equation and use the orthonormality of the Lanczos vectors to obtain α_1:

$$\alpha_1 = x_1^\dagger H x_1 \, . \tag{4.5}$$

(α_1 is real because H is Hermitian.) Next calculate

$$\beta_1 x_2 = H x_1 - \alpha_1 x_1 \tag{4.6}$$

and take the scalar product with x_2, using $x_2^\dagger x_2 = 1$, to obtain β_1 and hence x_2. (We can take either sign for β_1.) Continue in a similar fashion with all the other Lanczos equations in turn:

$$\alpha_i = x_i^\dagger H x_i \tag{4.7}$$

$$\beta_i x_{i+1} = H x_i - \beta_{i-1} x_{i-1} - \alpha_i x_i . \tag{4.8}$$

When we calculate

$$\alpha_N = x_N^\dagger H x_N \tag{4.9}$$

we are finished because the last equation is automatically satisfied:

$$u \equiv H x_N - \beta_{N-1} x_{N-1} - \alpha_N x_N \tag{4.10}$$

is zero (as it is orthogonal to all the Lanczos vectors). In fact, a good check on the accuracy of the calculations is that

$$\beta_N \equiv |u| = 0. \tag{4.11}$$

In exact arithmetic, there is only one thing which could cause the algorithm to fail: some β_i might be zero. This will happen if the first Lanczos vector x_1 was chosen to be orthogonal to some eigenvector of H, and it is inevitable if H has a degenerate eigenvalue. The solution would be to choose the next x_i to be any

unit vector orthogonal to all the previous ones and continue - in practice this may be difficult to implement but since it is extremely unlikely to occur (due to rounding errors) we can ignore it.

The advantage of the Lanczos algorithm over other methods (such as Gaussian elimination) is that it does not require the matrix H to be stored in a large NxN array which is "filled in" by the calculation, even if H is sparse (that is, has a large number of zero elements). We only require storage space for three Lanczos vectors and a routine to multiply a vector by H. If H is sparse, the multiplication can be done quickly and with a minimum of storage space. Once H is in tridiagonal form, its eigenvalues can be obtained using standard methods, for example, Sturm sequences.

Before we can use the Lanczos algorithm on large matrices, we must overcome the problem of rounding errors which lead to $\beta_N \neq 0$. This is due to a loss of orthogonality between the first few Lanczos vectors and the last ones. These errors tend to build up exponentially so that no matter what precision is used in the calculation we soon find an x_i which is not orthogonal to x_1. The obvious way to get around this is to reorthogonalise each new Lanczos vector x_i with some or all of the previous ones x_j by the projection

$$x_i \longrightarrow x_i - x_j (x_j^{+} x_i). \tag{4.12}$$

Unfortunately, reorthogonalisation greatly slows down the calculation and requires all the Lanczos vectors to be stored, so it is impractical for $N \gtrsim 1000$. However, it is possible to use the Lanczos algorithm without reorthogonalisation and therefore deal with much larger matrices. This has been discussed by Cullum and Willoughby, 1979; and Haydock, 1983. The procedure is to generate *more* than N Lanczos vectors, say $\tilde{N}$. We then have a $\tilde{N}x\tilde{N}$ tridiagonal matrix $\tilde{T}$. Next, we construct the $(\tilde{N}-1)x(\tilde{N}-1)$ matrix $\hat{T}$ by deleting the first row and column of $\tilde{T}$. From the two sets of eigenvalues (found by the standard method of Sturm sequences), $\{\tilde{\lambda}_i\}$ of $\tilde{T}$ and $\{\hat{\lambda}_i\}$ of $\hat{T}$, we can obtain the N eigenvalues of T (and therefore H) using the observations:

1. Some eigenvalues of H (mainly the ones which are relatively well separated) converge very fast (and, in fact, can be

obtained from $\tilde{T}$ when $\tilde{N}$ is still much smaller than N). By the time $\tilde{N}$ is large enough for all the eigenvalues to have converged (in practice, $\tilde{N} \simeq 2N$ is sufficient), the faster ones will appear many times as eigenvalues of $\tilde{T}$. These duplicates can be recognised and removed because we assume H to be non-degenerate.

2. $\tilde{T}$ and $\hat{T}$ also contain spurious eigenvalues which are not degenerate with the eigenvalues of H. However, these are different for $\tilde{T}$ and $\hat{T}$ and so can also be eliminated.

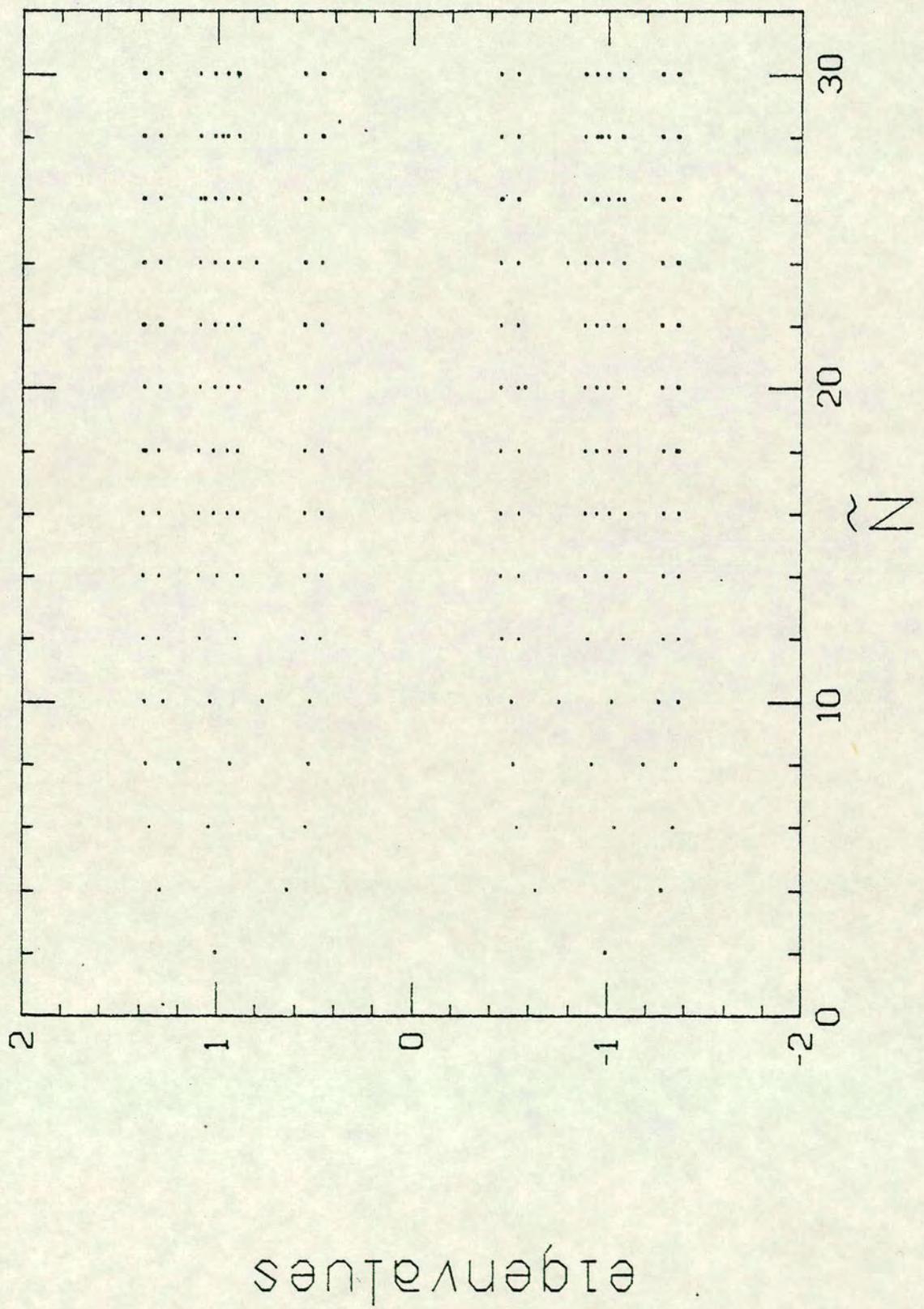
Hence we are left with the N eigenvalues of our original NxN matrix H. As an example to illustrate this, we apply the Lanczos algorithm to the Hermitian matrix $H = i\not{\partial}$, of size 16x16, which is i times the fermion matrix (Dirac operator) for Susskind fermions in random U(1) gauge fields on a 4x4 lattice (this is just two dimensional QED at strong coupling). The resulting eigenvalues for various $\tilde{N}$ are shown in Fig. 4.1. For this small system we can calculate the eigenvalues exactly using a standard library routine. We find that the Lanczos algorithm gives these eigenvalues to within 10^{-6} for $\tilde{N} = N = 16$. From Fig. 4.1, we see that for $\tilde{N} < N$ the smallest eigenvalues converge first and for $\tilde{N} > N$ we get duplicate and spurious eigenvalues (which vary with $\tilde{N}$), as expected.

There is a useful simplification when the Lanczos algorithm is applied to the fermion matrix for Susskind fermions. This is due to the fact that $i\not{\partial}$ has the following block structure between odd and even sites:

$$i\not{\partial} = \begin{pmatrix} 0 & \hat{M} \\ \hat{M}^{+} & 0 \end{pmatrix}. \qquad (4.13)$$

This implies that the eigenvalues of $i\not{\partial}$ come in plus and minus pairs (as we see in Fig. 4.1):

Fig. 4.1 Eigenvalues computed by the Lanczos algorithm, applied to a random Hermitian matrix H with N = 16, for various $\tilde{N}$.



$$\begin{vmatrix} \lambda & \hat{M} \\ \hat{M}^{+} & \lambda \end{vmatrix} = \left| \lambda^{2} - \hat{M}\hat{M}^{+} \right| = 0. \tag{4.14}$$

If we choose the initial Lanczos vector to be zero on all odd sites,

$$\chi_{1} = \begin{pmatrix} \hat{\chi}_{1} \\ 0 \end{pmatrix}, \tag{4.15}$$

then we find that all $\alpha_i = 0$, the odd Lanczos vectors take the form

$$\chi_{2i+1} = \begin{pmatrix} \hat{\chi}_{2i+1} \\ 0 \end{pmatrix} \tag{4.16}$$

and the even ones take the form

$$\chi_{2i} = \begin{pmatrix} 0 \\ \hat{\chi}_{2i} \end{pmatrix}. \tag{4.17}$$

The Lanczos equations then reduce to

$$\left.\begin{aligned} \hat{M}^{+}\hat{\chi}_{1} &= \beta_{1}\hat{\chi}_{2} \\ \hat{M}\hat{\chi}_{2i} &= \beta_{2i-1}\hat{\chi}_{2i-1} + \beta_{2i}\hat{\chi}_{2i+1} \\ \hat{M}^{+}\hat{\chi}_{2i+1} &= \beta_{2i}\hat{\chi}_{2i} + \beta_{2i+1}\hat{\chi}_{2i+2} \end{aligned}\right\} \quad i \geq 1 \tag{4.18}$$

with the even vectors being mutually orthogonal and similarly for the odd ones. The advantage of this is that we have halved the amount of computation, since there is no need to compute α_i and each Lanczos vector is half zero, and we have saved on storage space. Moreover, if we add in the mass term and consider the massive fermion matrix, $iM = i\not{\partial} + im$, we find the same odd-even splitting as above, though with all $\alpha_i = im$ and with Lanczos equations

$$\left.\begin{aligned} \hat{M}^{+}\hat{x}_1 &= im\hat{x}_1 + \beta_1\hat{x}_2 \\ \hat{M}\,\hat{x}_{2i} &= \beta_{2i-1}\hat{x}_{2i-1} + im\hat{x}_{2i} + \beta_{2i}\hat{x}_{2i+1} \\ \hat{M}^{+}\hat{x}_{2i+1} &= \beta_{2i}\hat{x}_{2i} + im\hat{x}_{2i+1} + \beta_{2i+1}\hat{x}_{2i+2} \end{aligned}\right\} \quad i \geq 1. \tag{4.19}$$

Thus the β_i and Lanczos vectors are independent of the mass and we can simultaneously tridiagonalise the matrix at a number of different masses without increased computation.

The Lanczos algorithm can also be used to invert a matrix column by column, which is what we need for our dynamical fermion simulations (Chap. 1.3.4). We shall aim to calculate $H^{-1}x_1$ as a series in the Lanczos vectors

$$H^{-1}x_1 = c_1 x_1 + c_2 x_2 + \ldots \tag{4.20}$$

by using the Lanczos equations iteratively. This is complicated algebraically and explained in detail in Barbour *et al*, 1985b, so we will just illustrate the method by considering the simpler massless case $\alpha_i = 0$. We use only every other Lanczos equation, starting with the second:

$$H^{-1}x_1 = \frac{1}{\beta_1}x_2 - \frac{\beta_2}{\beta_1}H^{-1}x_3 \tag{4.21}$$

in sequence eliminating the remainder term by substitution; this yields

$$H^{-1}x_1 = \frac{1}{\beta_1}x_2 - \frac{\beta_2}{\beta_1\beta_3}x_4 + \frac{\beta_2\beta_4}{\beta_1\beta_3\beta_5}x_6 - \ldots. \tag{4.22}$$

At first sight it seems unlikely that this will converge, since the β_i fluctuate randomly about some constant value. However, in practice we find that although the series proceeds for many iterations without any sign of convergence, it eventually reaches a point (where the smallest eigenvalues of the tridiagonal form are converging to the true eigenvalues of H) at which there is rapid

convergence. Instead of writing down the recurrence relations for this Lanczos matrix inversion algorithm (which are given by Barbour *et al*, 1985b, in any case), we shall first generalise to the block Lanczos algorithm (Scott, 1981) which block tridiagonalises a matrix so that the α_i and β_i become small LxL matrices. The α_i are Hermitian and the β_i can be chosen to be triangular, so that H, with N = ML, is transformed into the following band matrix of width 2L+1:

$$T = \begin{pmatrix} \alpha_1 & \beta_1^+ & & \\ \beta_1 & \alpha_2 & \ddots & \\ & \ddots & & \beta_{M-1}^+ \\ & & \beta_{M-1} & \alpha_M \end{pmatrix}. \qquad (4.23)$$

The M Lanczos vectors are NxL arrays and the Lanczos equations are

$$\begin{aligned} H x_1 &= x_1 \alpha_1 + x_2 \beta_1 \\ H x_i &= x_{i-1} \beta_{i-1}^+ + x_i \alpha_i + x_{i+1} \beta_i \; ; \; i \geq 2. \end{aligned} \qquad (4.24)$$

The algorithm proceeds in a way analogous to the L = 1 case, with the Lanczos vectors half zero for α_i = 0 or α_i = im. At step i $\geq$ 2, with $U \equiv Hx_i - x_{i-1}\beta^{\dagger}_{i-1} - x_i\alpha_i$, we have

$$\beta_i^+ \beta_i = U^+ U \qquad (4.25)$$

which we solve for β_i as an upper triangular matrix, in order to compute

$$\alpha_{i+1} = U \beta_i^{-1}. \tag{4.26}$$

We can now apply block Lanczos to matrix inversion, calculating L rows of the inverse at a time - this is more efficient than inverting the matrix one row at a time because transforming a matrix to block tridiagonal form is less constraining than tridiagonalising it. We obtain (by generalising the L = 1 case with $\alpha_i = im$) the recurrence relations (Barbour *et al*, 1985a)

initial step:

$$\begin{aligned} A_1 &= 1 \\ B_1 &= 0 \\ y_1 &= 0 \\ t_1 &= 1 \\ V_1 &= 0 \\ U_1 &= -\alpha_1 \beta_1^{-1} \end{aligned} \tag{4.27a}$$

even step:

$$\begin{aligned} A_{2i} &= A_{2i-1} + m^2 \left(\beta_{2i-1}^{+}\right)^{-1} B_{2i-1} \\ B_{2i} &= -\beta_{2i} \left(\beta_{2i-1}^{+}\right)^{-1} B_{2i-1} \\ y_{2i} &= y_{2i-1} - A_{2i}^{-1} \left(\beta_{2i-1}^{+}\right)^{-1} t_{2i-1} \\ t_{2i} &= -\beta_{2i} A_{2i-1} A_{2i}^{-1} \left(\beta_{2i-1}^{+}\right)^{-1} t_{2i-1} \\ U_{2i} &= U_{2i-1} + im \alpha_{2i} \left(\beta_{2i-1}^{+}\right)^{-1} B_{2i-1} \\ V_{2i} &= V_{2i-1} + \alpha_{2i} \left(\beta_{2i-1}^{+}\right)^{-1} t_{2i-1} \\ &\quad + im U_{2i} A_{2i}^{-1} \left(\beta_{2i-1}^{+}\right)^{-1} t_{2i-1} \end{aligned} \tag{4.27b}$$

odd step:

$$
\begin{aligned}
A_{2i+1} &= -\beta_{2i+1}\left(\beta_{2i}^{+}\right)^{-1} A_{2i} \\
B_{2i+1} &= B_{2i} - \left(\beta_{2i}^{+}\right)^{-1} A_{2i} \\
y_{2i+1} &= y_{2i} \\
t_{2i+1} &= t_{2i} \\
U_{2i+1} &= U_{2i} + \alpha_{2i+1}\left(\beta_{2i}^{+}\right)^{-1} A_{2i} \\
V_{2i+1} &= V_{2i}
\end{aligned}
\tag{4.27c}
$$

final step:

$$
\left(1 - \beta_1^{-1} m^2 y_{2i+1}\right)^{-1} V_{2i+1} \longrightarrow H^{-1}\alpha_1 . \tag{4.27d}
$$

The coefficients A, B, y and t are all LxL matrices, and U and V are NxL matrices. However, if only a small part of the inverse is required, as is the case for fermion updating (see next section), it is not necessary to compute the whole of U and V but only some KxL block of them.

4.2. Computational details

Following Barbour *et al*, 1985a, we use the block Lanczos algorithm (4.27) to obtain the block of the inverse matrix $M^{-1}(U)$ required to calculate the change in effective action for Monte Carlo simulations with dynamical fermions (Chap. 1.3.4):

$$e^{-\Delta S_{eff}} = e^{-\Delta S_G} \det\left[1 + M^{-1}(U)\,\delta M(U)\right] \quad (4.28)$$

For a SU(N) gauge theory on a lattice of L^d sites, M(U) is a large sparse matrix of size $NL^d \times NL^d$ with only 2dN non-zero elements in each row. If one gauge field link variable is changed then $\delta M(U)$ is non-zero only in the 2Nx2N block at the intersection of the 2N rows and 2N columns of M(U) corresponding to the two end points of the link. Consequently the only elements of $M^{-1}(U)$ which contribute are those in the same 2Nx2N block. If we write $\overline{\delta M}$ and $\overline{M^{-1}}$ for these blocks then

$$e^{-\Delta S_{eff}} = e^{-\Delta S_G} \det\left[1 + \overline{M^{-1}}\,\overline{\delta M}\right], \quad (4.29)$$

where the determinant is now only of a 2Nx2N matrix. The block Lanczos algorithm can be used to calculate 2N columns of $M^{-1}(U)$. This is sufficient to update the same link as many times as desired (in, for example, the multi-hit Metropolis algorithm), since the ratio of determinants for two different changes is

$$\frac{\det(M+\delta M_1)}{\det(M+\delta M_2)} = \frac{\det(1+M^{-1}\delta M_1)}{\det(1+M^{-1}\delta M_2)}. \quad (4.30)$$

This idea can be extended to allow the updating of a number of links at once, for example, in four dimensions we choose all 32 links of a hypercube. To calculate the determinant for the change in effective action arising from any change to these links we require the 16Nx16N block of $M^{-1}(U)$ corresponding to the 2^4 sites of the hypercube. In fact, we need only calculate this block of $M^{-1}(U)$ once, before changing any links, and then update it after each change by rank annihilation, as follows. Consider a change to one link of the hypercube. This causes the change $\overline{\delta M}$ (as in (4.29)) which is a 2Nx2N sub-block of $M^{-1}(U)$ with $(2N)^2/2$ non-zero elements (the other half are zero because they connect each site to itself) which we separate into $(2N)^2/2$ consecutive changes, each to just one element,

$$\overline{\delta M} = \delta M_1 + \delta M_2 + \ldots + \delta M_{(2N)^2/2} . \quad (4.31)$$

We write

$$\delta M_i = a u v^+, \quad (4.32)$$

where a is the change to the element, and u and v are unit vectors which are zero in all elements but one. Hence

$$\begin{aligned} (M+\delta M_i)^{-1} &= M^{-1} - M^{-1}\delta M_i M^{-1} + M^{-1}\delta M_i M^{-1}\delta M_i M^{-1} - \ldots \\ &= M^{-1} - a(M^{-1}u)(v^+M^{-1})\left[1 - av^+M^{-1}u + a^2(v^+M^{-1}u)^2 - \ldots\right] \\ &= M^{-1} - \frac{a(M^{-1}u)(v^+M^{-1})}{1 + av^+M^{-1}u} . \end{aligned} \quad (4.33)$$

The convergence of the series is not relevant since the final result can be verified by back substitution. It is obvious that (4.33) can be applied to update the 16Nx16N block of $M^{-1}(U)$ without knowing the rest of its elements. In practice, for updating a hypercube, we calculate the initial 16Nx16N block of $M^{-1}(U)$ in two 8Nx16N pieces (one to cover the odd sites and the other for the even sites) in two separate inversions, using the block Lanczos algorithm with L = 16N and K = 8N.

To summarise, the Monte Carlo simulation is carried out as follows. To cover all the links in one sweep we must visit 1/8 of all possible hypercubes which touch each other at corners only, so that there are no links in common. We take each of these hypercubes in turn, either in sequence or at random, and calculate the 16Nx16N block of the inverse required to update its links. We then tour each of the 32 links, in any order, extract the appropriate 2Nx2N sub-block from the 16Nx16N block and update the link using the Metropolis algorithm a large number of times (multi-hit), which requires the calculation of only 2Nx2N determinants each time. Before going on to the next link in the hypercube, we update the

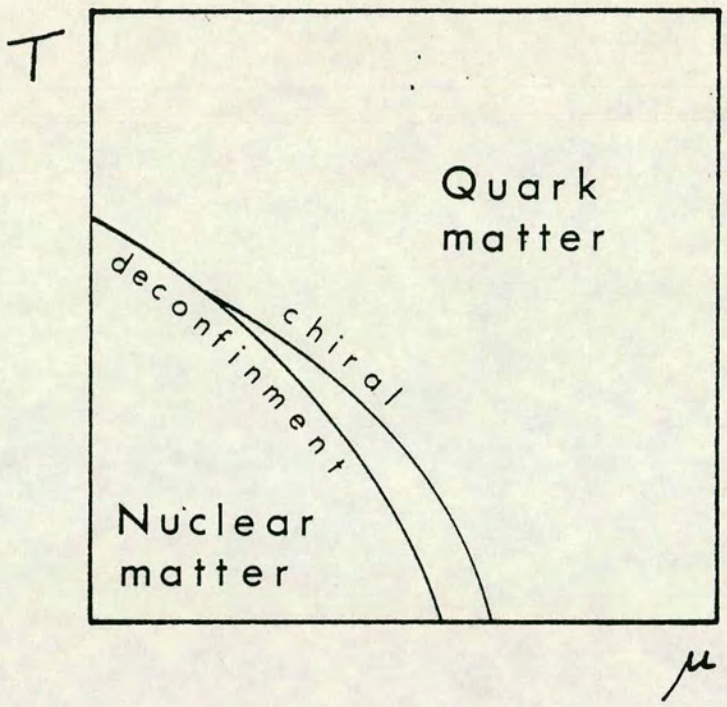
16Nx16N block by rank annihilation (4.33) for the overall change to the link. It proves worthwhile to go round each hypercube a few times until it is close to equilibrium within itself before proceeding to the next, as this brings the whole configuration into equilibrium two or three times faster.

Unfortunately, the block Lanczos algorithm (4.27) is not highly parallel and therefore inefficient on computers such as the ICL DAP (Appendix I) or the GRID (Appendix II). Most of the time is spent repeatedly updating a single link (which involves multiplying LxL and KxL matrices together) and, in order not to violate detailed balance, one can only update two links of a hypercube (and only 1/8 of the hypercubes) simultaneously. Hence we performed the simulation of SU(2) with Lanczos fermions, discussed below, on a conventional computer (the Gould PN9080, in fact).

4.3. Finite density SU(2)

The properties of matter at high temperature and density are important in heavy-ion collisions at high energies and in astrophysical phenomena such as neutron stars (for a review see Cleymans, Gavai and Suhonen, 1986). This has led to considerable interest in what QCD as a theory of strong interactions has to say about these extreme conditions. Analytical calculations, however, are only possible at very high temperatures and densities (where distances are short and energies high enough for asymptotic freedom to make perturbation theory applicable), or at strong coupling which is far from continuum physics. Hence we perform numerical simulations to investigate these effects. QCD at zero temperature and density, as discussed in Chap. 1, confines quarks and has a spontaneously broken chiral symmetry. We shall see that at high temperature and/or density there is a deconfinement transition, producing a quark-gluon plasma, and a chiral symmetry restoration transition, rendering quarks massless. The overall phase diagram is thought to be as depicted in Fig. 4.2.

Fig. 4.2 Schematic phase diagram for QCD at finite temperature and density.



A great deal of work has been done on the effects of finite temperature in both quenched and dynamical QCD (SU(3)) and SU(2) gauge theories. We will investigate the effects of finite density for SU(2), on which relatively little work has been done. (Why we do not look at SU(3) as well will be explained below.)

We introduce finite density, that is, non-zero chemical potential μ, into a system with Hamiltonian H by constructing the conserved baryon number N. The partition function is then given by

$$Z = Tr\, e^{-\beta(H-\mu N)} \qquad (4.34)$$

On the lattice at finite density Hasenfratz and Karsch, 1983, showed that, in order to obtain the correct continuum limit, this leads to the naive free fermion action

$$S_\mu = \sum_n \left\{ \frac{1}{2a} \sum_{i=1}^{3} \left[\bar{\psi}(n) \gamma_i \psi(n+e_i) - \bar{\psi}(n) \gamma_i \psi(n-e_i) \right] \right.$$
$$+ \frac{1}{2a} \left[e^{\mu a} \bar{\psi}(n) \gamma_4 \psi(n+e_4) - e^{-\mu a} \bar{\psi}(n) \gamma_4 \psi(n-e_4) \right] \quad (4.35)$$
$$\left. + m \bar{\psi}(n) \psi(n) \right\}.$$

with similar modification for Wilson and Susskind actions. (Bilic and Gavai, 1984, suggested an alternative formulation.) Unfortunately this leads to a complex fermion determinant for SU(N), N ≠ 2, lattice gauge theories (Gavai, 1985) - whereas all methods known so far for performing numerical simulations with dynamical fermions require a real determinant. Hence at this point in time, we can only meaningfully investigate SU(2) with fermions. (Of course, we can still look at quenched SU(N) for all N.)

The effect of finite density on SU(N) gauge theories with fermions has been investigated analytically at strong coupling by van den Doel, 1984; by Damgaard, Hochberg and Kawamoto, 1985; and by Dagotto, Moreo and Wolff, 1986. They conclude that there is a chiral symmetry restoration transition, which is first order for SU(3) and second order for SU(2), at some critical chemical potential μ_c (in units of lattice spacing, Damgaard, Hochberg and Kawamoto, 1985, predict μ_c = 0.66 for SU(3) and μ_c = 1.04 for SU(2)). There should also be a deconfinement transition which may occur at around the same temperature as the chiral restoration transition. (This appears to be the case at finite temperature.) We can picture this as follows. In the confining phase, any particles produced at finite density will be baryons consisting of N (for SU(N)) quark world lines bound together with an effective chemical potential $N\mu$. As μ is increased there will be a value μ_c such that $N\mu_c$ equals the baryon mass. For $\mu > \mu_c$, it will be favourable for long loops to wind right round the lattice in the time direction yielding a finite density of baryons. Thus we expect μ_c to be equal to the mass of the lowest baryonic state divided by its quark number, that is, one third the mass of the nucleon in SU(3) and one half the mass of the pion in SU(2).

The first simulation of quenched QCD at finite density (Kogut *et al*, 1983) found, by extrapolation to zero quark mass, an abrupt restoration of chiral symmetry at $\mu_c \simeq 0.3$. However, further investigation (Barbour *et al*, 1986) revealed that at zero quark mass chiral symmetry is restored for any $\mu > 0$. Moreover, for non-zero quark mass μ_c was found to be $m_\pi/2$ rather than

$m_{nucleon}/3$, which suggests that $m_{nucleon} = 1.5m_\pi$ so that the lowest baryonic state in QCD becomes massless (like the pion) at zero quark mass! In contrast, the same result, $\mu_c = m_\pi/2$, found for SU(2) is as expected. This seems to imply that there is something wrong with finite density calculations in the quenched approximation, for QCD at least. Gibbs, 1986, argues that the quenched approximation actually becomes invalid for $\mu > m_\pi/2$. The obvious way to proceed is to add quarks and simulate the full theory. Engels and Satz, 1985, attempted this for QCD (using the leading term in the hopping parameter expansion) by ignoring the imaginary part of the complex determinant - they find that the temperature at which deconfinement occurs decreases as μ increases, in agreement with the expected phase diagram (Fig. 4.2). More work is required to ascertain the validity of their approach and, of course, to discover better methods for dealing with the complex determinant (see Gibbs, 1986, for a discussion of the latter).

Turning to SU(2) we find the same story. Kogut *et al*, 1983, also investigated the chiral symmetry restoration transition in the quenched approximation for SU(2), obtaining a smooth, presumably second order, transition around $\mu_c \simeq 0.3 - 0.45$. However, Dagotto, Karsch and Moreo, 1986, subsequently found that (as in quenched QCD) chiral symmetry is restored for all non-zero μ in the massless limit. The deconfinement transition was examined by Nakamura, 1984, but his results were inconclusive. Again, what is required is a simulation including quarks. This is possible and meaningful using standard Monte Carlo methods for SU(2), as the determinant is real, so we shall undertake it. First, we will outline the importance of eigenvalues (and hence the Lanczos algorithm) for such a finite density simulation.

A clue to what is happening in finite density simulations is given by the distribution of eigenvalues of the lattice Dirac operator in the background gauge fields. The Dirac operator for Susskind fermions is anti-Hermitian (Chap. 3.2) and has purely imaginary eigenvalues λ_k, in terms of which the chiral symmetry order parameter $<\bar{\psi}\psi>$ is given by

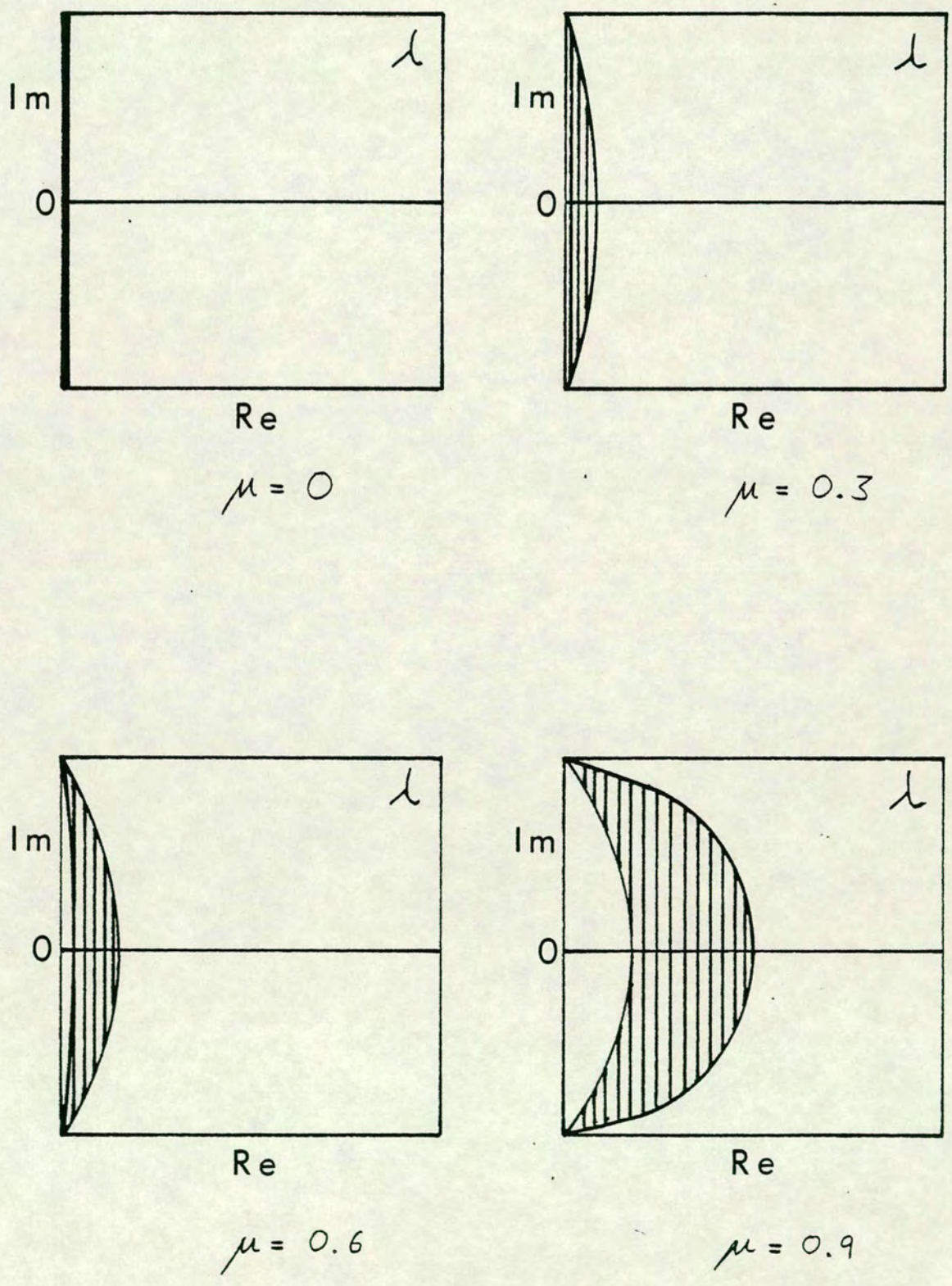
$$\langle \bar{\psi}\psi \rangle = \langle \mathrm{tr}\,(\not{D}+m)^{-1} \rangle = \left\langle \sum_k \frac{1}{\lambda_k + m} \right\rangle . \qquad (4.36)$$

(The trace is over colour.) In the infinite volume limit the eigenvalues coalesce to form a cut which provides the discontinuity in $<\bar{\psi}\psi>$ at m = 0:

$$\langle \bar{\psi}\psi \rangle_{m=0+} - \langle \bar{\psi}\psi \rangle_{m=0-} = N\pi\rho(0), \qquad (4.37)$$

where N is the number of colours and $\rho(\lambda)$ is the normalised eigenvalue density on the imaginary axis. (This expression has been used to obtain strong evidence for spontaneous chiral symmetry breaking in (quenched) QCD at zero density: Barbour *et al*, 1983; Barbour *et al*, 1984; Barbour, Gibbs, Bowler and Roweth, 1985.) If we allow a non-zero value for μ then the anti-Hermitian nature of $\not{D}$ is lost and the eigenvalues move off the imaginary axis, initially in a perpendicular direction. In practice, evaluation of the eigenvalue distribution by use of the Lanczos algorithm shows that (Barbour *et al*, 1986), at all couplings, the eigenvalues move off axis to form a roughly uniform strip whose width increases monotonically with μ. Eventually, for larger μ, the eigenvalues form a band, leaving the region around $\lambda = 0$ empty of eigenvalues; this is shown schematically (for SU(3) at strong coupling) in Fig. 4.3. We see that for all μ there is a $\lambda_{max} = \max |\lambda|_{\mathrm{Im}\lambda=0}$, and for $\mu > \mu_o$ (with $\mu_o = 0.5 \pm 0.05$ for this case) there is a $\lambda_{min} = \min |\lambda|_{\mathrm{Im}\lambda=0}$. These maximal and minimal eigenvalues on the real axis are directly related to the behaviour of physical observables. For quark masses $m > \lambda_{max}$ all observables will agree with their $\mu = 0$ values, while for $m < \lambda_{min}$ they will have reached their limiting high density values. In particular, $<\bar{\psi}\psi> = 0$ for all $m < \lambda_{min}$. Thus in order to investigate the chiral symmetry restoration transition we should discuss what happens for $\mu < \mu_o$, that is, when $\lambda_{min} = 0$. We should then find some effect on $<\bar{\psi}\psi>$ as the quark mass is brought inside the strip (from $m > \lambda_{max}$ to $m < \lambda_{max}$) since $<\bar{\psi}\psi>$ at mass m is effectively determined by the small eigenvalues less than m. Alternatively, if we vary μ at a non-zero value of m then there is a critical value μ_c where the width of the strip becomes equal to the quark mass. For $\mu < \mu_c$, $<\bar{\psi}\psi>$ is independent of μ but at μ_c there is a transition and $<\bar{\psi}\psi>$ drops

Fig. 4.3 Eigenvalue distribution (for SU(3) at $\beta = 0$) for different values of the chemical potential.



rapidly. We note in passing that the fact that the delta-function in eigenvalue distribution for $\mu = 0$ becomes a uniform strip for $\mu > 0$ explains why chiral symmetry breaking disappears at any finite density. Barbour *et al*, 1986, also find that as the inverse coupling β is increased the eigenvalues move away from the real axis; this is the case for any μ, with the eigenvalues still occupying the appropriate strip or band about the imaginary axis. This is shown schematically (for SU(3) at small μ) in Fig. 4.4. To conclude, in a simulation of dynamical finite density SU(2) we expect to find the chiral symmetry restoration, and perhaps the deconfinement, transition at around $\mu_c = m_\pi/2$, signalled by $<\bar{\psi}\psi>$ dropping to zero and the eigenvalues of the fermion matrix moving away from the real axis.

4.3.1. Fixed μ; varying m

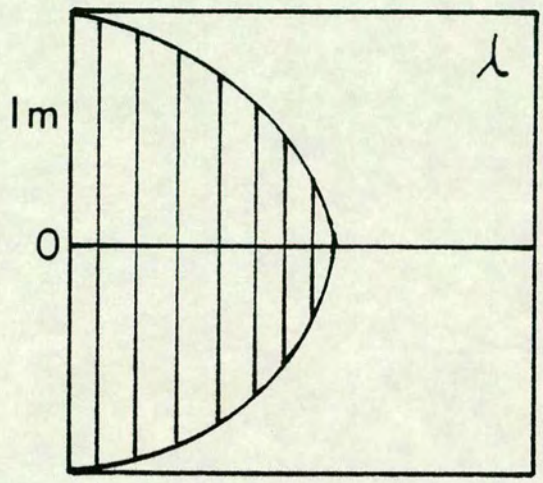
As this simulation is performed using a conventional computer (the Gould PN9080), we study a small lattice of 4^4 sites. The full dynamical fermion Lanczos algorithm then takes 2.65 hours for one sweep through the lattice, going round each of the 32 hypercubes (touching at corners only) 4 times and performing the multi-hit Metropolis algorithm with 10 hits on each of the links. (We note in passing that a similar sweep of an 8^4 lattice would take approximately 600 hours - over 3 weeks - on this computer!) This algorithm converges much faster than the pseudofermion method so we need only carry out tens rather than hundreds of sweeps to achieve statistical equilibrium.

We choose the number of fermion flavours $n_f = 4$, set the inverse coupling $\beta = 1.7$ and investigate fixed $\mu = 0.1$; varying m. The history of the chiral condensate $<\bar{\psi}\psi>$ and the plaquette

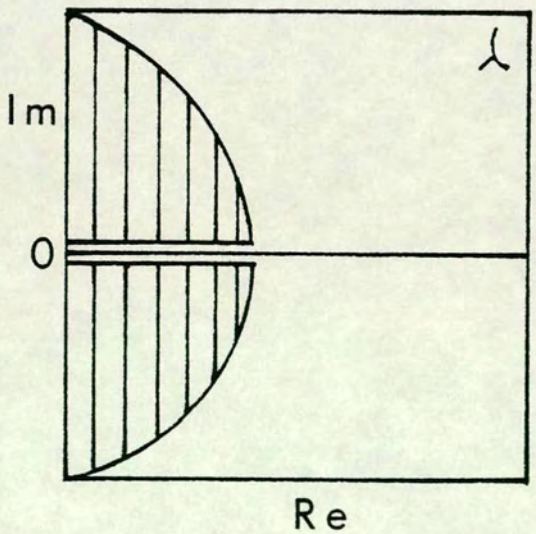
$$P_\square \equiv Re\, U_\square \tag{4.38}$$

(not the plaquette energy $E_\square = 1 - P_\square$ used in Chap. 3.3.2) during the simulation is given in Fig. 4.5. Firstly, we performed 5 quenched sweeps (-4, -3, -2, -1 and 0) at $\beta = 2.1$ to generate an appropriate start configuration for the dynamical sweeps (1-75). The first 20 of these (1-20) were done at fermion mass m = 0.05, the next 30 (21-50) at m = 0.0125 and the last 25 (51-75) at m = 0.00625. If we average $<\bar{\psi}\psi>$ over the last 10 sweeps at each mass, that is, 11-20 at m = 0.05,

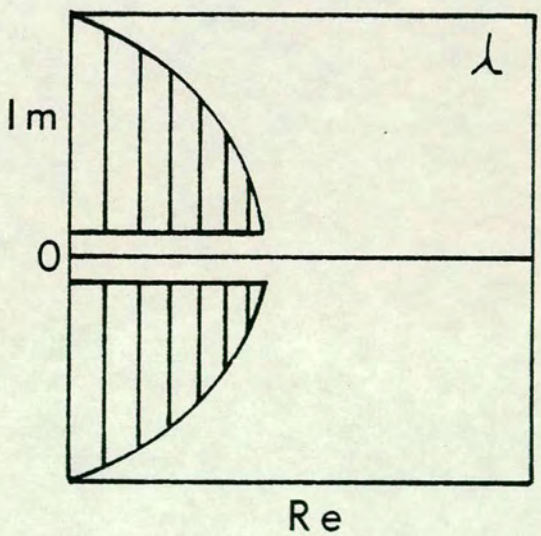
Fig. 4.4 Eigenvalue distribution (for SU(3) at $\mu = 0.1$) for different values of the inverse coupling. (Scale of real axis is expanded relative to Fig. 4.3.)



$\beta = 0$



$\beta = 5.6$



$\beta = 6.0$

Fig. 4.5 $\langle\bar{\psi}\psi\rangle$ and plaquette history of simulation at $\mu = 0.1$.

31-50 at 0.0125 and 56-75 at 0.00625, then we get the values in Table 4.1.

Table 4.1

$\langle\bar{\psi}\psi\rangle$ for $\mu = 0.1$.

m	$\langle\bar{\psi}\psi\rangle$
.05	.169 ± .033
.0125	.049 ± .018
.00625	.014 ± .002

Thus $\langle\bar{\psi}\psi\rangle$ is consistent with extrapolating to zero at zero fermion mass as it must on a finite lattice.

We now look at the eigenvalue distributions (in the complex plane) which are given for the three masses in Figs. 4.6a,b,c. We plot superimposed the eigenvalues for the last 5 configurations, at each mass. As the eigenvalues occur in complex conjugate pairs we only plot half of them (those with imaginary part > 0) - the other half can be obtained by reflection in the real axis. We see that the distributions for m = 0.05 and m = 0.0125 are very similar but the distribution for the lowest mass m = 0.00625 appears to have a lower density of eigenvalues around the real axis. We can investigate this further as follows. Write the eigenvalues $\lambda = x + iy$ and use the fact that they occur in complex conjugate pairs to rewrite the sum in (4.36) as a sum over half the eigenvalues

$$\begin{aligned} sum &= \sum_{\frac{k}{2}} \frac{1}{x+iy+m_s} + \frac{1}{x-iy+m_s} \\ &= \sum_{\frac{k}{2}} \frac{2(x+m_s)}{(x+m_s)^2+y^2} . \end{aligned} \tag{4.39}$$

(We have denoted the mass appearing in this sum m_s to distinguish it from the fermion mass used in the simulation m.) Now if the eigenvalue density is uniform across the strip and the width of the strip is constant then this sum is independent of the real part x and therefore, for a given m_s, determined solely by the imaginary part y which is the distance of the eigenvalues from the real axis. (The width of the strip varies only for varying μ; this case is discussed in the next section.) Hence we can use (4.39) to discover if the eigenvalues are moving

Fig. 4.6a Eigenvalue distribution (5 configurations superimposed).

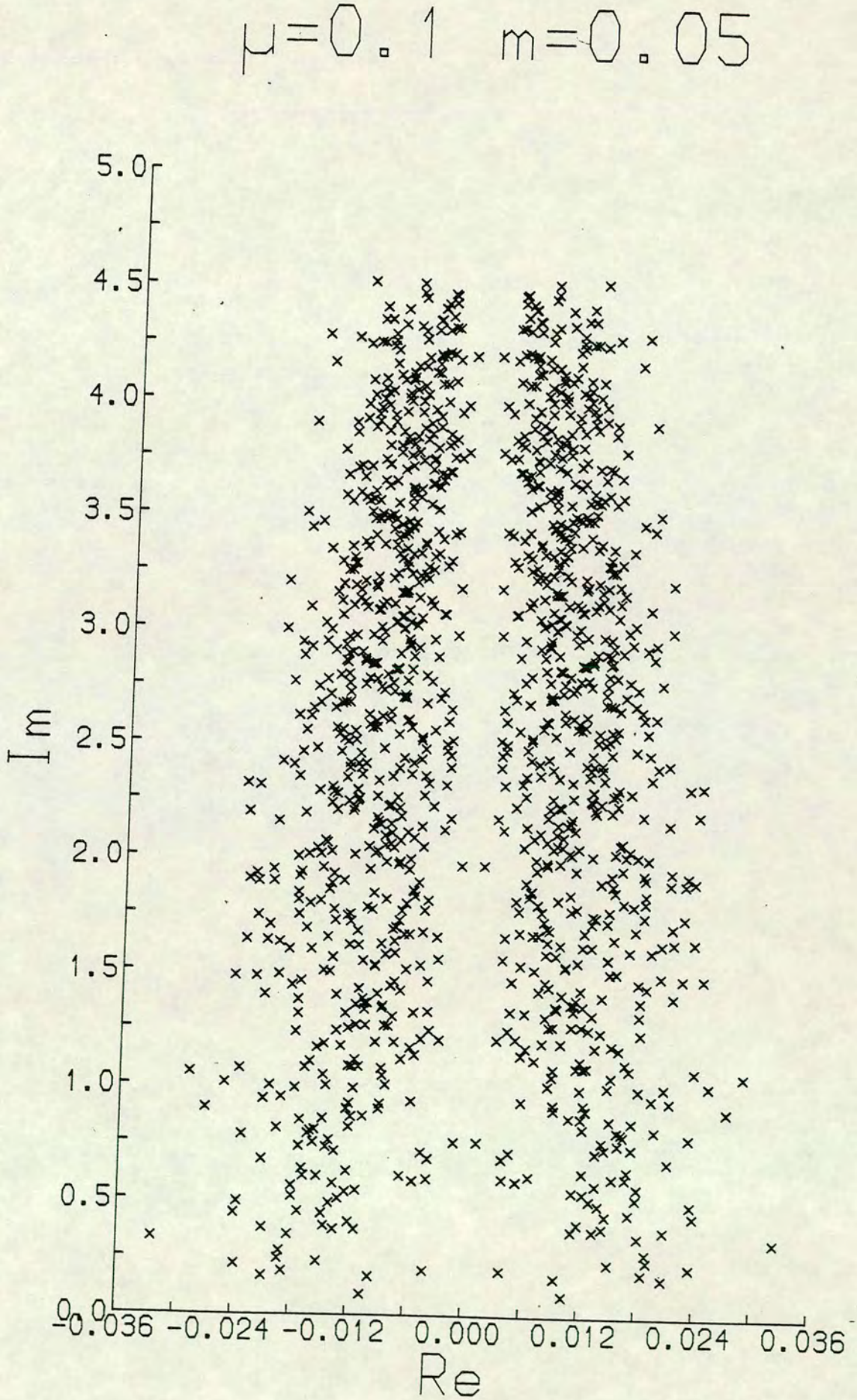


Fig. 4.6b Eigenvalue distribution (5 configurations superimposed).

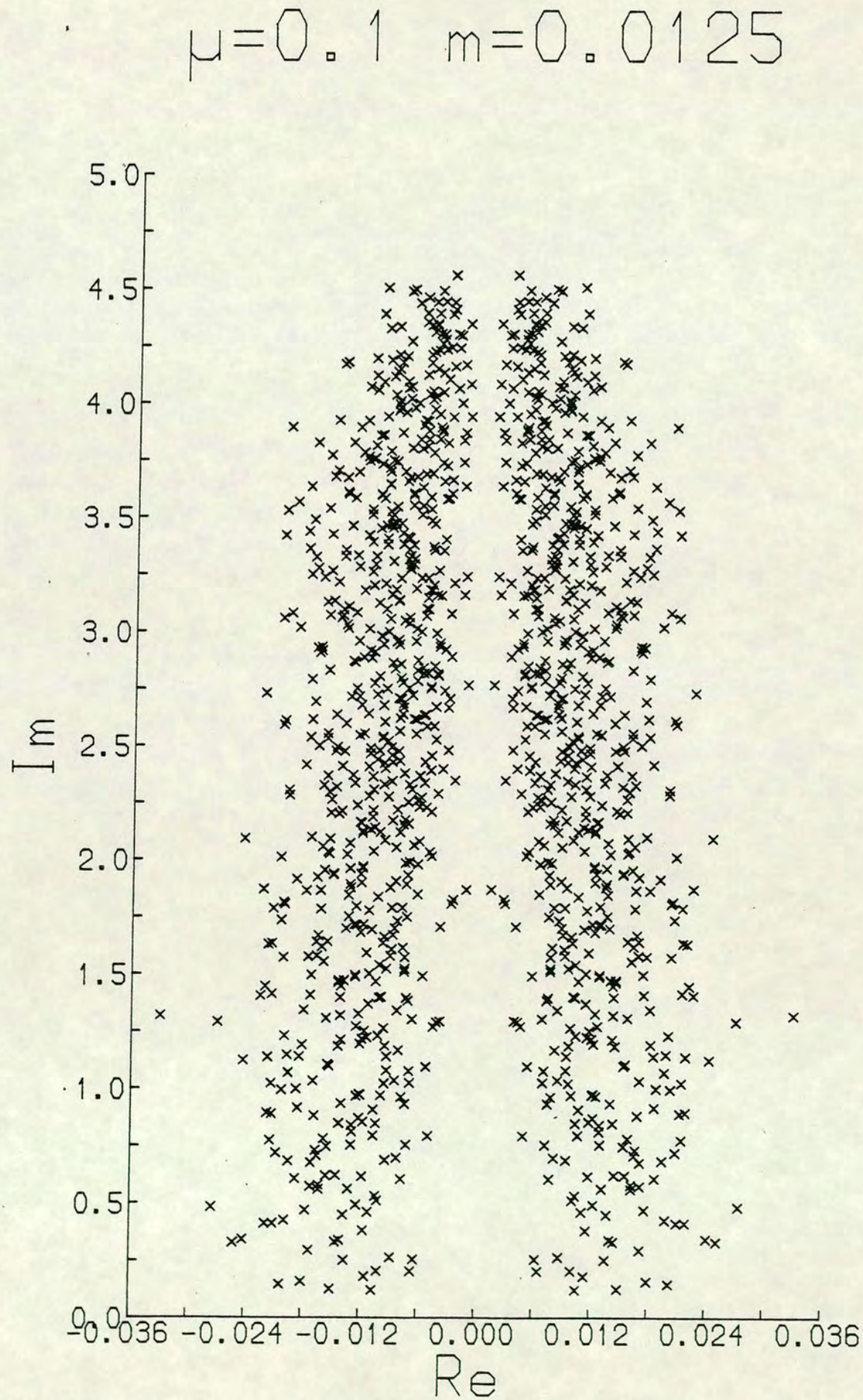
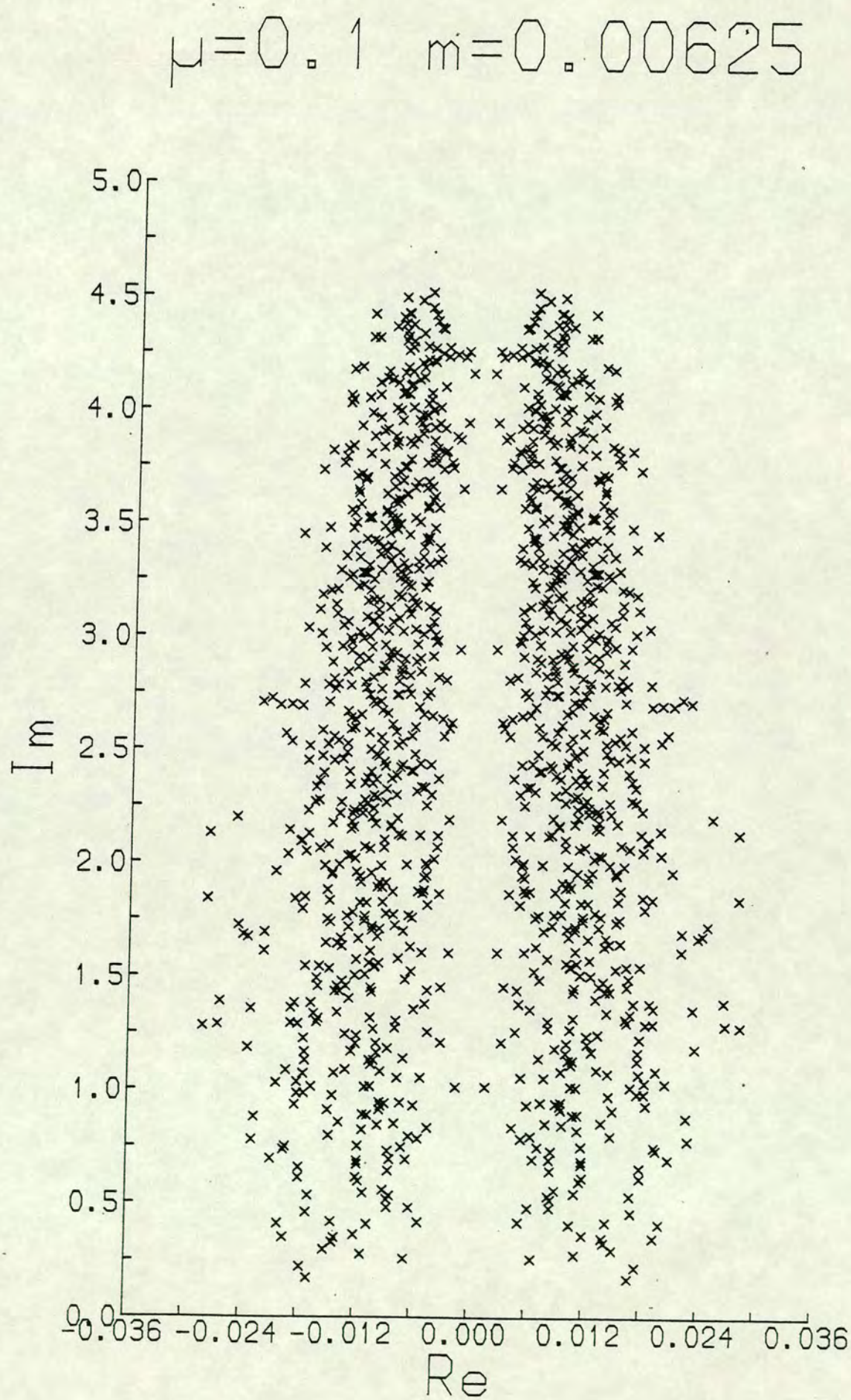


Fig. 4.6c Eigenvalue distribution (5 configurations superimposed).



away from the real axis as the fermion mass is reduced. In fact, we vary m_s and calculate the sum for only those eigenvalues whose real part is such that $x + m_s \geq 0$. In other words, we scan across the strip summing the eigenvalues to the right of the line $x = -m_s$; effectively verifying that the eigenvalue density is uniform. The resulting "sum ≥ 0", averaged over the last 5 configurations at each mass, is plotted in Fig. 4.7a. It is clear that the sum is smaller for the lowest mass implying that y is larger and the eigenvalues are further from the real axis. The sums for the two larger masses are indistinguishable within the errors, even though these masses differ by a factor of four. To show that the behaviour of the sum, that is, $<\bar{\psi}\psi>$, is determined mainly by the smallest eigenvalues we plot the "sum ≥ 0" calculated from the lowest 20 eigenvalues of each configuration in Fig. 4.7b. The observation that the eigenvalues move away from the real axis when the fermion mass is reduced by a factor of two from 0.0125 to 0.00625, as well as $<\bar{\psi}\psi>$ decreasing, supports the conjecture of a phase transition (chiral symmetry restoration and/or deconfinement) induced by the fermion mass moving inside the eigenvalue strip.

One could of course argue that the eigenvalue density falls near the real axis because the Monte Carlo method used in the simulation simply does not generate any configurations with eigenvalues there, since the weight involves detM which is proportional to the smallest eigenvalue. In order to rule out this possibility we repeated the simulation with a different weight in the Monte Carlo method. Up to now we have been calculating

$$\langle \bar{\psi}\psi \rangle = \langle \mathrm{tr}\, M^{-1} \rangle = \frac{\int \mathcal{D}U \, \mathrm{tr} M^{-1} \det M \, e^{-S_G}}{\int \mathcal{D}U \det M \, e^{-S_G}} \quad (4.40)$$

which uses the weight detM $\exp(-S_G)$. Instead, we can write this as

Fig. 4.7a Sum of eigenvalues, for each m (averaged over 5 configurations), with real part x such that $x + m_s \geq 0$.

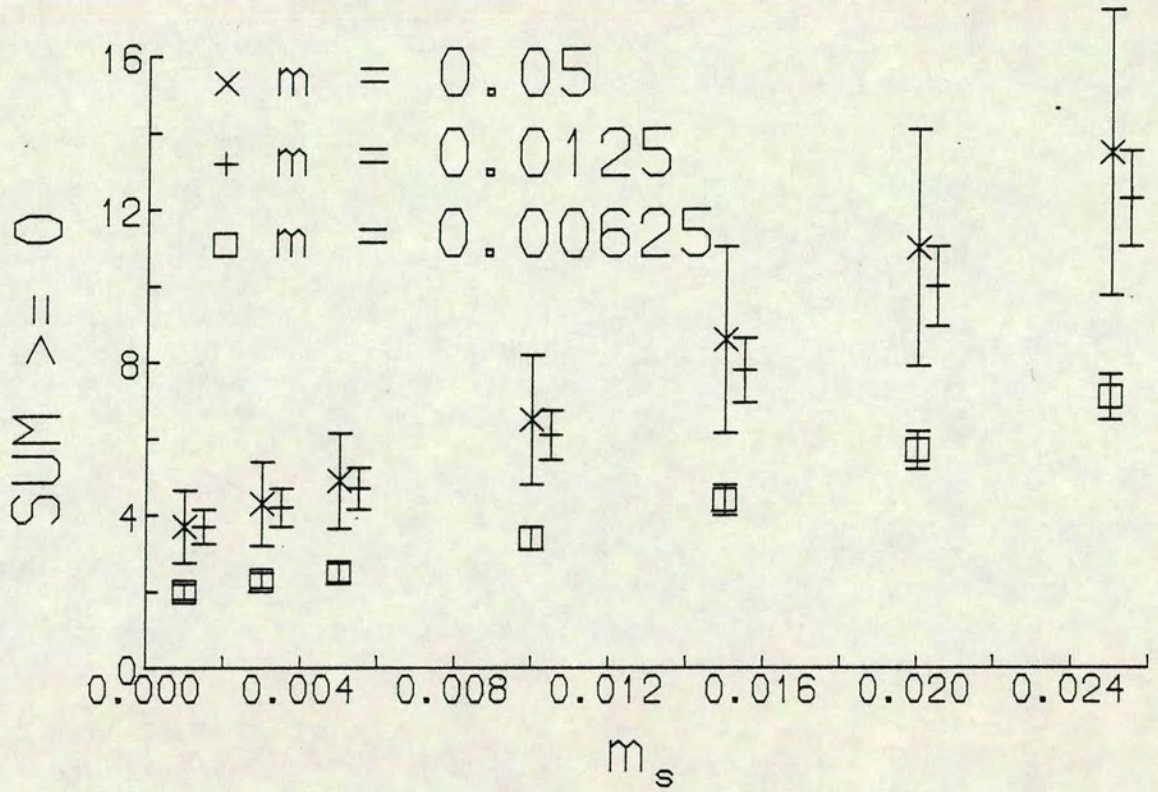
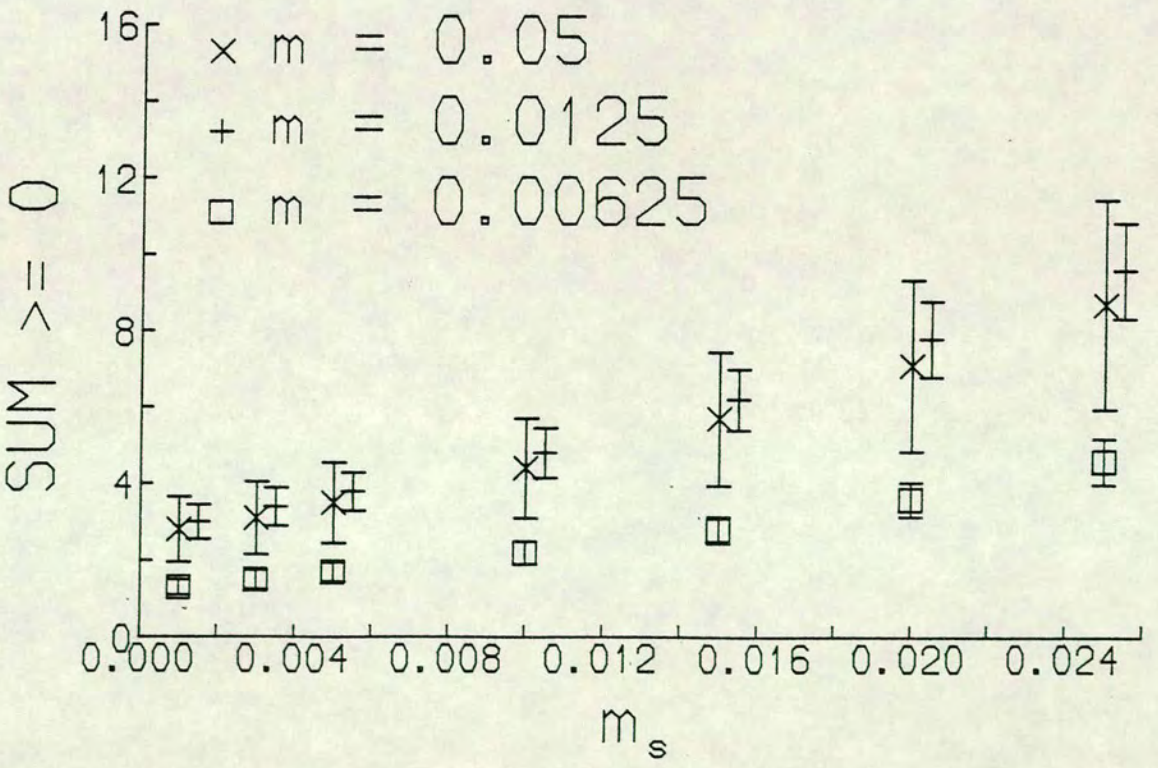


Fig. 4.7b As Fig. 4.7a for lowest 20 eigenvalues only.



$$\langle \bar{\psi}\psi \rangle_{new} = \frac{\int \mathcal{D}U \left[\mathrm{tr} M^{-1} \det M \right] e^{-S_G}}{\int \mathcal{D}U \left(\mathrm{tr} M^{-1} \right)^{-1} \left[\mathrm{tr} M^{-1} \det M \right] e^{-S_G}} \equiv \left\langle \left(\mathrm{tr} M^{-1} \right)^{-1} \right\rangle^{-1}_{new}, \tag{4.41}$$

where this new expectation value involves the weight $\mathrm{tr}M^{-1}\det M \exp(-S_G)$. We calculate $\mathrm{tr}M^{-1}$ as the sum of the inverses of the eigenvalues of M, obtained using the Lanczos algorithm. $<\bar{\psi}\psi>$ obtained from this new simulation at the highest and lowest masses, m = 0.05 and m = 0.00625, is plotted in Fig. 4.8. As each sweep now takes 4 hours we have not done as many sweeps as before so the average of $<\bar{\psi}\psi>_{new}$ over the last 8 sweeps at each mass, given in Table 4.2, is not as accurate as $<\bar{\psi}\psi>$ in Table 4.1.

Table 4.2

$<\bar{\psi}\psi>$ for $\mu = 0.1$ with new weight.

m	$<\bar{\psi}\psi>_{new}$
.05	.237 ± .028
.00625	.034 ± .011

Looking at the eigenvalue distributions of the last 4 configurations superimposed, Figs. 4.9a,b, we see that they are now closer to the real axis - perhaps confirming our suspicions about the usual Monte Carlo weight's inadequacies - but there still appears to be a gap around the real axis at the lowest mass. This shows up in the "sum ≥ 0" plot (calculated from the last 8 configurations), Fig. 4.10, as before. Hence we conclude that the Monte Carlo methods are operating well enough to signal the phase transition.

We also performed a simulation, with the usual Monte Carlo weight, at $\mu = 0$. The history of $<\bar{\psi}\psi>$ for this is shown in Fig. 4.11: we started from the same quenched configuration as for $\mu = 0.1$, did 40 sweeps (1-40) at m = 0.05, 30 (41-70) at 0.0125 and 30 (71-100) at 0.00625. Averaging $<\bar{\psi}\psi>$ over the last 20 sweeps at each mass yields the values in Table 4.3.

Fig. 4.8 $\langle\bar{\psi}\psi\rangle$ history of simulation with new weight at $\mu = 0.1$.

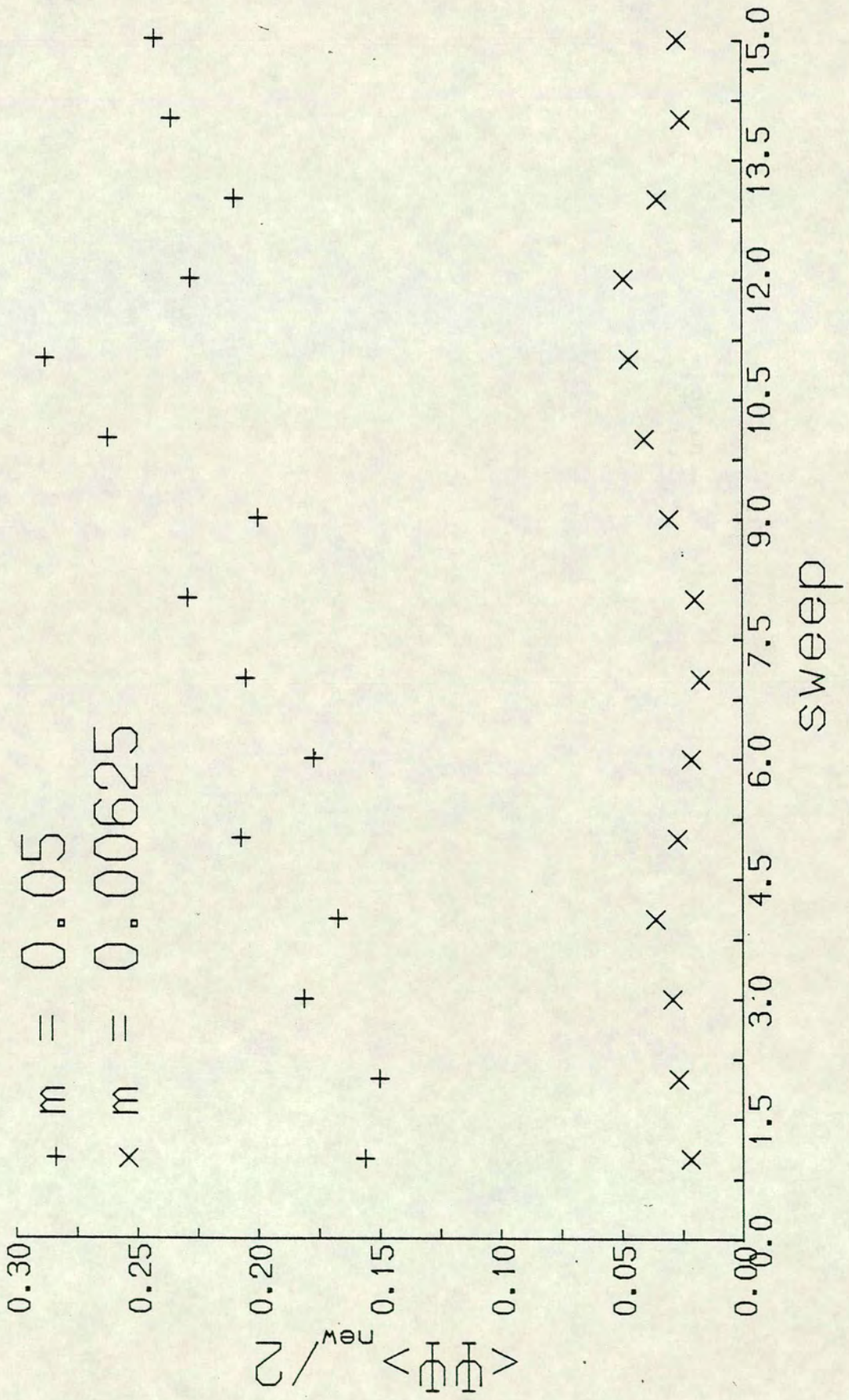


Fig. 4.9a Eigenvalue distribution with new weight (4 configurations superimposed

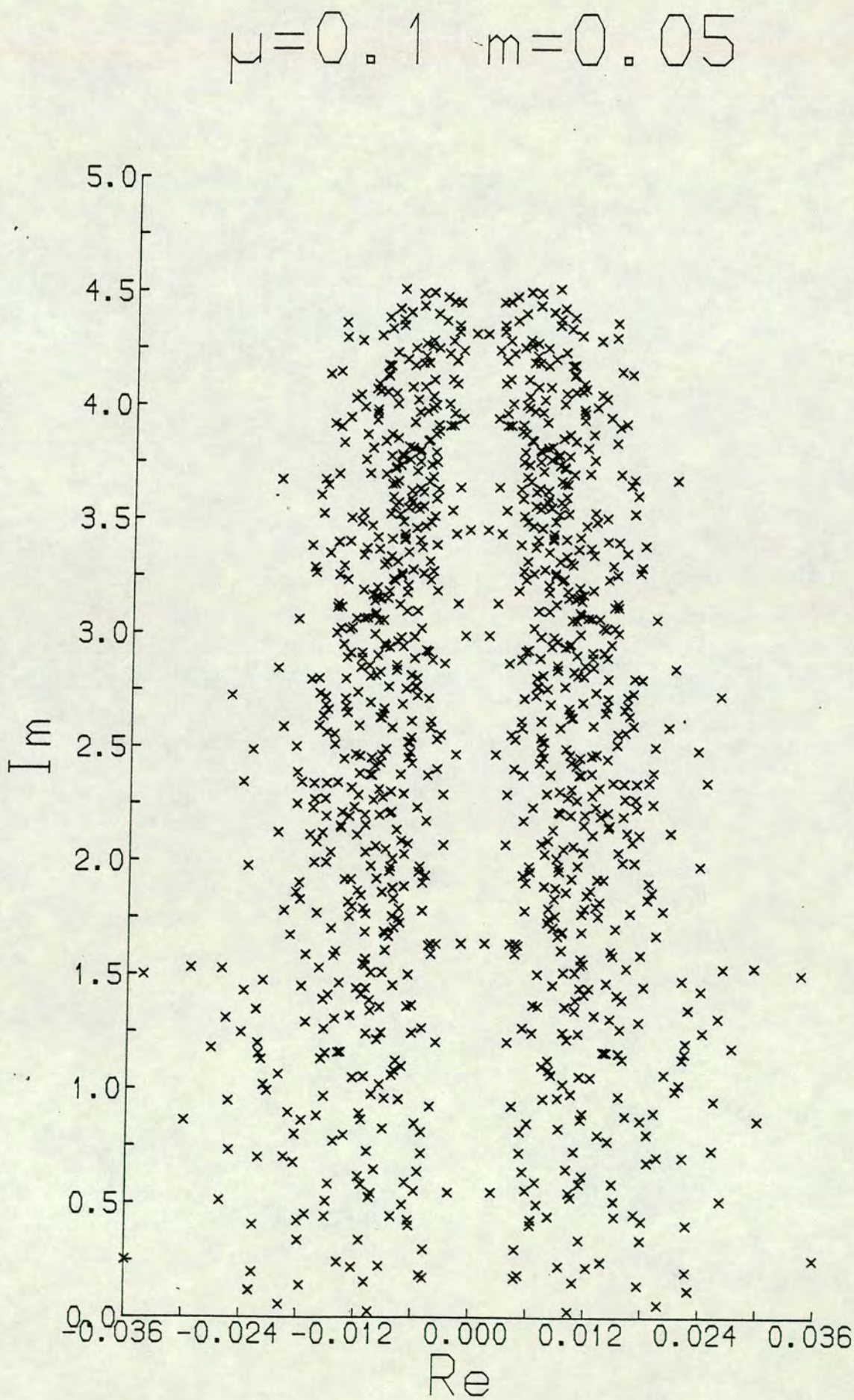


Fig. 4.9b Eigenvalue distribution with new weight (4 configurations superimpose

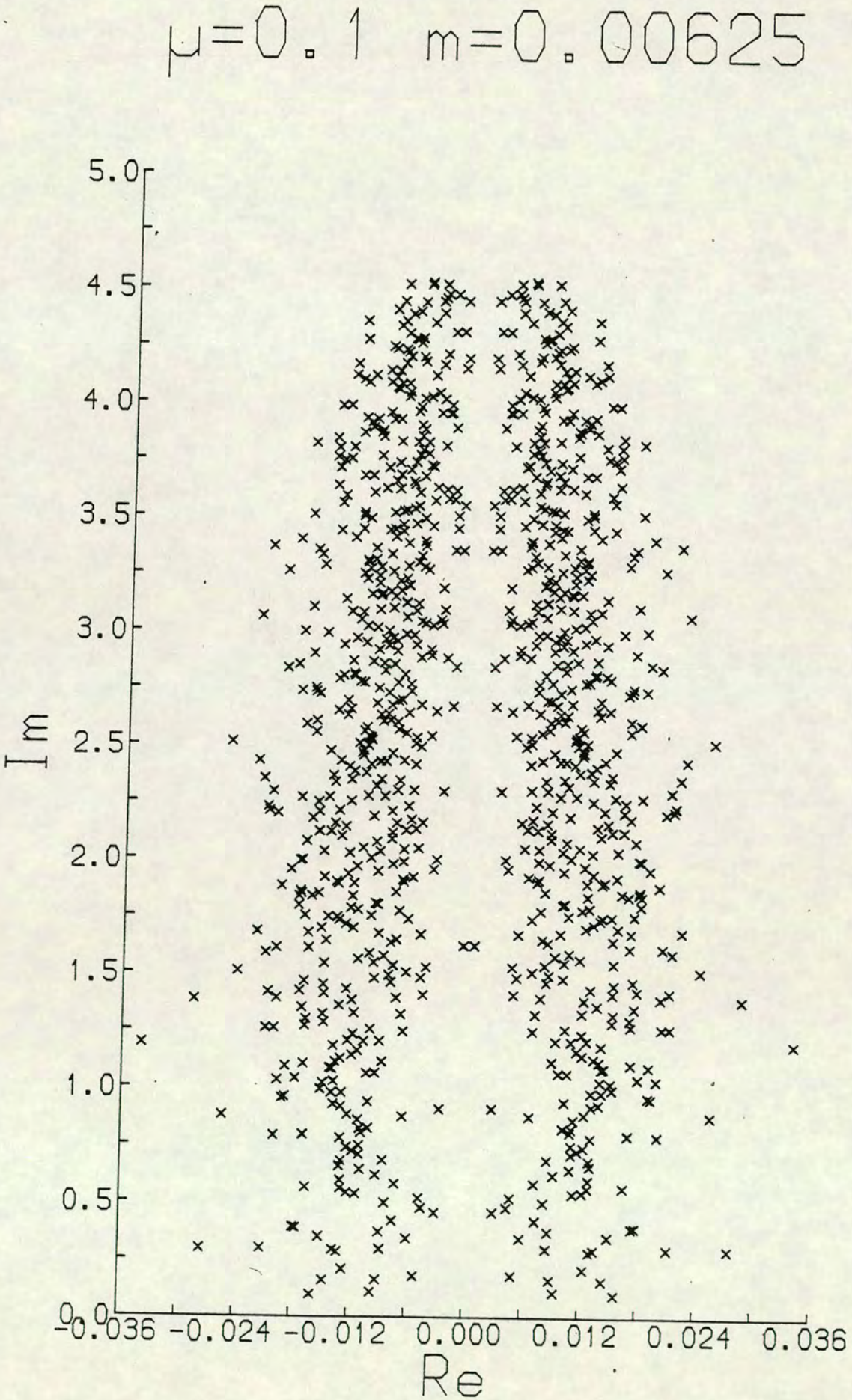


Fig. 4.10 Sum of lowest 20 eigenvalues, for each m (averaged over 8 configurations) with new weight, with real part x such that $x + m_s \geq 0$.

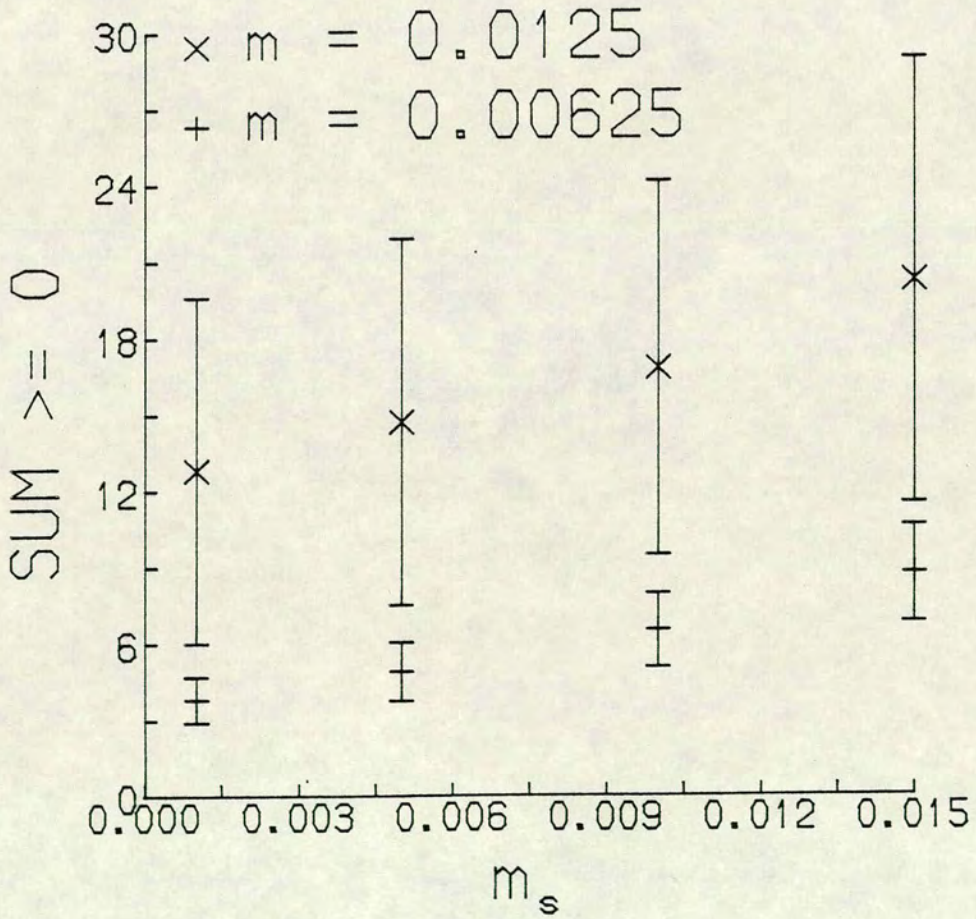


Fig. 4.11 $\langle\bar{\psi}\psi\rangle$ history of simulation at $\mu = 0$.

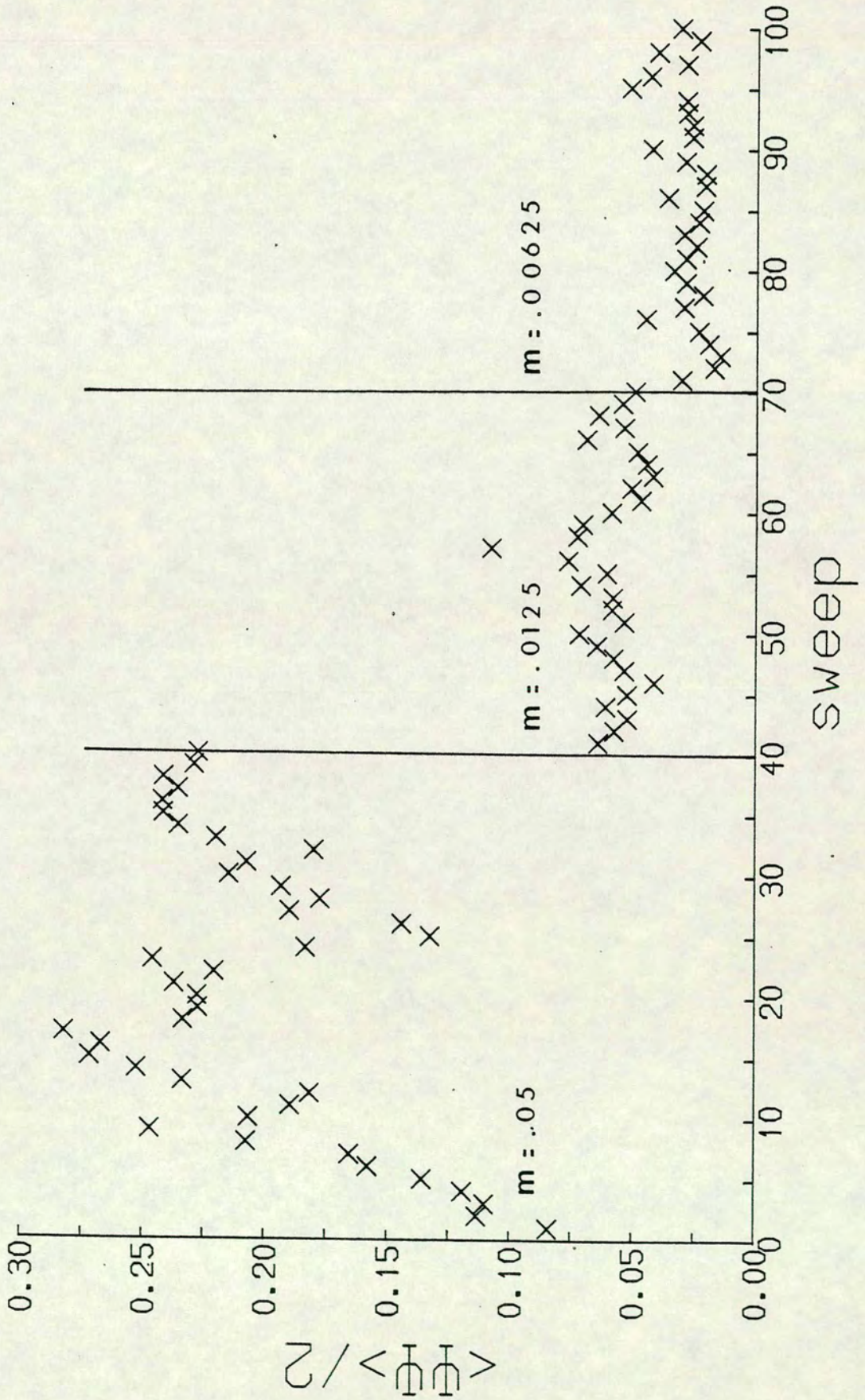


Table 4.3

$<\bar{\psi}\psi>$ for $\mu = 0$.

m	$<\bar{\psi}\psi>$
.05	.210 ± .033
.0125	.061 ± .015
.00625	.031 ± .008

As expected, these values are larger than those for non-zero μ (Table 4.1); though they also appear to extrapolate to zero. The eigenvalues are all pure imaginary since $\mu = 0$. The lowest 20 of them for the last 4 configurations, at the highest and lowest masses, are superimposed in Figs. 4.12a,b. We notice that there is a larger gap around the real axis for the lowest mass - but the scale of the imaginary axis has been expanded by a factor of about 5 so that the gaps are actually the same within error bars. This supports our conclusion that the eigenvalues move away from the real axis as the fermion mass decreases due to the finite density.

However, we cannot rule out the possibility that what we are seeing is due to finite-size effects (which we know to be large on a 4^4 lattice for free fermions - Chap. 2.1.2) - fermions with lower mass propagate further - without performing a simulation on a larger lattice which would require a larger (i.e. super-) computer.

4.3.2. Fixed m; varying μ

We now turn to the alternative regime in which to investigate the chiral symmetry restoration transition at finite density: fixed m; varying μ. In a recent preprint Dagotto, Moreo and Wolff, 1986, calculate the behaviour of $<\bar{\psi}\psi>$ in the strong coupling limit of SU(N) at finite chemical potential using a dimer approach and mean field techniques; they predict a first order phase transition for $N \geqslant 3$ and find a continuous transition for $N = 2$. We shall try to verify the latter using Lanczos dynamical fermions. We simulate on a 4^4 lattice with $n_f = 4$, $m = 0.2$ and μ varying between 0 and 1 in steps of 0.1 (as do Dagotto, Moreo and Wolff), and choose $\beta = 0.5$ to achieve strong coupling.

The history of $<\bar{\psi}\psi>$ and the plaquette (4.38) as μ is varied between 0.1 and 1 is shown in Fig. 4.13. ($<\bar{\psi}\psi>$ for $\mu = 0$ was calculated separately.) We

Fig. 4.12a Eigenvalue distribution (4 configurations superimposed).

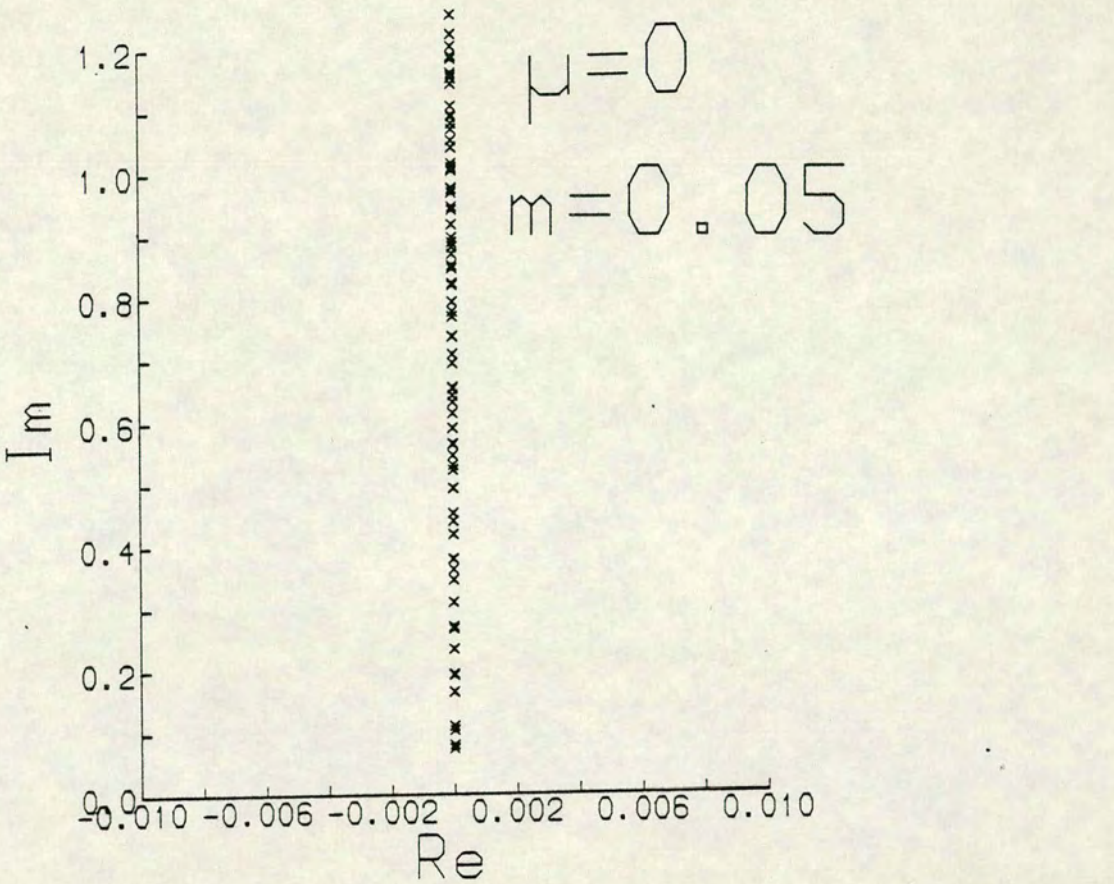


Fig. 4.12b Eigenvalue distribution (4 configurations superimposed).

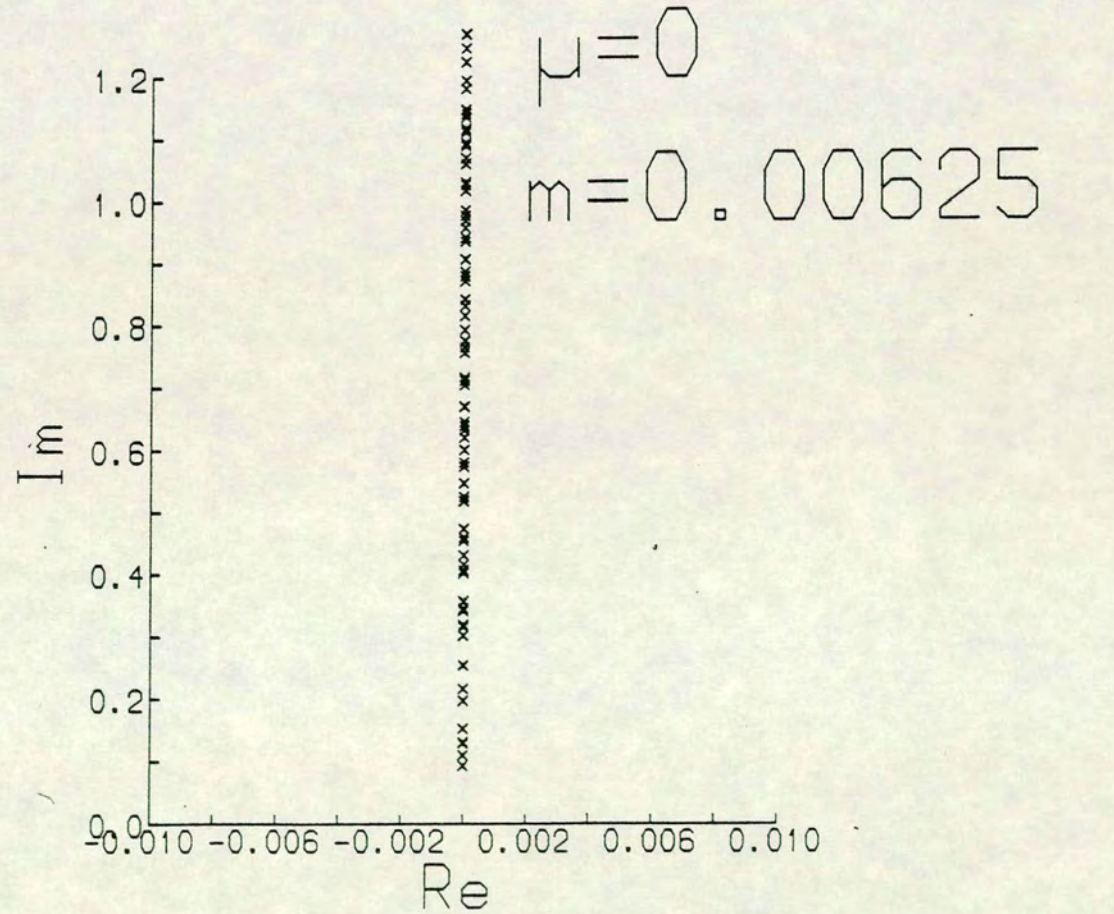
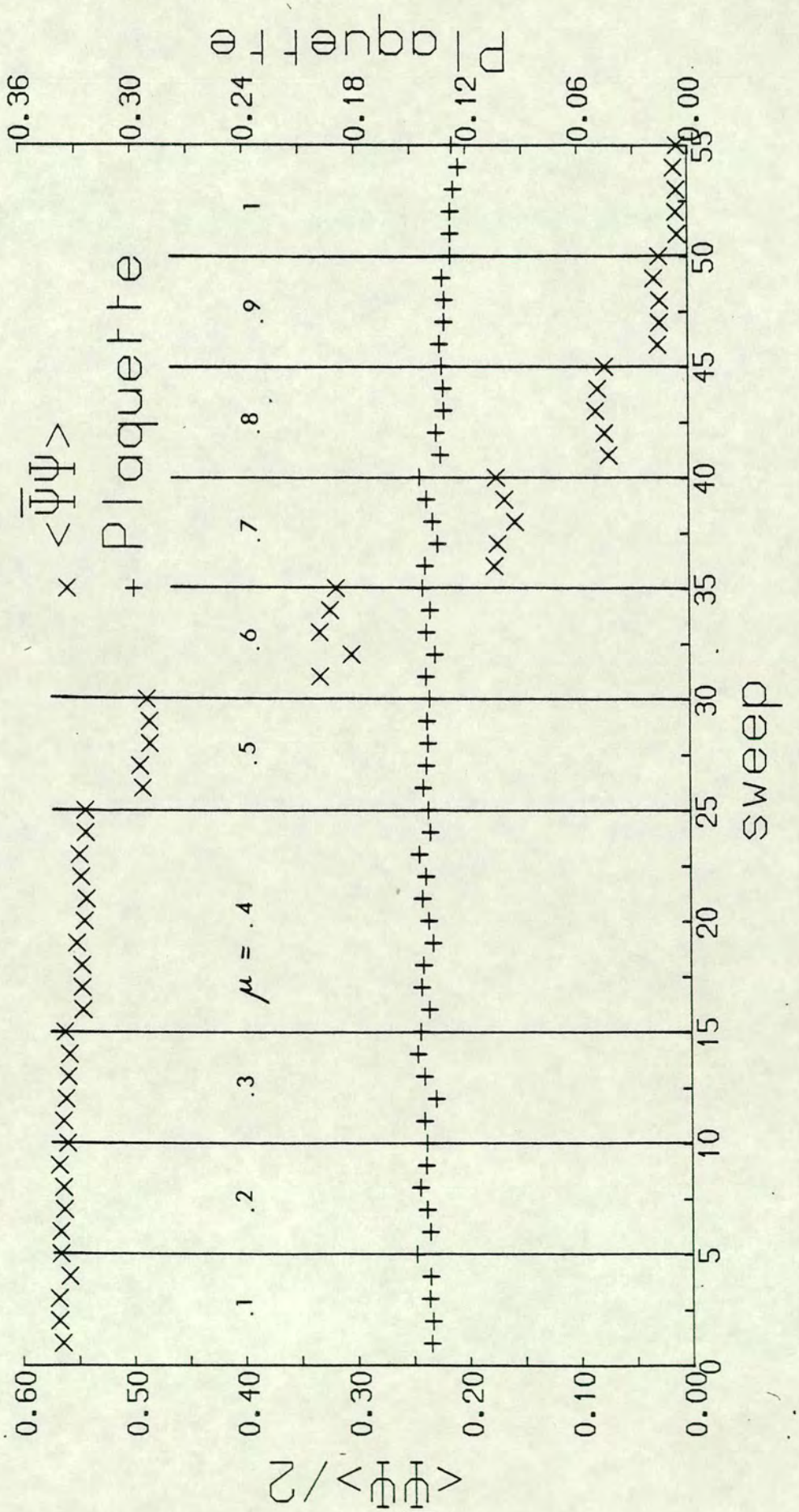


Fig. 4.13 $\langle\bar{\psi}\psi\rangle$ and plaquette history of simulation for $\beta = 0.5$ at $m = 0.2$.



performed 5 sweeps at each μ value (except for μ = 0.4 where we did 10 to ensure equilibrium had been attained) and averaged $<\bar{\psi}\psi>$ over the last 4 sweeps to obtain Fig. 4.14 (the error bars are smaller than the crosses), in which the line obtained by Dagotto, Moreo and Wolff, 1986, is also shown. We see very good agreement, particularly near the phase transition which occurs at $\mu_c = 0.6 \pm 0.2$. The discrepancy at small and zero μ is probably due to the fact that we are at finite coupling β = 0.5, whereas Dagotto, Moreo and Wolff are in the strong coupling limit β = 0.

We plot the eigenvalue distributions (of the last 4 configurations superimposed) at μ = 0.3, 0.4, ..., 1 in Figs. 4.15a,b,...,h respectively. We find the behaviour discussed earlier, and shown schematically in Fig. 4.3, as expected. (Note that in Figs. 4.3 and 4.4 λ with $Re\lambda > 0$ is plotted, whereas in Figs. 4.15a,b,...,h λ with $Im\lambda > 0$ is plotted.)

Finally, we calculate the "sum $\geqslant$ 0" which was defined in the last section as the sum in (4.39) for eigenvalues whose real part x is such that $x + m_s \geqslant 0$. Now, of course, the width of the eigenvalue strip is varying (as μ varies) so the sum will depend on both x and y. However, the width is changing dramatically while the length remains nearly constant so the dependence is mainly on x and we should find that "sum $\geqslant$ 0" decreases as μ increases (widening the strip and increasing x). That this is indeed the case is shown in Fig. 4.16.

4.3.3. Concluding remarks

We have performed simulations of SU(2) at finite density with dynamical fermions using the Lanczos algorithm in the two regimes: fixed μ; varying m, and fixed m; varying μ. In the former we find that, for a small chemical potential μ = 0.1, $<\bar{\psi}\psi>$ is less than its μ = 0 value and the eigenvalues of the fermion matrix move away from the real axis as the fermion mass is reduced - presumably because the fermion mass is moving inside the eigenvalue strip - this is probably the signal of a (chiral symmetry restoration and/or deconfinement) phase transition. In the latter we find that, at strong coupling, chiral symmetry is restored in a continuous phase transition, around μ_c = 0.6, in agreement with the strong coupling limit calculation of Dagotto, Moreo and Wolff, 1986.

Fig. 4.14 $\langle\bar{\psi}\psi\rangle$ against μ for $\beta = 0.5$ at $m = 0.2$, with line from Dagotto, Moreo and Wolff, 1986.

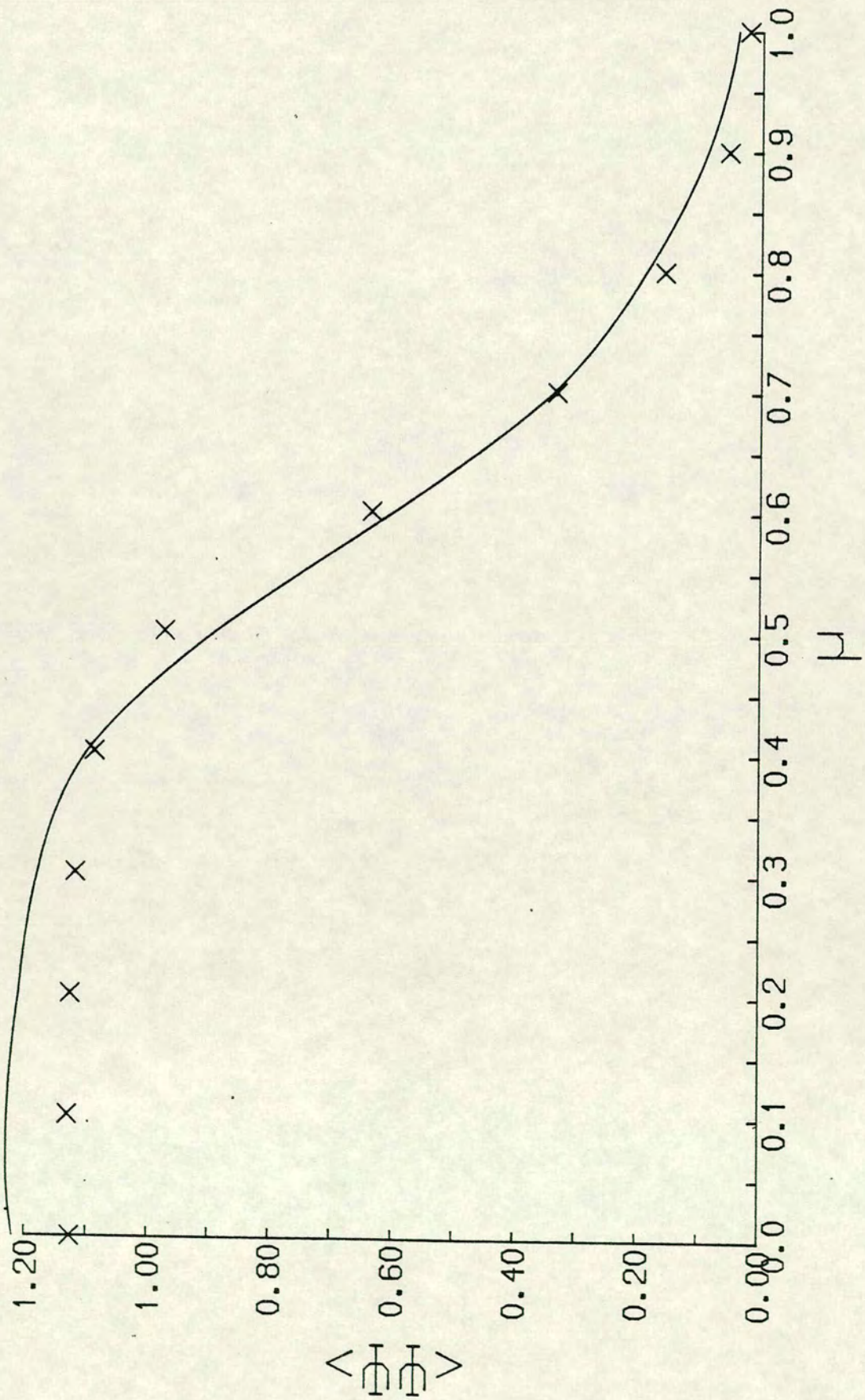


Fig. 4.15 Eigenvalue distributions for β = 0.5 at m = 0.2 for various μ.

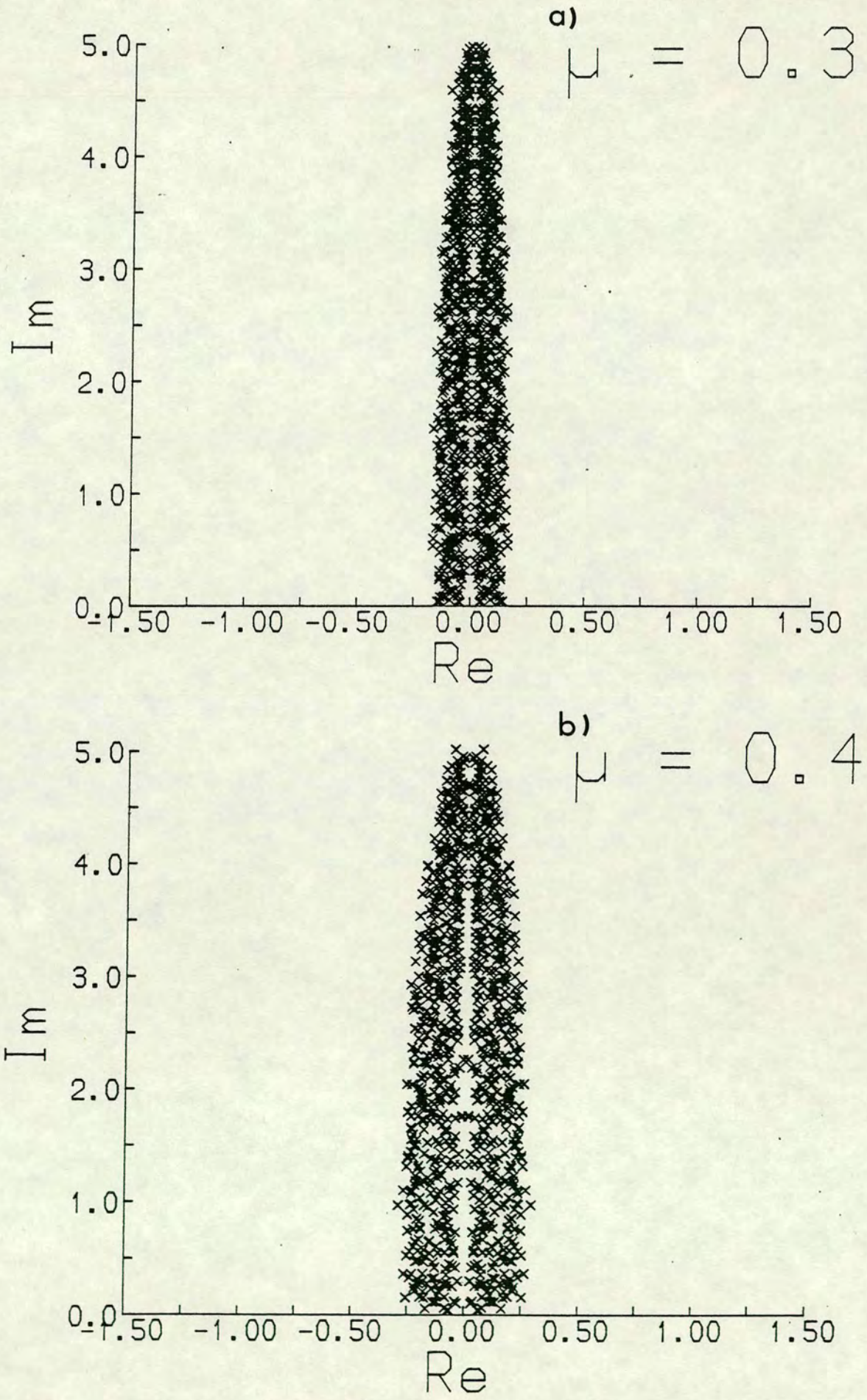
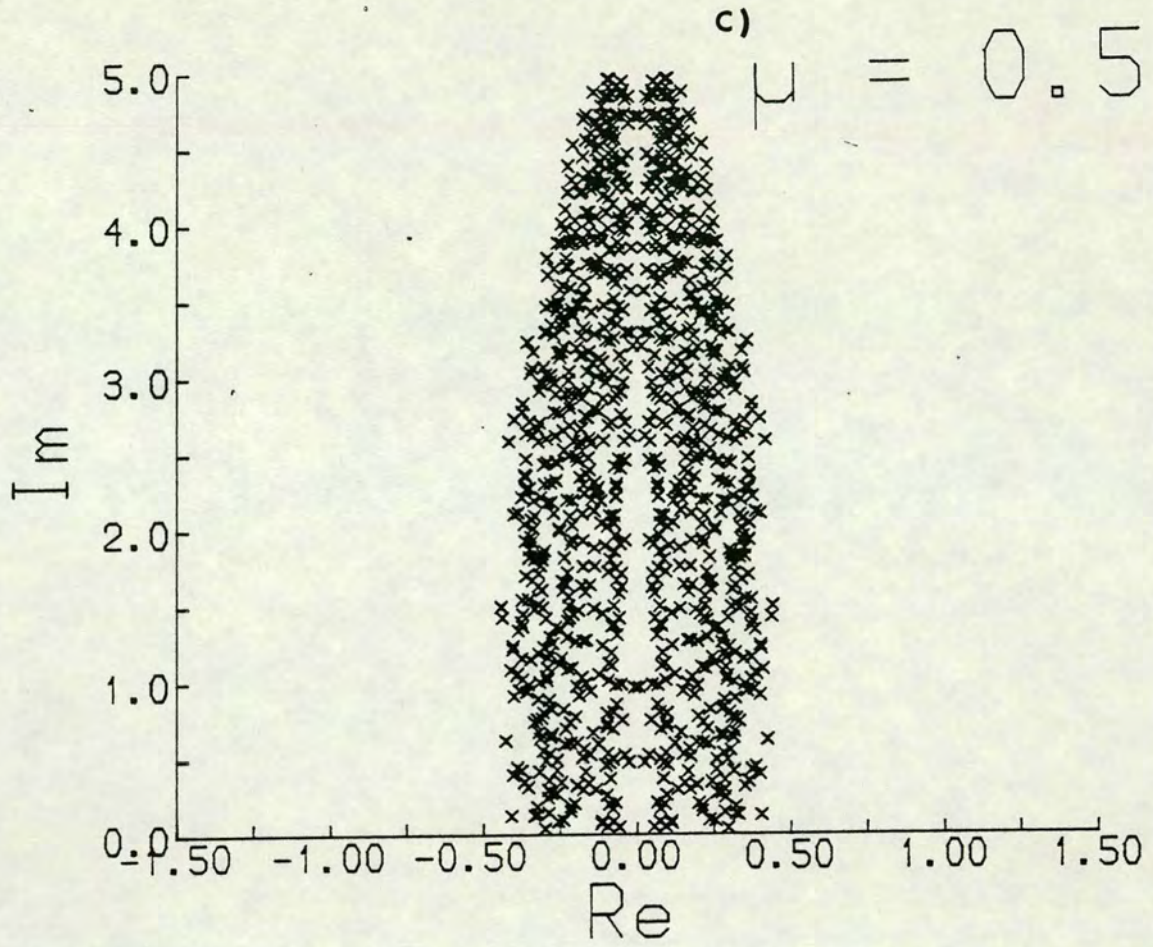


Fig. 4.15 continued



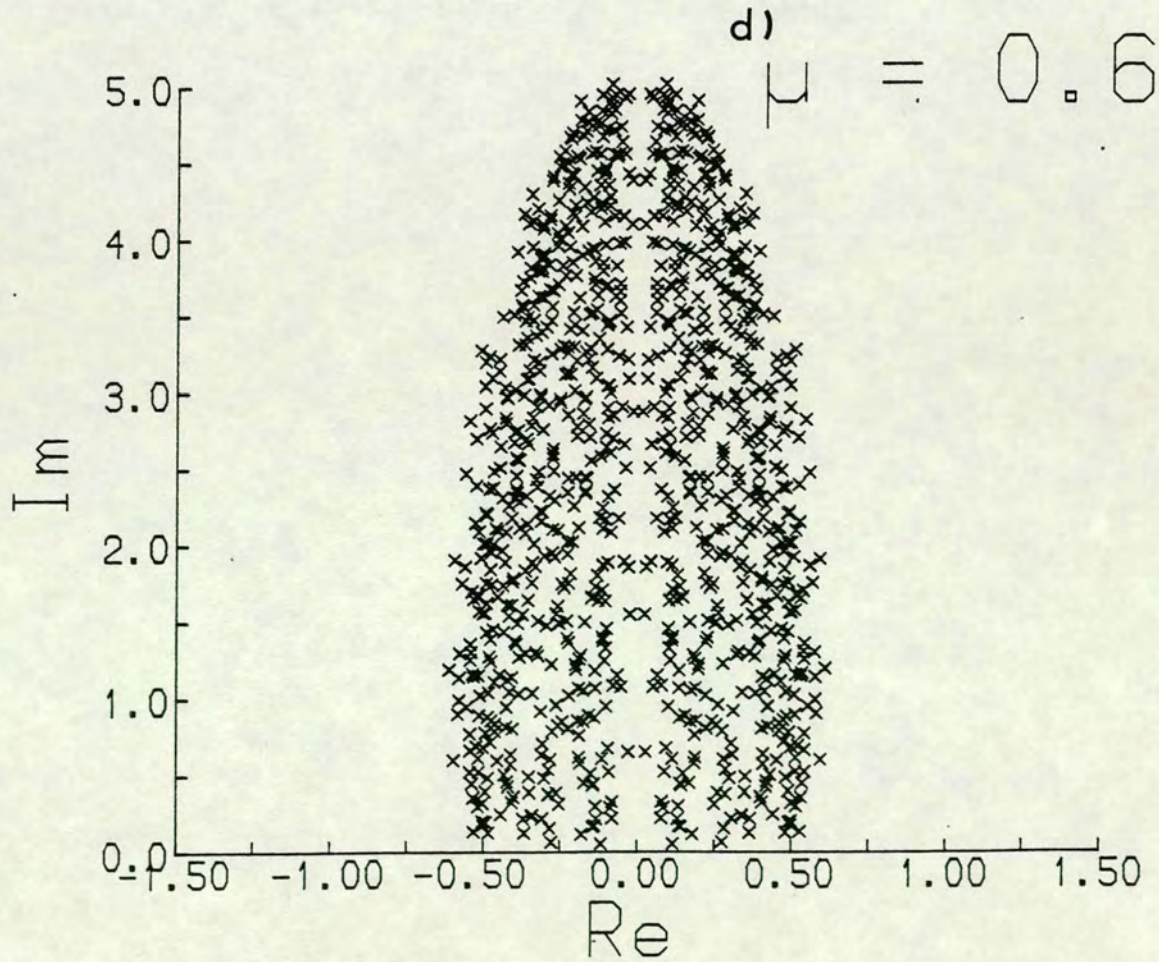
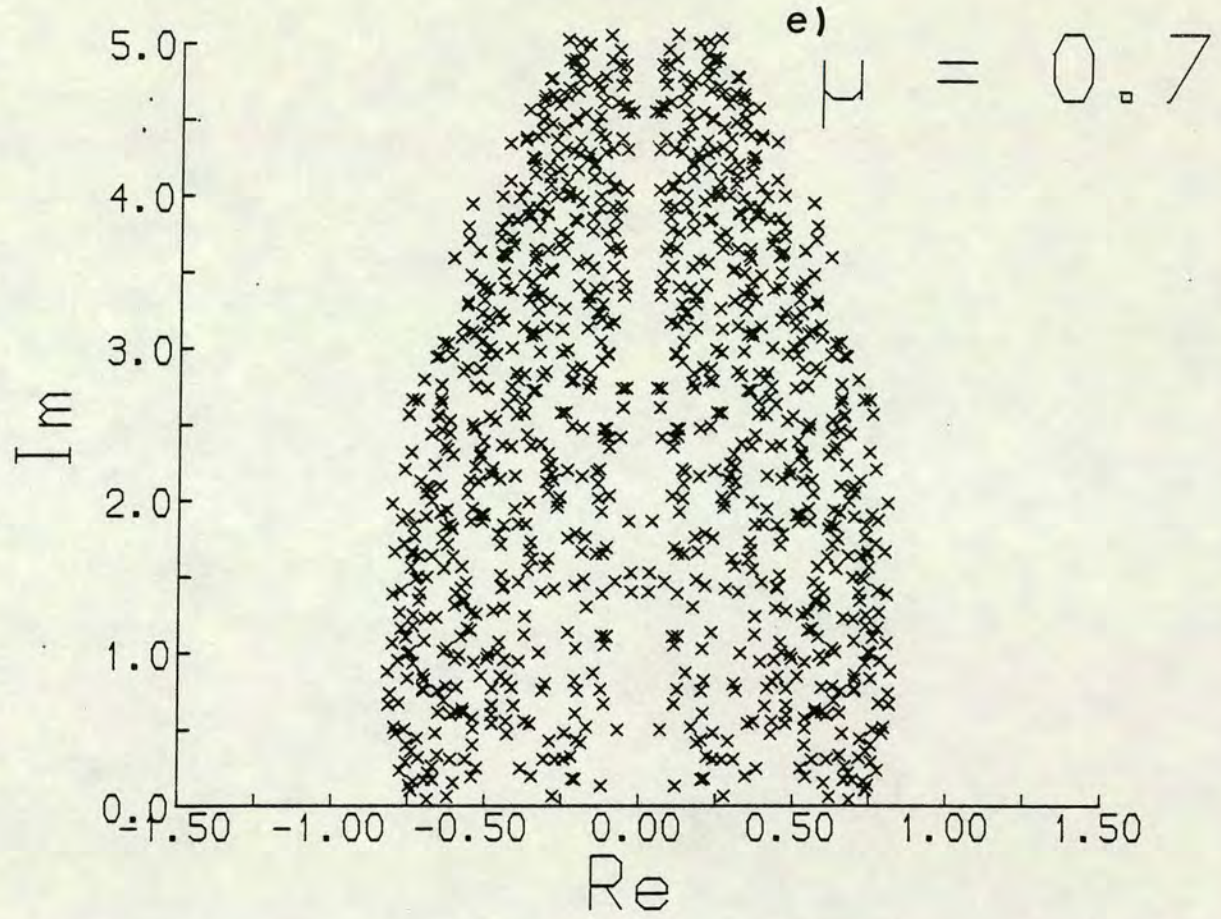


Fig. 4.15 continued



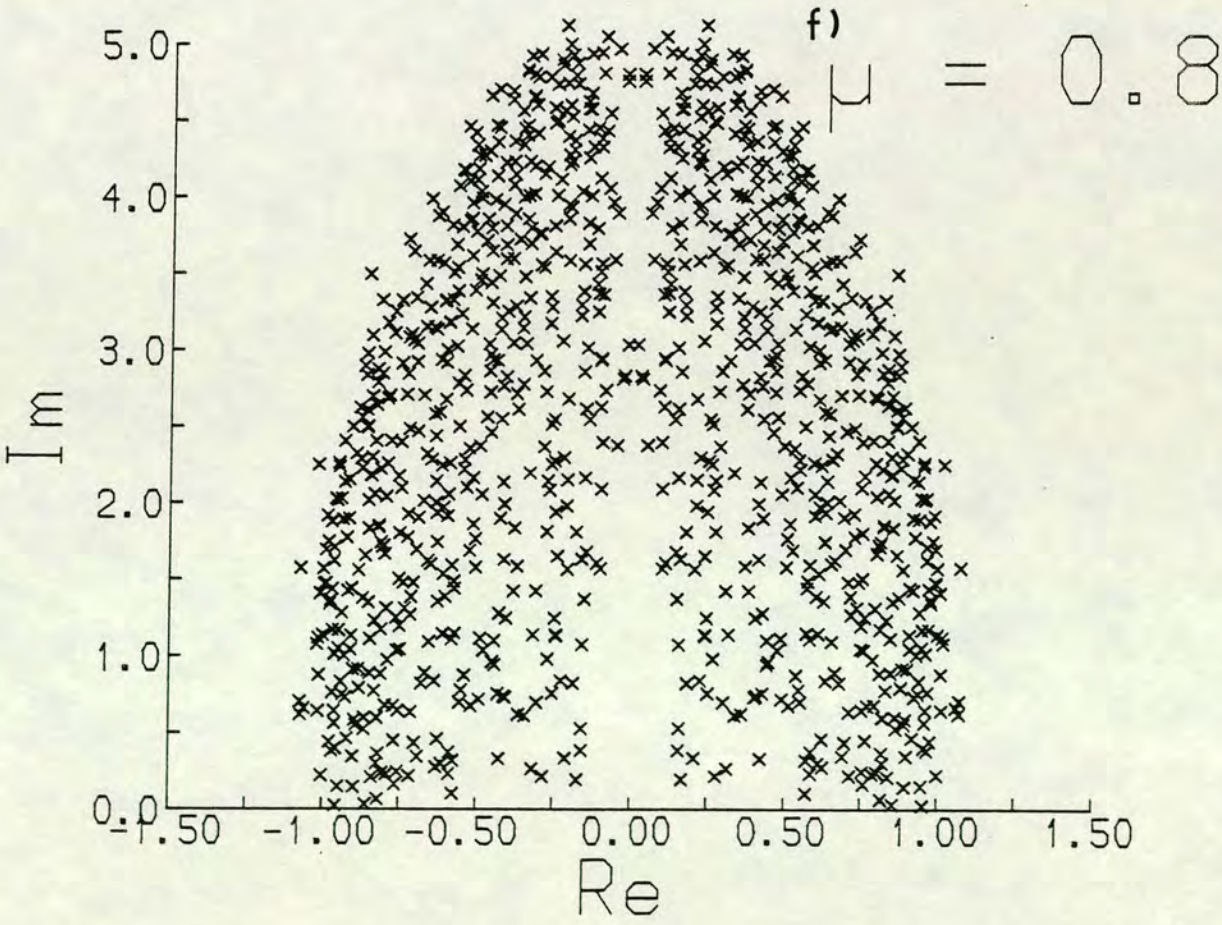
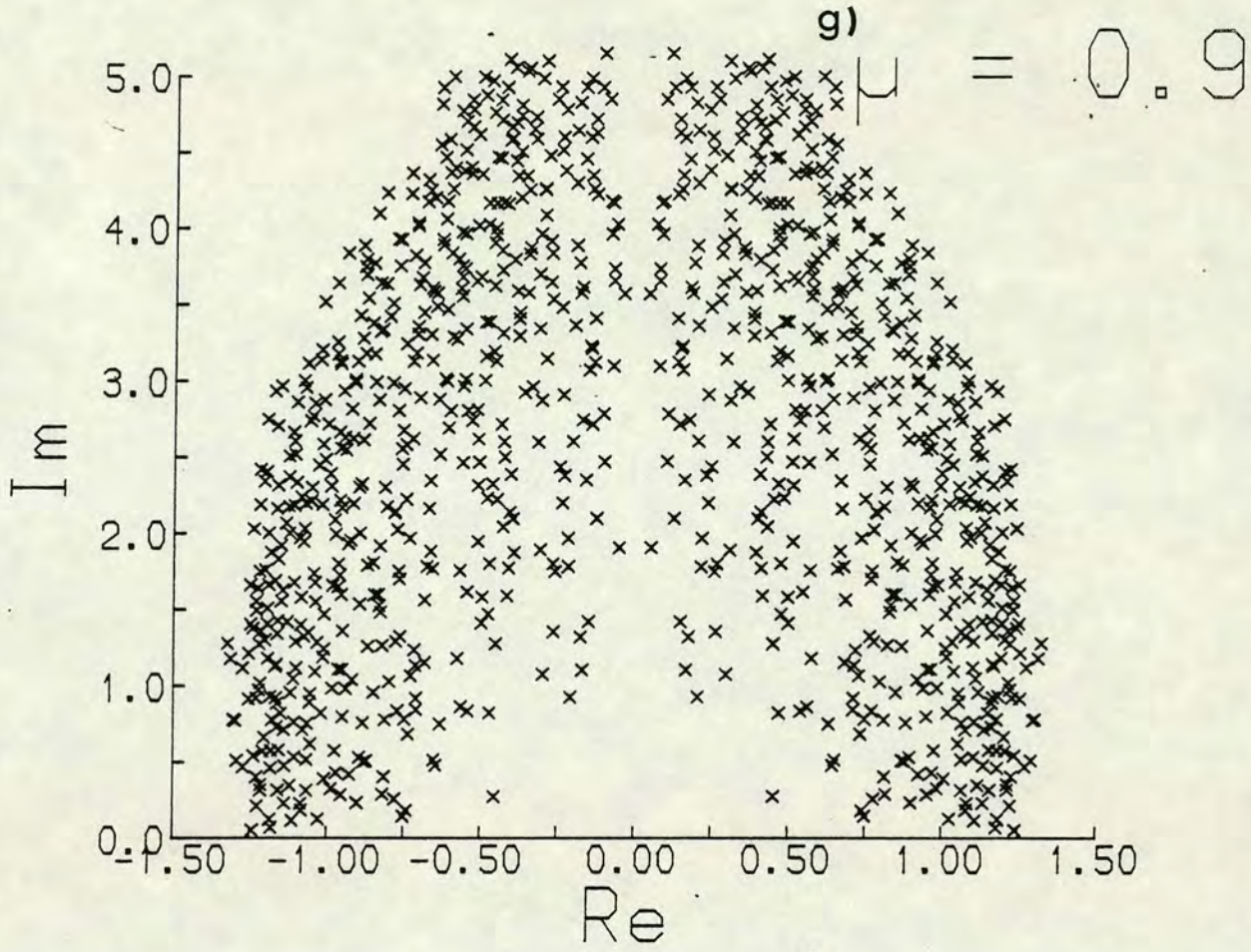


Fig. 4.15 continued



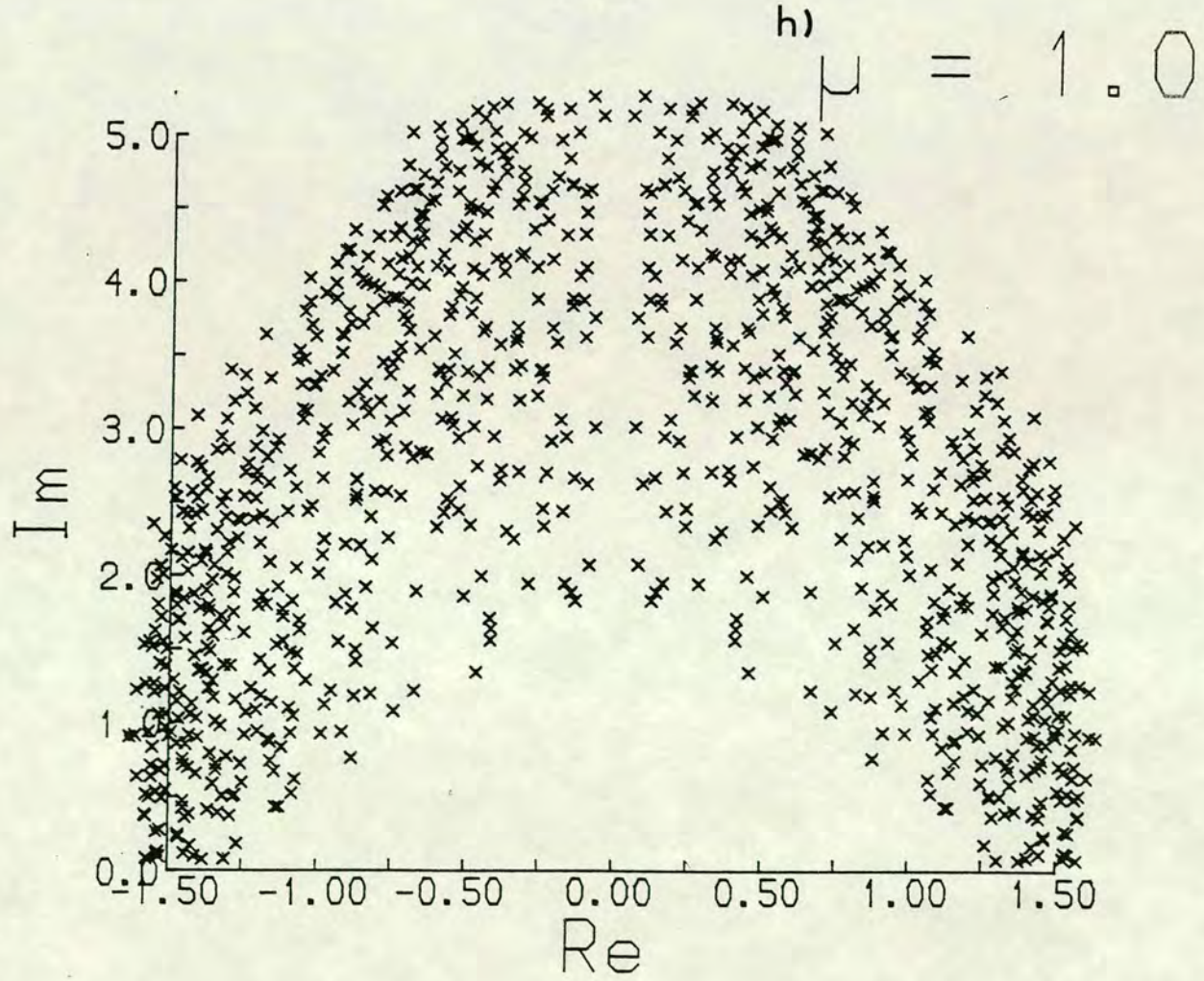
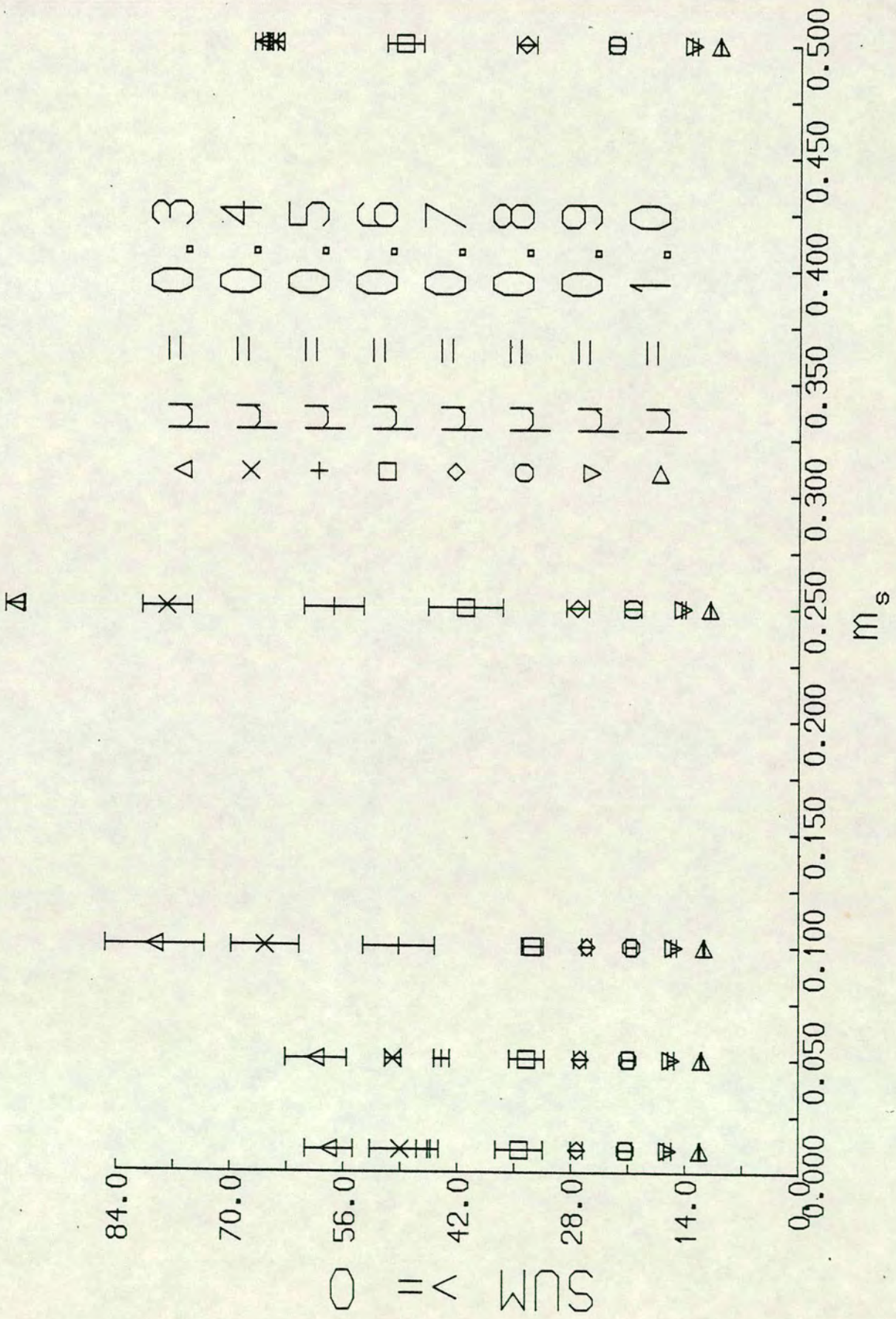


Fig. 4.16 Sum of lowest 20 eigenvalues, for each μ (averaged over 4 configurations) for β = 0.5 at m = 0.2, with real part x such that $x + m_s \geq 0$.



Chapter 5
A general FORTRAN to C translator

At Edinburgh University there are two ICL Distributed Array Processors (DAPs), which are used to perform a variety of numerical simulations (Pawley and Thomas, 1982; Bowler, 1983; Bowler and Pawley, 1984; Wallace, 1984), including the simulation of the Schwinger model described in Chap. 3.3. The DAP, which is more fully described in Appendix I, is a Single Instruction stream, Multiple Data stream (SIMD) computer comprising a 64x64 square array of bit-serial processing elements (PEs) each with 4Kbits of local memory and connections to the four nearest neighbours. It forms a (2Mbyte) memory module of the host ICL 2900 series mainframe computer. Although each PE only deals with one bit of its own store at a time, all 4096 of them perform the same operation in parallel, that is, simultaneously; this yields a very powerful computer. As described in Appendix I.II, programs consist of two parts: a serial part (written in FORTRAN (77)) which executes on the host 2900 and a parallel part (in DAP FORTRAN) for the DAP, communicating via shared COMMON blocks. A great deal of such FORTRAN/DAP FORTRAN software now exists.

However, many of the next generation of array processors, in particular the GEC Rectangular Image and Data processor (GRID), are programmed in parallel extensions of C. The GRID, which is more fully described in Appendix II, is similar to the DAP in that it contains a 64x64 square array of bit-serial PEs for parallel code, but it also contains a scalar processor to deal with serial code and it is hosted by a (mini-)computer. The GRID is programmed in GRID extended C (GEC), which is described in Appendix II.II.

Eventually one would hope to devise a Common Array Target Language (CATL), that is, an intermediate machine-independent pseudo-code for SIMD processor array computers (like DAP and GRID), into which both DAP FORTRAN and GEC would be compiled. Initially, however, it is more convenient to develop some software which automatically translates DAP FORTRAN into GEC, as well as FORTRAN 77 into C of course. In this chapter we shall describe such a general FORTRAN to C translator, which effectively enables DAP FORTRAN programs to

run on the GRID. A brief description of this software is to be published (Baillie, 1986a); detailed information can be found in a "Users Manual" (Baillie, 1986b) and a "Maintainers Manual" (Baillie, 1986c).

Note that in the following: all FORTRAN is in upper case, C is in bold face and names used in the translator software itself are enclosed in quotes.

The translator consists of two parts: a prepass and a translation pass. Before going on to describe these, we give the reason for this. In C (Kernighan and Ritchie, 1978) all symbolic names must be declared before they are used, whereas in FORTRAN (DEC, 1982) some may not be (and are given implicit types). This means that halfway through the translation of a typical FORTRAN program we may come across a symbolic name X for a variable which has not been declared (but has implicit type REAL) by which time it is too late to declare it in C. The easiest way to deal with this is for the translator to consist of two "passes": a prepass which makes up lists of symbolic names, that is, routines and their associated variables and parameters, with their types (from declarations if they are declared or implicit otherwise); and a translation pass which uses these lists to declare the symbolic names before translating the statements in which they are used.

We should also, at this point, describe the lexical analyser since it is common to both the prepass and the translation pass. A FORTRAN program is made up from lines which can be up to 72 characters long and have three fields: the statement label, the continuation indicator and the statement. In C, however, there is no concept of lines - lexemes are separated by blanks, tabs, newlines or comments. Thus the lower level of the lexical analyser reads lines and combines them into statements, making line continuations transparent to the higher level and preserving labels. It also converts FORTRAN comments directly into C comments. Then the higher level of the lexical analyser picks out the lexemes from the statement. It can do this in two ways: with or without blank spaces being significant. In standard FORTRAN, blanks are ignored, so by default the lexical analyser collects characters from the statement until it recognises what it has got. The alternative possibility - blanks being significant - is selected when the user specifies a flag ("-s") to the translator and is useful for detecting FORTRAN ambiguities like "DO 10 I = 1.5". (If blanks are ignored then this statement will set an implicitly declared REAL variable "DO10I" to "1.5", whereas if

blanks are significant, it will be spotted as a mis-typed DO-loop.) The lexemes are classified as follows:

EOF
label
STRING
DIGIT
LEX_NOT
SPECIAL
reserved character
reserved word
OTHER

where EOF (end of file) is the end of the FORTRAN program; label is a statement label; STRING is a character string constant (for example, 'Fred'); DIGIT is an integer, real, double or logical constant (for example, -123, 10.01, 6.3D5 or .TRUE.); LEX_NOT is the unary operator .NOT.; SPECIAL is a FORTRAN binary operator (.GT. .LT. .GE. .LE. .AND. .OR. .EQ. .NE. .EQV. .NEQV. .XOR.) or a binary operator particular to DAP FORTRAN (.NAND. .NOR. .LEQ. .LNEQ.); reserved characters are newline = + - : , . ( ) * /; reserved words are BLOCKDATA, DATA, CONTINUE, FUNCTION, SUBROUTINE, IMPLICIT, INTEGER, LOGICAL, REAL, DOUBLE PRECISION, COMPLEX, CHARACTER, DIMENSION, FORMAT, WRITE, PRINT, READ, CALL, DO, IF, THEN, ELSE, ELSEIF, ENDIF, END, GOTO, PROGRAM, PARAMETER, COMMON, EQUIVALENCE, STOP, ASSIGN, RETURN, SAVE, PAUSE, ENTRY, INTRINSIC, EXTERNAL, BACKSPACE, CLOSE, ENDFILE, INQUIRE, OPEN, REWIND, GEOMETRY, MATRIX and VECTOR; and OTHER is a symbolic name (routine, variable, parameter or intrinsic function). (By routine we mean FUNCTION or SUBROUTINE; variable includes array; and by parameter we mean a variable passed into a routine as one of its arguments.) The lexical analysis of FORTRAN is context sensitive, for example, given the reserved character /, the lexical analyser checks for // which may be the string concatenation operator, a blank COMMON block or two newlines in a FORMAT specification statement.

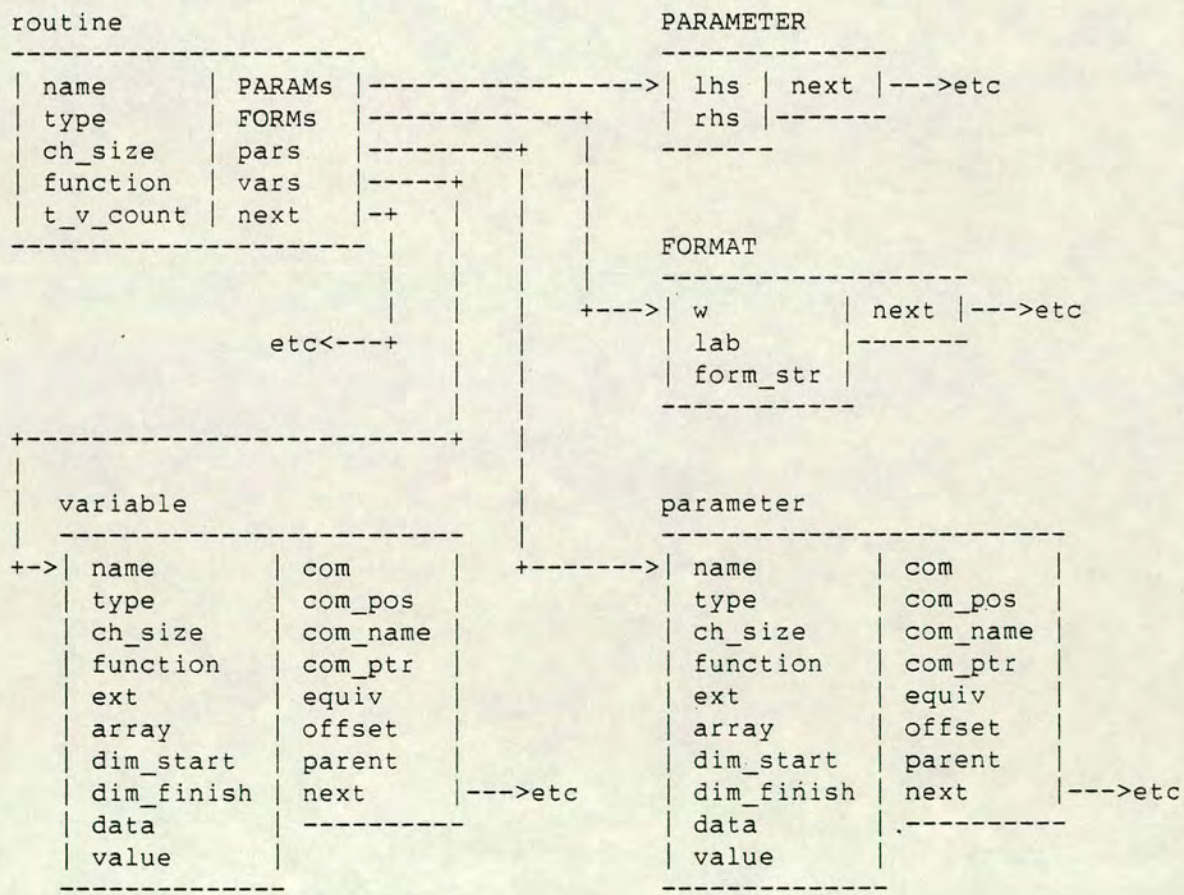
5.1. Prepass

The prepass goes through the FORTRAN program making up a list of routines and associated variables, parameters, PARAMETER definitions and FORMAT specifiers, which it stores in an intermediate file. (The translation pass will use

this information to translate the FORTRAN into C, making the necessary declarations and definitions.) In other words, the prepass deals with specification statements, that is, non-executable statements which declare, initialise, make common and equivalence, variables. We shall divide our discussion of the prepass into the following sub-sections: declaration, COMMON, EQUIVALENCE, initialisation, PARAMETER and FORMAT.

First of all, an outline of the data structures used to store the information about symbolic names is useful. There is a linked-list of routines with each routine having its own linked-list of PARAMETER definitions, one of FORMAT specifiers, one of parameters and one of variables. This structure is shown diagramatically in Fig. 5.1, where 'etc' stands for the rest of the linked-list.

Fig. 5.1 Data structures used in the translator.



Each individual data structure contains a field 'next' which points to the next one on the list, or points to null if at the end of the list. The routine data structure

also contains pointers to its PARAMETER definitions ('PARAMs'), FORMAT specifiers ('FORMs'), parameters ('pars') and variables ('vars'). It has fields for the name, type and character string size ('ch_size') - if appropriate - of the routine; a field indicating when it is a FUNCTION ('function') - as opposed to a SUBROUTINE; and a field giving the number of temporary variables required ('t_v_count') - see Sec. 2.3. The same data structure is used for both variables and parameters. It has fields for the name, type and character string size of the variable or parameter; a field indicating when it is a FUNCTION and one specifying if it is EXTERNAL as well ('ext'); some fields for when it is an array ('array', 'dim_start' and 'dim_finish'); two fields for when it is initialised ('data' and 'value'); and some for COMMON and EQUIVALENCE (which will be described later). PARAMETER is straightforward having two fields: one for its left-hand side ('lhs') and one for the right-hand side ('rhs'). FORMAT has fields for the label ('lab') and the (translated) FORMAT specification string ('form_str') as well as one to indicate whether the format is being used in a WRITE statement ('w') - see Sec. 1.6.

5.1.1. Declaration

In FORTRAN, variables and arrays may or may not be declared explicitly. Explicit declarations, for example,

```
REAL X, Y(5)                                    (5.1)
```

are easy to deal with: X is a real variable, Y is a real one-dimensional array of dimension 5. Implicit declarations are a little harder. Firstly, the prepass must keep track of what the implicit types are, as these may be changed by the IMPLICIT statement. Then, it has to identify the lexeme of class OTHER (that is, a symbolic name) as a variable, an array, a routine or an intrinsic function. Intrinsic functions are known by the translator (this, incidentally, renders the INTRINSIC statement redundant). Arrays, if they are not declared explicitly, as in (5.1), are always declared implicitly by a DIMENSION statement, for example,

```
DIMENSION Y(5)                                  (5.2)
```

Routines are indicated either by EXTERNAL statements, or as lexemes of class OTHER followed by opening brackets and not declared as arrays. (SUBROUTINEs

are also indicated by the preceding reserved word CALL, of course.) Variables are then the remaining lexemes of class OTHER. There is, however, one complication: character substrings like Y(1:3) are variables not routines. Hence to distinguish between these, the prepass follows the logic:

```
if OTHER is EXTERNAL then
        it is a function
else if it is not an array and yet is followed by '(' then
        if ':' is found amongst the arguments then
                it is a character substring i.e. variable
        else
                it is a routine.
```

The prepass also checks if the variable being (explicitly or implicitly) declared is actually a parameter passed into the routine, since parameters are stored in a separate list from variables – see Fig. 5.1 above.

FORTRAN data types are translated into the obvious C equivalents, or nearest equivalents, as listed in Table 5.1.

Table 5.1

FORTRAN data types with corresponding C translations.

FORTRAN	C
INTEGER	int
INTEGER*1, INTEGER*2	short int (WARNING)
INTEGER*3	int (WARNING)
INTEGER*4, INTEGER*I	int
INTEGER*_	long int (ERROR)
REAL	float
DOUBLE PRECISION	double
REAL*3	float (WARNING)
REAL*4, REAL*E	float
REAL*8	double
REAL*_	double (ERROR)
COMPLEX	COMPLEX (WARNING)
DOUBLE COMPLEX	COMPLEX (WARNING)
CHARACTER ...	char ...[1]
CHARACTER*N, CHARACTER*(N) ...	char ...[N]
CHARACTER*0, CHARACTER*(*) ...	char *...
CHARACTER*_ ...	char *... (ERROR)
LOGICAL	int

where '_' denotes anything else, '...' stands for a variable name list, and (WARNING) or (ERROR) signify that a warning or error is given respectively.

Note that COMPLEX is left simply as COMPLEX, since C has no complex type. DAP FORTRAN variables have the same types as the FORTRAN ones but they have different modes (Appendix I.II.I), whereas GRID extended C variables have different types form the C ones (Appendix II.II.I) - though the difference is simply the parallel extension "_array". Therefore DAP FORTRAN variables, that is, variables of mode matrix or vector, have their types translated in the same way as FORTRAN ones except that **_array** is appended to **int**, **short**, **long**, **float**, **double** and **char**; and LOGICAL becomes **bool_array**.

5.1.2. COMMON

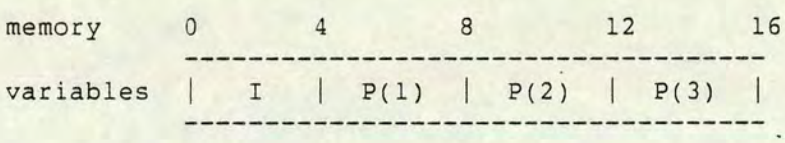
The COMMON statement defines a contiguous area (block) of storage identified by a symbolic name, in which variables and arrays are stored in a certain order. This block is accessible to any routine which refers to it explicitly. In C global variables and arrays are declared at the beginning of the program (outside the functions) and referred to as **extern** in functions that wish to use them. So to mimic a COMMON block in C we declare an **extern** (one-dimensional) array of the correct size, with the name of the COMMON block, and then declare the variables and arrays in the COMMON block as pointers into this array (counting their lengths in bytes to obtain the positions). This is fine for arrays which are essentially pointers anyway, but for variables it implies that they must always be preceded by the operator * (or have [0] appended) so that their value is taken, that is, a FORTRAN variable A which is COMMON must be written *a in C. For example (see Fig. 5.2 also)

```
INTEGER I,P(3)
COMMON /COM/ I,P
I = 0
P(1) = 1
END
```

```
char _com[16];

main()
{
        int (*i) = (int(*))(&_com[0]);
        int (*p) = (int(*))(&_com[4]);
        *i = 0;
        p[1-1] = 1;
}
```

Fig. 5.2 The COMMON block COM.



There is a linked-list of COMMON block data structures, each one containing fields for the name and length as well as a field to specify whether the COMMON is DAP FORTRAN mode matrix or vector. Note that if any member of the COMMON block is a matrix or a vector then they all must be. Each variable added to the COMMON block has the fields in its variable data structure - 'com', 'com_ptr', 'com_name' and 'com_pos' - set to indicate its position in the COMMON block. Finally we note that different routines may have different versions of the same COMMON block - the actual amount of storage required is the size of the longest version; for example,

Routine 1	Routine 2
INTEGER P1,P2	INTEGER P
COMMON /COM1/ P1,P2	COMMON /COM1/ P(2,3)

thus, COM1 has length 2x3x4 = 24 bytes. This means that at the end of the prepass the linked-list of COMMON blocks is searched and all shorter duplicates removed.

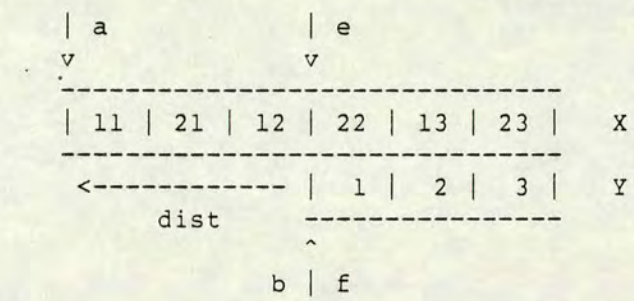
5.1.3. EQUIVALENCE

The EQUIVALENCE statement partially or totally associates two or more variables with the same storage location. Variables can be character substrings or array elements. When an element of one array is made equivalent to an element of another, equivalences are introduced between the other elements of the two arrays. This is dependent on how the arrays are stored - in FORTRAN they are stored in column-order, that is, with the left-most subscripts varying fastest. There is no concept of EQUIVALENCE in C but this can be achieved by means of pointers. Given a collection of EQUIVALENCEd variables and arrays, we compute the net (total minus overlap) amount of storage required for them and declare an array this size, with the variables and arrays declared as pointers into it, as for COMMON (see previous section). Typically we have

```
INTEGER X(2,3), Y(3)
EQUIVALENCE (X(2,2), Y)
```

resulting in the storage pattern depicted in Fig. 5.3, where X(2,2) is at the same location as Y(1).

Fig. 5.3 Showing how dist is calculated.



To calculate dist we compute e, the offset of X(2,2) from X(1,1) in units of the element size, and f, the distance of Y(1) from Y(1). In general, given an array declared as P(L,M,N), the element P(I,J,K) is offset (I−1) + (J−1)L + (K−1)ML, in units of the element size, from the first element P(1,1,1). Then the beginning of X is at a distance a = −4e bytes from the point of equivalence; similarly for Y, b = −4f bytes. (The 4's arise because an INTEGER is 4 bytes long.) Hence dist = a − b bytes. This is shown in Fig. 5.3. Effectively now having

EQUIVALENCE X, Y + dist (5.3)

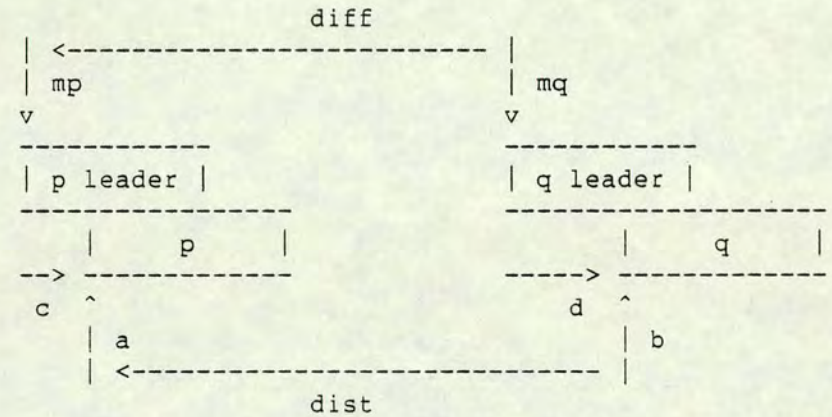
means that we can use a simple equivalencing algorithm such as the one given by Aho and Ullman, 1977 (in Sec. 10.3), to deal with a sequence of EQUIVALENCE statements which groups variables into "equivalence sets". To compute these equivalence sets we create a tree for each one. Each node of the tree is a variable data structure which has a field ('offset') containing the offset in bytes of that variable relative to the variable at the parent of this node and a field ('parent') containing a pointer to this parent node. The variable at the root of the tree is called the leader; its offset is 0 and its parent pointer is null. The position of any variable relative to the leader can be computed by following the path from the node for that variable to the leader and adding offsets along the way. Now consider equivalencing a variable p in the equivalence set tree with leader tp to a variable q in tq's tree, with dist = a − b as before. We must equivalence one tree

to the other with the correct offset, that is, we either change the offset and parent pointer of tp to make tq its parent or change tq to point at tp. To do this we follow the path from p to its leader tp summing the offsets along the way to obtain c, then the location of tp which we shall denote mp is given by a = mp + c; similarly b = mq + d, where d is the offset of q from tq. Hence the offset we require for equivalencing tp to tq is

$$diff = mp - mq = (a - c) - (b - d) = dist - c + d. \qquad (5.4)$$

A picture of this is given in Fig. 5.4.

Fig. 5.4 Showing how diff is calculated.



For efficiency we make sure the trees grow squat by equivalencing the tree with the smaller number of nodes to the other. (We could obtain maximum efficiency by path compression as well.)

EQUIVALENCE/COMMON interaction: What happens if one of the variables in an equivalence set is in a COMMON block? The entire equivalence set is put into the COMMON block at the correct place, which means we must know the extent of this set. To handle this we attach a header to each equivalence set which has two fields: low and high, giving the offsets relative to the leader of the lowest and highest locations used by any member of the equivalence set. This header is also used for the other equivalence sets to tell how long an array of char to declare in C to hold the whole set. Now when two members of different equivalence sets are equivalenced, forcing the sets to be merged, we must change the low and high of the resultant equivalence set appropriately: if we

merge tp to tq then the new fields for tq are given by

```
new lowq = min( lowq, lowp + diff)
new highq = max( highq, highp + diff)                    (5.5)
```

(if we merged tq to tp instead then would have had "- diff"). An example should illustrate how this works:

```
INTEGER X(2,3), Y(3)
COMMON /C/ X
EQUIVALENCE (X(2,2), Y)
```

```
char _c[24];
char *_e1 = &_c[0];

int (*x)[2] = (int(*)[2])(&_e1[0]);
int (*y) = (int(*))(&_e1[12]);
```

The COMMON and EQUIVALENCE are processed separately, where they occur, and then at the end of the prepass variables which are both COMMON and EQUIVALENCEd are looked for - it is then that X is noticed. The prepass transfers the COMMON attribute of X to the header of its equivalence set. (Note that if the header is already COMMON then we have an error - no two COMMONs can be EQUIVALENCEd in FORTRAN.) Then it checks that low is not before the start of the COMMON block - it is an error if it is - and if high is after the end of the COMMON block, it extends the block (giving a warning). When we come to declare the storage required (in the translation pass) for the equivalence set, we find that it is COMMON and just declare a pointer to the required position in the COMMON block. The position of the start of the equivalence set in the COMMON block for the above example is given by

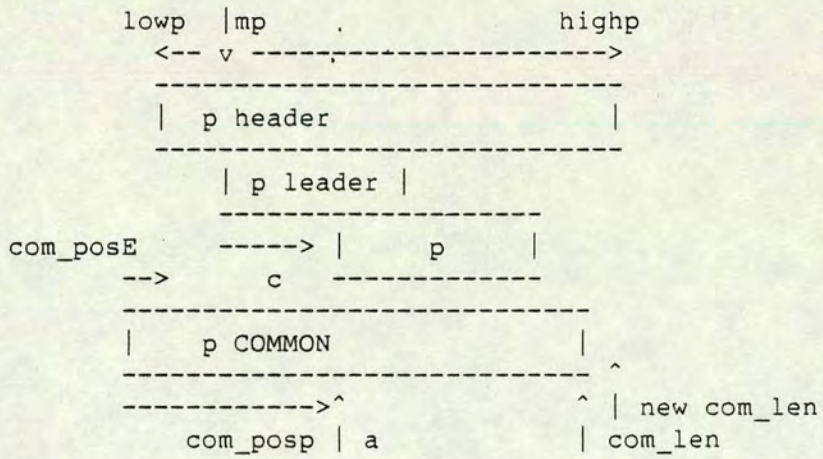
```
com_posE = com_posp - c + lowp                           (5.6)
```

and the new length of the COMMON block is

```
new com_len = com_posE + highp - lowp,                   (5.7)
```

this is shown pictorially in Fig. 5.5.

Fig. 5.5 Showing how com_posE and new com_len are calculated.



5.1.4. Initialisation

In FORTRAN, the DATA statement assigns initial values to variables *before program execution*. In C this is also the case for **static** variables which are initialised, but not for *automatic* ones - these are initialised *where they are declared* by simply adding "= value" to the declaration. Only variables and arrays which are COMMON or EQUIVALENCEd need be automatic (since they are pointers) and we can take care of these by simply initialising them separately in a special routine which is called at the beginning of the program. For example

```
INTEGER I,J
COMMON I
DATA I,J/0,1/
...
END
```

```
char _blank[4];

DATA_init()
{
        int (*i) = (int(*))(&_blank[0]);
        *i = 0;
}

main()
{
        int (*i) = (int(*))(&_blank[0]);
        static int j = 1;
        DATA_init();
        ...
}
```

There are some subtleties in the translation of declarations and initialisations of character strings, because FORTRAN reserves space for strings whereas C

does not. For variable-length character strings, declared as CHARACTER*(*) in FORTRAN 77, DATA should be used to initialise them to some string of maximum length required in order to force C to reserve enough space, for example,

```
CHARACTER*(*) S                    static char *s = "123";
DATA S /'123'/
```

otherwise they will be initialised to null strings, that is,

```
CHARACTER*(*) S                    static char *s = { "" };
```

Fixed length character strings, declared as CHARACTER*N in FORTRAN 77, which are not initialised are set equal to a string of N blanks in C to reserve this amount of space (plus one extra space for the end of string character), that is,

```
CHARACTER*5 S                      static char s[5+1] = { "     " };
```

5.1.5. PARAMETER

The PARAMETER statement assigns a symbolic name to a constant, for example,

```
PARAMETER (PI = 3.1415927)
```

In C, this can be done using a **#define** statement:

```
#define pi 3.1415927
```

Hence the prepass simply stores a linked-list of left- and right-hand sides of the PARAMETER definitions for the translation pass to output at the beginning of each routine (before the rest of the declarations, in case they make use of these PARAMETER definitions).

5.1.6. FORMAT

FORMAT statements describe the format in which data is to be input or output; each one is uniquely identified by a label. We translate them into **#define** statements as follows:

label FORMAT(fspec)	**#define** l_label Cfspec

where Cfspec is the translated FORMAT specifier, fspec. For example

1 FORMAT(3(' ',F6.3))	**#define** l_1 " %6.3f %6.3f %6.3f"

The FORMAT specifier is defined by the following regular expression grammar:

```
fspec       : primary
              primary , fspec
              primary / fspec
              / fspec

primary     : (fspec)
              i (fspec)
              fdes
              i fdes

fdes        : c
              c1 [w]
              c2 [w[.p]]

c           : c1 | c2 | X | T | H
c1          : A | I | L | O | Z
c2          : F | G | E | D
w           : field width
p           : number of decimal places
```

and therefore parsed recursively. Each FORMAT descriptor, fdes, is translated as shown in Table 5.2.

Table 5.2

FORTRAN FORMAT descriptors with corresponding C translations.

FORTRAN	C
I L	%d
O	%o
Z	%x
c1 w	%w c1
F	%f
G	%g
E D	%e
c2 w.p	%w.p c2
A	%s
A w	%.w s
X	space
T	\t
H	* (see below)
/	\n

* Hollerith strings in format specifiers are restricted to be alphanumeric with no spaces i.e. "4Habcd" is allowed but "4Ha cd" is not.
Note that there are some restrictions in what can be translated:
fdes cannot be ':', BN, BZ, S, SP, SS, TL, TR or P

If an i precedes part of the specification then the translation of this part is repeated i times (see last example). FORMAT specifiers should be given completely since C will not use them more than once like FORTRAN does.

The prepass stores the translated FORMAT specifier, Cfspec, along with its label, label, in the FORMAT data structure fields 'form_str' and 'lab', respectively. The other field, 'w', is necessary to surmount the following complication. At the end of FORMAT specifiers there is an implicit newline character since FORTRAN takes a new line after every READ and outputs a newline after every WRITE. However, in C newlines are disregarded on input (as are blanks and tabs) and so should be dropped. The field 'w' indicates whether the FORMAT specifier is used in a WRITE statement. If it is then a newline character is appended to 'form_str'. Note that if the same FORMAT specifier is used for READ and WRITE, WRITE takes priority and a newline character is appended - this means that a READ with this FORMAT specifier will fail in C. Note also that anything at the end of a line of input data which FORTRAN ignored by taking a new line will be read in by C and could lead to unexpected results!

5.2. Translation pass

Once the prepass has discovered what all the routines, variables and parameters are; sorted out the variables which are COMMON and/or EQUIVALENCEd; and dealt with PARAMETER definitions and FORMAT specifiers, the translation pass can proceed. The translation pass deals with control statements, I/O statements, routines, expressions, end intrinsic functions. These are discussed in turn in the sub-sections below and are summarised in Table 5.3.

Table 5.3

FORTRAN statements with corresponding C translations.

FORTRAN	C
ASSIGN s TO v	#define l_v l_s
BACKSPACE[(][UNIT=]u[,ERR=s][)]	if (fseek(f_u,-1L,1) == NULL) [goto l_s];
BLOCKDATA [n]	used by prepass
CALL f[([n][,n]...)]	f([n][,n]...);
CLOSE([UNIT=]u[,ERR=s])	if (fclose(f_u) == NULL) [goto l_s];
COMMON [/[cb]/]nlist[[,]/[cb]/nlist]...	used by prepass
CONTINUE	; (null statement)
DATA nlist/clist/[[,]nlist/clist/]...	used by prepass
DIMENSION a(d)[,a(d)]...	used by prepass
DO s[,] n=e1,e2[,e3]	for (n=e1;n<=e2;n+=[e3]|1) {
ELSE	} else {
ELSEIF (e) THEN	} else if (e) {
END	} (terminates program unit)
ENDFILE	not implemented
ENDIF	} (terminates block IF)
ENTRY	not implemented
EQUIVALENCE (nlist)[,(nlist)]...	used by prepass
EXTERNAL f[,f]...	used by prepass
label FORMAT(fspec)	#define l_label Cfspec
[typ] FUNCTION f[([n[,n]...])]	Ctyp f([n[,n]...]) {
GOTO s	goto l_s;
IF (e) st	if (e) Cst;
IF (e) THEN	if (e) {
IMPLICIT typ(l[,l]...)[,typ(l[,l]...)]...	used by prepass
INQUIRE	not implemented
INTRINSIC f[,f]...	used by prepass (actually ignored)
OPEN([UNIT=]u[,FILE=n][,ERR=s])	if ((f_u=fopen([n]|default,"r"|"w")) == NULL) [goto l_s];
PARAMETER (n=c[,n=c]...)	#define n c ...
PAUSE	not implemented
PRINT	not implemented
PROGRAM n	/* n */
READ([UNIT=]u,[FMT=]label[,END=s]) [nlist]	if (fscanf(f_u,l_label[,nlist]) == EOF) [goto l_s];
RETURN	return[(f)];
REWIND[(][UNIT=]u[,ERR=s][)]	if (fseek(f_u,0L,0) == NULL) [goto l_s];
SAVE nlist	used by prepass (actually ignored)
STOP [disp]	[fprintf(stderr,"%d\n"|"%s\n",disp);] exit();
SUBROUTINE f[([n[,n]...])]	f([n[,n]...]) {
WRITE([UNIT=]u,[FMT=]label) [nlist]	fprintf(f_u,l_label[,nlist]);

where

a(d)	array declarator
c	constant
clist	list of constants separated by commas
cb	common block name
default	system dependent file name
disp	integer or character constant
e	logical expression
e1,e2,e3	numeric expressions
f	subprogram name
fspec	format specifier
Cfspec	equivalent C format specifier
l	single letter, or range of letters (l-l)
n	symbolic name
nlist	list of variable names separated by commas
s	statement label
st	statement
Cst	equivalent C statement
typ	type specifier
Ctyp	equivalent C type specifier
u	logical unit specifier
v	integer variable name

and

...	indicates that the preceding item(s) can be repeated one or more times
[]	implies optionality
|	denotes or
[]|	means first thing if present, second otherwise

5.2.1. Control statements

First, GOTO is translated very easily into **goto** in C; CONTINUE is the null statement ";" and a label like "123" becomes "l_123" (since labels, like other symbolic names in C, must begin with a letter). IF, THEN, ELSE, ELSEIF and ENDIF obviously present no difficulties and DO can be translated into **for**. For example

```
      DO 10 I = 1,100                  static int i;
         IF (MOD(I,2) .EQ. 0) THEN     for (i = 1; i <= 100; i += 1) {
            ...                            if (((i) % (2)) == 0)
         ELSE                              {
               GOTO 10                        ...
         ENDIF                             }
10    CONTINUE                           else
                                         {
                                               goto l_10;
                                         }
```

```
l_10:
    ;
}
```

Note that we have not dealt with computed GOTO and arithmetic IF, as they are essentially redundant in FORTRAN 77 (and will be "deprecated" in FORTRAN 8X). STOP can also be translated easily - into **exit**. However, PAUSE can not be translated as there is simply no analogous statement in C.

5.2.2. I/O statements

We have already discussed how FORMAT specification statements are translated into **#define** statements in Sec. 1.6. Here we describe how READ and WRITE are converted into **fscanf** and **fprintf**. In FORTRAN, READ and WRITE input from and output to logical units which are connected to files in the outside world. In C, file pointers perform essentially the same function - they are declared using the special type FILE. Most logical units are assigned using the OPEN statement, however, 5 and 6 are pre-defined as the default READ and WRITE I/O units respectively. Similarly, most file pointers are set up by the function fopen, however, **stdin** and **stdout** are automatically initialised. This explains the following example translation:

```
    WRITE(6,1) R1,R2,R3          #define l_1 "%6.2f%6.2f%6.2f\n"
1   FORMAT(3F6.2)                    static FILE *f_6 = stdout;
                                     fprintf(f_6, l_1, r1, r2, r3);
```

The logical unit can be given as *, in which case the translator will choose the correct default (5 or 6). However, the FORMAT specifier may not be * because the translator cannot cope with list-directed formats. Similarly, it cannot cope with PRINT, or READ without brackets. (It would be trivial to modify the translator so that these cases could be handled - one simply checks the types of the variables in the READ, WRITE or PRINT and sets up the appropriate C format specifier.) It will cope with an END transfer-of-control specifier in READ statements.

The file manipulation statements OPEN, CLOSE, BACKSPACE and REWIND can be translated into the functions fopen, fclose and fseek as shown in Table 5.3.

However, ENDFILE and INQUIRE have no analogy in C and so cannot be translated.

5.2.3. Routines

Routines (FUNCTIONs and SUBROUTINEs) are on the whole easy to deal with, but there are some complications. SUBROUTINE calls are preceded by the reserved word CALL, this is simply dropped in C. FUNCTIONs return their result in a variable of the same name so this is also done in the C. However, in FORTRAN, one can also have a so-called "alternate return" from a routine which causes a transfer-of-control in the calling program - this most certainly does not exist in C. Similarly absent is the concept of an ENTRY statement which allows one to jump into the middle of a routine in FORTRAN. The main complication stems from the fact that routine arguments in FORTRAN are call-by-address but C's function arguments are call-by-value. This means that if a variable X which has value 1.0 is passed into a function F which sets its argument to 2.0, then in C X will still be 1.0 after the function has returned but in FORTRAN it will be 2.0. Fortunately we can coerce C into using call-by-address by means of pointers; in fact, we use call-by-value but with the addresses of the values. Then, of course, the arguments passed to a function are addresses so inside the function we must precede references to them with * (or append [0]). This is all well and good for variables and arrays whose addresses can be taken but not for constants and expressions - we cannot write &1 or &(x+0.5) to obtain their addresses. To cope with this, we must assign arguments which are constants or expressions to temporary variables and pass the addresses of those into the function. Note, however, that this is not necessary for intrinsic function arguments. For example

```
SUBROUTINE FRED(A,B,C)
A = B + C
RETURN
END

CALL FRED(X,1.0,COS(0.0))
END
```

```
fred(a,b,c)
float *a;
float *b;
float *c;
{
        (*a) = (*b) + (*c);
        return;
}

main()
{
        float t_v_1, t_v_2;
```

```
        static float x;
        t_v_1 = 1.0;
        t_v_2 = cos(0.0);
        fred(&x, &t_v_1, &t_v_2);
}
```

This is done by the recursive expression parser which is described in the next section.

5.2.4. Expressions

We shall firstly discuss the translation of FORTRAN expressions and then say something about translation specific to DAP FORTRAN' expressions. The FORTRAN infix operators // (string concatenation) and ** (exponentiation) correspond to the C functions concat and power which are effectively prefix. Thus we must know and be able to change the structure of expressions - this is done by a recursive expression parser which builds each expression into a tree and generates C, in the correct order, from it. (Since all the types of the symbolic names are known we can tell if have parallel expressions, that is, ones of mode matrix or vector in DAP FORTRAN, and deal with them appropriately as well.) The parser also takes care of the temporary variables required to ensure that routine arguments remain call-by-address rather than become call-by-value, as described in the previous section. However, the parser takes no account of the differences in precedence and associativity of operators between FORTRAN and C. As there are so few differences, it was felt not to be worthwhile introducing operator-precedence parsing (Sec. 5.3 of Aho and Ullman, 1977).

The regular expression grammar which the recursive expression parser follows is

```
expr        : opand binop expr

opand       : ( expr )
              unop opand
              DIGIT
              STRING
              variable
              parameter
              routine
              routine passed as parameter
              intrinsic function
```

```
binop       : , = + - * / ** // SPECIAL

unop        : + - LEX_NOT (

DIGIT       : integer constant
              real or double precision constant
              logical constant (.FALSE. .TRUE.)
STRING      : character string constant
SPECIAL     : .GT. .LT. .GE. .LE. .AND. .OR. .EQ. .NE.
              .EQV. .NEQV. .XOR. .NAND. .NOR. .LEQ. .LNEQ.
LEX_NOT     : .NOT.
```

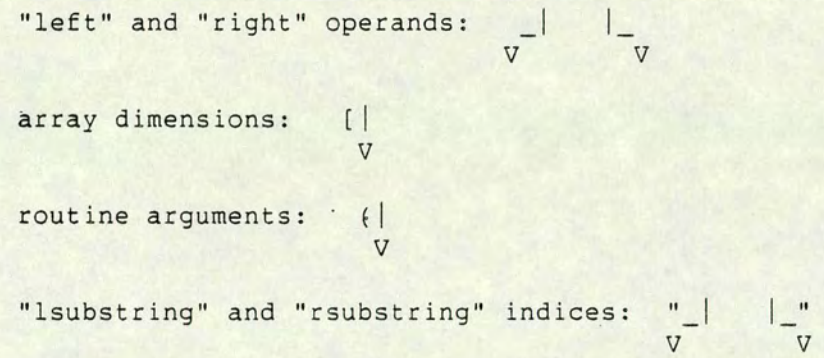
Note that '(' has been made a unary operator and ',' has been made a binary operator for convenience.

We shall outline how the recursive expression parser actually works, describing the data structure used in building the parse tree. In what follows, "parameter" refers to the dummy arguments which are used inside a routine, whereas "argument" refers to the actual arguments of a routine call which occur outside the routine.

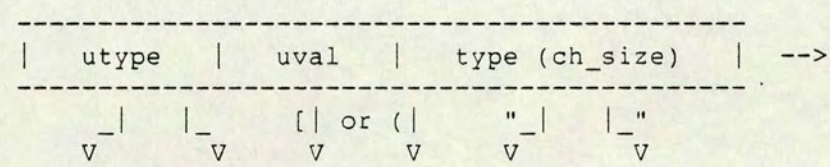
An expression consists of nodes which come in seven 'utypes' - UCONST, UOP, UVAR, UPAR, UVFN, UPFN and UINTR - with a 'uval' and a 'type' (plus 'ch_size' if the type is CHARACTER). UCONST means that the node is a DIGIT or a STRING; UOP means that it is a binary or unary operator; UVAR is a variable; UPAR is a parameter; UVFN is a routine; UPFN is a routine which is passed as a parameter; and UINTR is an intrinsic function. The reason for the distinction between a routine (UVFN) and a routine which is passed as a parameter (UPFN) is that they are declared differently in C: the former would be, say, "int ifn();", whereas the latter would be "int (*ifn)();". If the node is a unary operator then it has a pointer to its operand, if it is a binary operator then it has pointers to its 'left' and 'right' operands. If it is a (variable or parameter) array then it has a pointer to its dimensions; if it is a routine or a routine which is passed as a parameter then this pointer (called 'args') points to its arguments. If it is a character substring then it has pointers to its 'lsubstring' and 'rsubstring' indices. If it is part of a list of array dimensions or routine arguments then it has a pointer to the 'next' node on this list (null if it is the last one). If we denote these pointers as indicated in Fig. 5.6a then the node data structure can be represented as in Fig. 5.6b.

Fig. 5.6 Data structure node used by the recursive expression parser:

a) diagramatic representation of the pointers;



b) diagramatic representation of the whole thing.

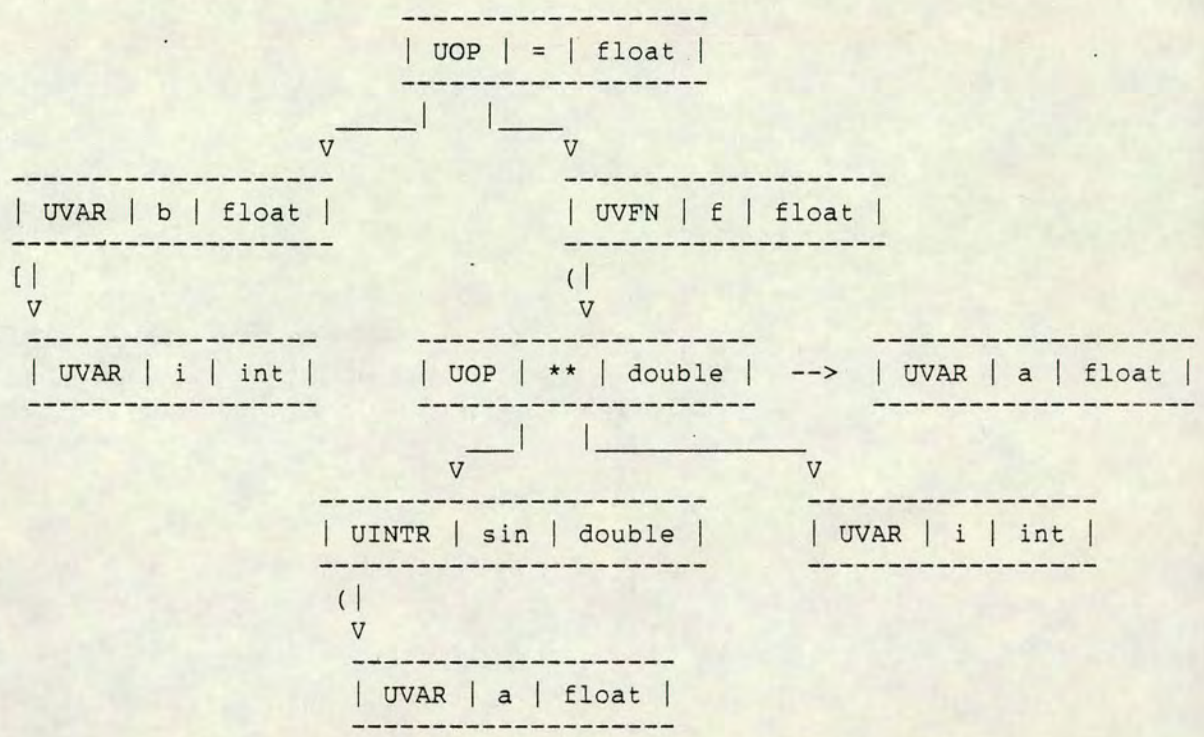


Then the parse tree for the expression

B(I) = F(SIN(A) ** I, A)

where B is a REAL array, F is a REAL function, A is a REAL variable and I is an INTEGER variable, can be drawn (dropping the pointers which are not being used) as Fig. 5.7.

Fig. 5.7 Diagramatic representation of the parse tree for
B(I) = F(SIN(A) ** I, A).



The expression translates into

```
{t_v_1.d = power(sin(a), i); b[i-1] = f(&t_v_1.d, &a);}
```

We notice that the infix operator ** has indeed been translated into the prefix C function power. Moreover, a temporary variable "t_v_1.d" has been declared so that the address of the result from power can be passed into the function f, as is done in FORTRAN.

This creation of temporary variables is by no means trivial and warrants an explanation. To recap, arguments which are constants or expressions, that is, arguments with 'utypes' UCONST, UOP, UVFN, UPFN or UINTR, are assigned to temporary variables and arguments with 'utypes' UVAR or UPAR are preceded by the & operator. Note that addresses of arguments to intrinsic functions must not be taken; similarly addresses of arguments which are intrinsic functions (specified by INTRINSIC) or EXTERNAL routines must not be taken (with the proviso that these arguments do not themselves have arguments - for then they are being called). In the translation pass there is a function 'get_expr' which parses the

"expr" part of the grammar recursively. When it comes across a routine, routine passed as a parameter or an intrinsic function, it calls 'get_args' to parse the arguments. It is 'get_args' which outputs the temporary variables; it works as follows. To get each argument, 'get_args' calls 'get_expr' which will in turn call 'get_args' again if the argument involves another routine, routine passed as a parameter or intrinsic function. This mutual recursion continues until the innermost argument (which must then be a constant, operator, variable or parameter) is reached. It is on the way back up from this recursion that 'get_args' outputs the appropriate temporary variables (and what they equal, by calling 'put_expr' - see below). The temporary variables are declared as follows:

```
typedef union { int i; float f; double d; char *c; } t_v_decl;
t_v_decl t_v_1, t_v_2, ...;
```

and so can hold any type. Hence they can be used sequentially - "t_v_1" can be an INTEGER in one statement (referred to as "t_v_1.i") then a CHARACTER in the next ("t_v_1.c"), for example. 'get_args' keeps two counts of these temporary variables: one for when they first appear on the left-hand side of the expressions generated ('par_t_v_count') and one for their subsequent appearance on the right-hand side of a later expression ('t_v_count') - note that the latter only happens once. 'par_t_v_count' starts at 1 and is incremented after 'get_args' has output a temporary variable; 't_v_count' starts at 0 and is incremented after the argument that the temporary variable is set equal to is output (the first such will always be something which involves no further arguments and so will not require the non-existent zeroth temporary variable). The example translation of the expression

```
K = I( J(1), 2 )
```

will perhaps make this clearer:

```
t_v_1.i = 1;                    /* par_t_v_count = 1 for lhs */
                                /* t_v_count = 0 for rhs */
t_v_2.i = j(&t_v_1.i);          /* par_t_v_count = 2 for lhs */
                                /* t_v_count = 1 for rhs */
t_v_3.i = 2;                    /* par_t_v_count = 3 for lhs */
                                /* t_v_count = 2 for rhs */
```

Of course the "top-level" expression has still to be output but 'get_args' cannot do this because it began analysing the arguments to this expression not the expression itself. In fact the "top-level" expression is output by a call of 'put_expr' immediately after 'get_expr' returns ('get_expr' returns the pointer to the top, or root, of the expression tree and 'put_expr' picks this up). For the above example, this results in

```
k = i( &t_v_2.i, &t_v_3.i );
```

and is done in the following way. 'put_expr' recursively goes through the parse tree of the expression outputting the translated version. When it comes to a routine or a routine passed as a parameter (intrinsic functions and their arguments are output by 'put_intr' - see Sec. 2.5), 'put_expr' calls 'put_args' which outputs the arguments preceded by & or the appropriate temporary variables in their place, depending on the 'utypes'. 'put_args' is called from two different regimes: the first is from 'get_args' when 'get_expr' is parsing the expression and the second is from 'put_expr' when the "top-level" expression is being output. In the first regime, 'put_args' knows which temporary variable to use in an argument replacement from the count 't_v_count'. In the second regime, on the other hand, it must work this out for itself. This is a little tricky and must be done recursively because as well as counting the "top-level" arguments which require temporary variables, the arguments within these arguments requiring temporary variables must also be counted.

We have mentioned the conversion of // and ** into prefix operators, that is, C function calls - this is done by 'put_expr'. It also makes character string assignments, the .EQ. operator used with character strings and character substrings into prefix C function calls:

```
STR = 'Fred'                    strass( str, "Fred", <length of str> )
STR1 .EQ. STR2                  !strcmp( str1, str2 )
STR(I:J)                        strbit( str, i, j )
```

(The functions strass and strbit are output by the translator when the flag "-c" is specified; strcmp is an intrinsic function in C.)

The other difficult job 'put_expr' (actually 'put_array_dimensions' which is called by 'put_expr') must do is the translation of array dimensions. In FORTRAN

77, array dimensions can begin from any integer, start, and go up to any (larger) integer, finish, (though normally start is 1 so finish is equal to the dimension of the array) whereas in C array dimensions range from zero to the dimension of the array minus one, that is, from 0 to finish-start. This means that start must be subtracted from array subscripts in the translation (as is done for b in the above example parse tree, assuming that start is 1). The other alternative, for start = 1, is to declare all arrays to be 1 bigger (in each dimension) then ignore the zeroth element(s) so that 1 need not be subtracted from every array subscript - this is done by the translator when run with the flag "-i". Furthermore, FORTRAN stores its arrays in column-order, that is, the left-most subscript varies fastest, whereas C (like everybody else) stores arrays in row-order, that is, the right-most subscript varies fastest. This makes a difference (to the number of page faults and hence program run time) when stepping sequentially through a large array so the translator reverses the array dimensions thus rendering the storage patterns identical. This is also essential for EQUIVALENCE statements to equivalence properly. However, this reversing of array dimensions can be overridden by specifying the flag "-r". There is also a difference in the way character strings are stored: FORTRAN has strings ranging from 1 to the length, whereas C has them ranging from 0 to the length with the last position containing an end of string (EOS) character. Therefore, again, 1 must be subtracted from all substring indices. (Note that the flag "-i" does not stop this happening.) Unfortunately, this EOS character at the end means that character strings will not line up properly in all translated EQUIVALENCE statements; and that special routines must be written to handle strings in general - these are included in the translation by specifying the flag "-c".

We now come to the translation specific to DAP FORTRAN expressions. Following the above, we discuss array dimensions first. In DAP FORTRAN the constrained dimensions of matrices and vectors come first, whereas in GRID extended C (GEC) they come last. Therefore the translator moves them (even if the flag "-r" has been specified; and, of course, the flag "-i" has no effect). In declarations, it replaces the null dimensions with the size of the DAP for GEC. Hence Appendix I (I.1) and (I.2) become Appendix II (II.2) and (II.1) respectively. DAP FORTRAN matrices and vectors may also be indexed when they appear on the right-hand side (Appendix I.II.II) or left-hand side (Appendix I.II.III) of expressions. The former case is used for two purposes. Firstly, to select an element from a matrix or vector (I.3a) - this is translated into the GEC intrinsic

function element (II.9) - or to select a row or column from a matrix (I.3b) - this is translated into the functions row or col (II.6). Secondly, to route or shift matrices or vectors by one place only (I.4) - this is translated, like shifts of more than one place, into the appropriate GEC shift functions (II.5). The latter case is used for conditional execution (Appendix I.II.III) which is done by the **where** construct in C (Appendix II.II.III), so that (I.5) becomes an instance of (II.3), namely,

```
where (L)
    A = 0.0;
```

Finally there are four extra operators in DAP FORTRAN: .NOR., .NAND., .LEQ. and .LNEQ. (Appendix I.II.II). The latter two present no difficulty; the former two must be translated as follows:

A .NOR. B	!(a || b)
A. NAND. B	!(a && b)

5.2.5. Intrinsic functions

C and GRID extended C have almost identical intrinsic functions as FORTRAN and DAP FORTRAN, so the translation is mostly straightforward. We begin with the FORTRAN to C intrinsic function translation, which is summarised in Table 5.4.

Table 5.4

FORTRAN intrinsic functions with corresponding C translations.

FORTRAN	C
SQRT, DSQRT	sqrt
EXP, DEXP	exp
LOG, DLOG, ALOG	log
LOG10, DLOG10, ALOG10	log10
SIN, DSIN	sin
COS, DCOS	cos
TAN, DTAN	tan
ASIN, DASIN	asin
ACOS, DACOS	acos
ATAN, DATAN	atan
ATAN2, DATAN2	atan2
SINH, DSINH	sinh
COSH, DCOSH	cosh
TANH, DTANH	tanh
AMOD, DMOD	fmod
AINT, DINT	fint
NINT, JNINT, IDNINT, ANINT, DNINT	fnint
DBLE, DFLOAT	(double)
DPROD	(double) *
REAL, SNGL, FLOAT	(float)
ABS, IABS, DABS	abs
MOD	% +
ICHAR	ichar
MIN, MIN0, MIN1, AMIN0, AMIN1, DMIN1	minf
MAX, MAX0, MAX1, AMAX0, AMAX1, DMAX1	maxf
DIM, IDIM, DDIM	dim
SIGN, ISIGN, DSIGN	sign
INT, IFIX, IDINT	(int)
LLT LLE LGT LGE	strcmp #
LEN	strlen
INDEX	index
CHAR	itoa

* i.e.	dprod(x,y)	->	((double)(x*y))
+ i.e.	mod(i,j)	->	((i) % (j))
# e.g.	llt(a,b)	->	(strcmp(a,b) < 0)

All the C functions listed there are intrinsic except for "fint", "fnint", "abs", "minf", "maxf", "dim" and "sign" (these are defined as the following macros in C:

```
#define fint(A)          ((A) < 0 ? ceil(A) : floor(A))
#define fnint(A)         (fint(A + .5 * sign(A)))
#define abs(A)           ((A) < 0 ? -(A) : A)
#define minf(A,B)        ((A) < (B) ? (A) : (B))
#define maxf(A,B)        ((A) > (B) ? (A) : (B))
#define dim(A,B)         ((A) > (B) ? ((A)-(B)) : 0)
#define sign(A,B)        ((B) < 0 ? ((A) < 0 ? A : -(A))
                                  : ((A) < 0 ? -(A) : A))
```

and output by the translator when the flag "-m" is specified) and also "ichar", "itoa" and "index" (which are functions to convert a character to an integer, convert an integer to a character, and return the index of one string in another, respectively; they are output by the translator with "-c"). There are three special cases (indicated in Table 5.4):

1) FORTRAN type conversion intrinsic functions (DBLE ..., REAL ..., INT ... and DPROD) translate into unary type casts in C ((**double**), (**float**), (**int**) and (**double**));

2) the prefix intrinsic function MOD translates into the infix operator %;

3) character string intrinsic functions (LLT, LLE, LGT and LGE) translate into C intrinsic function calls to strcmp.

Turning to the DAP FORTRAN to GRID extended C translation, we summarise this in Table 5.5.

Table 5.5

DAP FORTRAN intrinsic functions with corresponding GRID extended C translations.

DAP FORTRAN	GEC
MERGE(M,-M,L)	mergei(m, -m, l)
SHLC(V)	vshftc(v, -1);
SHRP(V,33)	vshftp(v, 33);
SHLC(M)	m; ERROR: shlc cannot cope with longvectors - shift ignored
SHRP(M,55)	m; ERROR: shrp cannot cope with longvectors - shift ignored
SHNC(M)	shnc(m, 1);
SHWP(M,55)	shwp(m, 55);
SUMR(M)	sumri(m);
SUMC(M)	sumci(m);
ANDROWS(M)	allr(m);
ANDCOLS(M)	allc(m);
ORROWS(M)	anyr(m);
ORCOLS(M)	anyc(m);
MATR(V)	matr(v);
MATC(V)	matc(v);
SUM(M)	sum(m);
MAXV(M)	max(m);
MINV(M)	min(m);
ALL(M)	all(m);
ANY(M)	any(m);
MAT(31)	31; WARNING: mat ignored - conversion automatic
VEC(32)	32; WARNING: vec ignored - conversion automatic
CALL CONVFMt(F)	stop(f);
CALL CONVMFt(M)	ptos(m);
CALL CONVFVt(F)	stop(f); WARNING: 2nd & 3rd arguments ignored in vector mode conversion - only 1 vector with 64 cmpts converted
CALL CONVVFt(V)	ptos(v); WARNING: 2nd & 3rd arguments ignored in vector mode conversion - only 1 vector with 64 cmpts converted
CALL CONVFSt(F)	WARNING: scalar mode conversion redundant
CALL CONVSFt(F)	WARNING: scalar mode conversion redundant
CALL CONVVMt(V)	convvm1(v);
CALL CONVMVt(M)	convmv1(m);
ALTR(12)	rowset(64, 64, 12-1, 12, 12);
ALTC(11)	colset(64, 64, 11-1, 11, 11);
ALT(10)	vecset(64, 10-1, 10, 10);
ROW(14)	rowset(64, 64, 14-1, 1, 64);
COL(13)	colset(64, 64, 13-1, 1, 64);
ROWS(22,6)	rowset(64, 64, 22-1, 6-(22)+1, 64);
COLS(21,5)	colset(64, 64, 21-1, 5-(21)+1, 64);

Firstly, the intrinsic function MERGE, which is used for conditional execution, is translated simply into a GEC version with the correct type (which is output by the translator when the flag "-p" is specified). For example, to deal with (I.6) we use

```
double_array merged(a, b, l)[64,64]   /* for double, float */
double_array a[64,64], b[64,64]; bool_array l[64,64];
{
    double_array res[64,64];
    where (l) res = a; else res = b;
    return(res);
}
```

We will now go through the rest of the intrinsic functions in the order they are dealt with in Appendices I.II.IV and II.II.IV.

1) Routing

As the architectures of the DAP and GRID are basically the same, these intrinsic functions are trivial to translate: (I.7) is changed directly into (II.4) and (I.8) into a subset of (II.5), with the appropriate signs of count. However, there are no longvectors on the GRID so (I.7) with these will fail to translate.

2) Matrix to vector

In the first implementation of GEC, there are not any intrinsic functions analogous to those in (I.9) so they must be coded by hand. This is relatively easy: for example, the intrinsic function SUMR with an argument of type INTEGER becomes

```
int_array sumri(a)[64]                /* for int, short, long */
int_array a[64,64];
{
    int_array res[64];
    int i;
    res = 0;
    for (i=0; i<64; i++)
          res += row(a, i);
    return(res);
}
```

and the intrinsic function ANDCOLS becomes

```
bool_array allc(l)[64]
bool_array l[64,64];
{
    bool_array res[64];
    int i;
    res = 0;
    for (i=0; i<64; i++)
        res = res && col(l, i);
    return(res);
}
```

3) Vector to matrix

The intrinsic functions in (I.10) translate trivially into those in (II.7).

4) Array to scalar

The intrinsic functions in (I.11) translate directly into those in (II.8). Again, there are no analogous functions in GEC to those in (I.12) but they can be easily coded by hand; for example, ALL becomes

```
int all(l)
bool_array l[64,64];
{
    bool_array int_res[64];
    int res, i;
    int_res = 0; res = 0;
    for (i=0; i<64; i++)
        int_res = int_res && row(l, i);
    for (i=0; i<64; i++)
        res = res && element(int_res, i);
    return(res);
}
```

5) Conversion

The FORTRAN array to/from DAP FORTRAN matrix conversions (I.13a) translate to (II.10); as do the FORTRAN array to/from DAP FORTRAN vector conversions (I.13b) provided that e = 64 and v = 1. The FORTRAN scalar to/from DAP FORTRAN scalar conversions (I.13c) become redundant in GEC; and the vector to/from matrix conversions (I.13d) are not (yet) defined on the GRID.

6) Masking

The intrinsic functions in (I.14), (I.15) and (I.16) can all be done using those in (II.11) as shown in Table 5.5.

5.3. Concluding remarks

The translator software described above is all written in C. There are four main sections of code: the lexical analyser, the prepass, the translation pass, and the recursive expression parser. Altogether there is approximately 9800 lines, or 232Kbytes of code (including comments and white space). Both the prepass and the translation pass deal with 40 lines of (serial or parallel) FORTRAN per second, thus the overall translation speed is 20 lines/sec.

The translator has been used in practice to convert a 400 line FORTRAN (molecular dynamics) program into C, in order to run it on a new parallel computer, developed by Bolt, Beranek and Newman (BBN), called the Butterfly Parallel Processor. Numerous DAP FORTRAN programs have also been converted into GRID extended C as a check on the translator. Unfortunately, neither the GRID nor a software simulator of it was completed in time to verify the correct functionality of the translated GEC.

Appendix I
The DAP

The International Computers Limited (ICL) Distributed Array Processor (DAP) was begun in 1972. By 1976 a pilot DAP (Reddaway, 1973; Flanders, Hunt, Reddaway and Parkinson, 1977) with a 32x32 array of processing elements (PEs), each with 1Kbit of memory, was completed. The first production model (Reddaway, 1979; Parkinson, 1983) was installed at Queen Mary College London in 1980. It consists of a 64x64 array of PEs, each having 4Kbits of memory giving a total of 2Mbytes, and is implemented in SSI on 256 PCBs (each containing 16 processors and associated memory) with a clock cycle of 200ns. It is intimately connected to a host ICL 2900 series mainframe computer. A second generation DAP, produced in 1986 and called the Mil-DAP, is a 32x32 PE array in LSI on 16 PCBs, with a clock cycle of 155ns and 2Mbytes of memory. In addition it has two inbuilt fast I/O buffers each with a capacity of 16Kx32bits which can be configured for both data input and output; maximum data transfer rate is 40Mbytes per second. Mil-DAP attaches to the ICL PERQ2 workstation. The third generation DAP, to appear in 1987, will come in a range of PE array sizes from 8x8 (with 1Mbyte memory) to 64x64 (with 64Mbytes memory) and will be VLSI (like the GRID - Appendix II). In this appendix we shall describe the hardware and software of the first generation DAP.

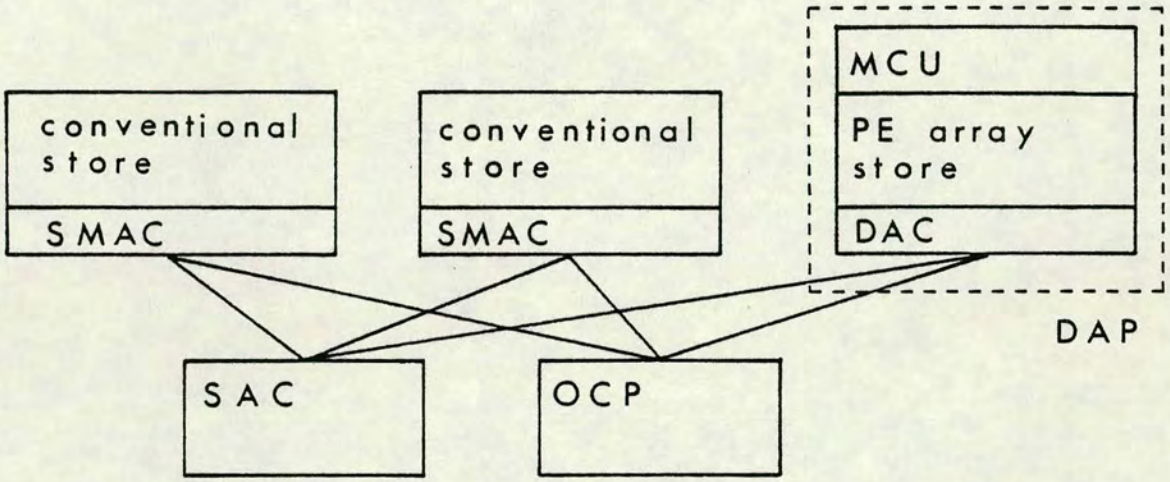
I.I. Hardware

The DAP is a Single Instruction stream, Multiple Data stream (SIMD) computer (Hockney and Jesshope, 1981) comprising a 64x64 square array of bit-serial processing elements (PEs), each with 4Kbits of local memory and connections to the four nearest neighbours. All 4096 PEs execute identical instructions, which are broadcast by the master control unit (MCU), simultaneously, on their own independent data. When it is not functioning autonomously under the control of its MCU, the DAP can act as a (2Mbyte) memory module of the host ICL 2900 series mainframe computer.

I.I.I. Host ICL 2900

A typical ICL 2900 series system, illustrated in Fig. I.1, consists of an order code processor (OCP) and a store access controller (SAC) both cross-connected to a number of memory modules.

Fig. I.1 Schematic diagram of an ICL 2900 series system containing a DAP (SMAC, store multiple access controller; DAC, DAP access controller; MCU, master control unit).

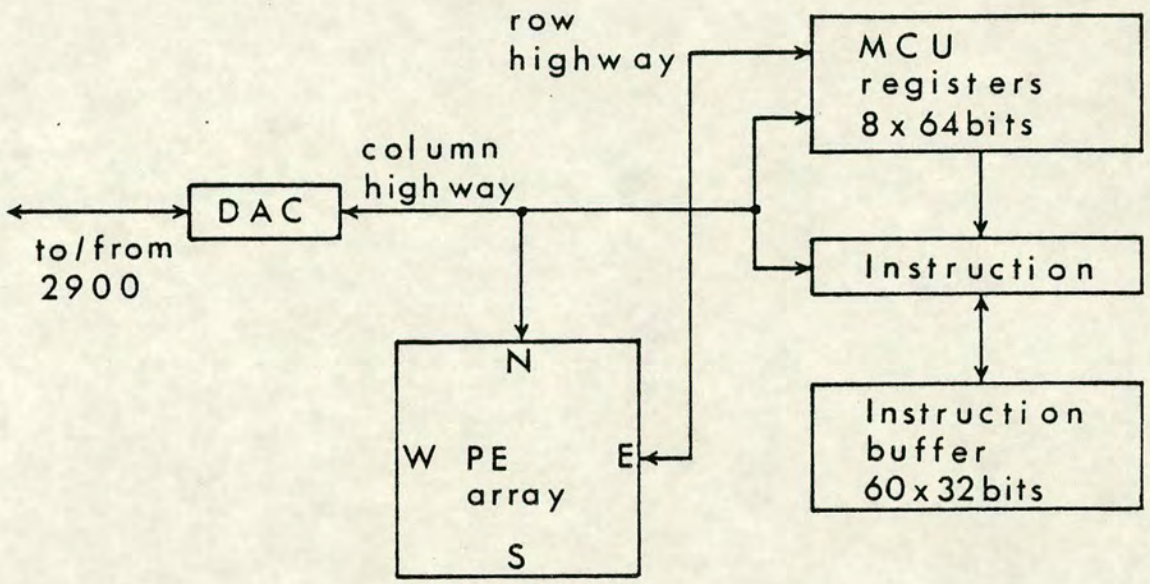


One or more of these memory modules may be a DAP, which provides memory in the conventional way and may also be instructed by the OCP to execute its own DAP code. If the DAP is considered as the main processor in the system then the other conventional stores can be considered as fast backing store to the DAP and the OCP as a pre- and post-processor.

I.I.II. DAP unit

The major components and data highways of the DAP unit are shown in Fig. I.2.

Fig. I.2 The major components and data highways of the DAP unit.

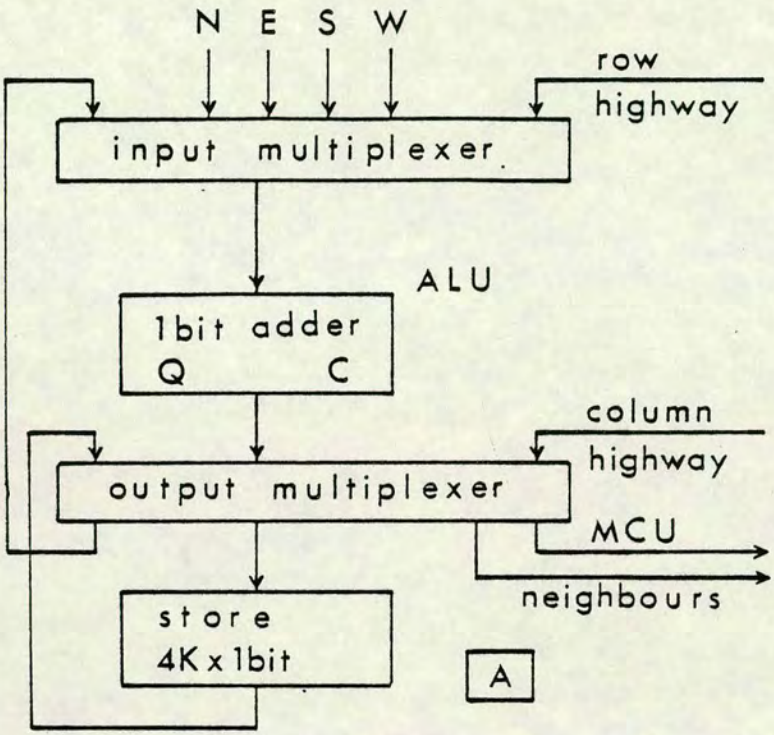


The DAP access controller (DAC) along with the 64bit-wide column highway provide the interface to the rest of the 2900 system. One 2900 mainframe 64bit word corresponds to a row across the DAP memory. The column highway also provides a path between rows of the DAP PE array and registers in the MCU, which can be used for data and/or instruction modification. Finally, the column highway provides the path for the MCU to fetch DAP instructions from the DAP store. DAP instructions are stored two per row and one row is fetched from memory in one clock cycle. Instructions within a special hardware DO-loop instruction are stored in the instruction buffer for repeated execution. There is also a row highway which is used exclusively for transmitting data to and from the MCU registers.

I.I.III. PE array

The various components and data paths which comprise a processing element are shown schematically in Fig. I.3.

Fig. I.3 The main components and data paths of the DAP PE.



The PE array is connected two-dimensionally, each PE being connected to four neighbours which may be defined by the points of the compass: N, S, E and W. The connections at the edge of the array are defined by the geometry of the instruction being executed. This may be planar, defining a zero input at the edges, or cyclic, giving periodic connections, independently in the rows or columns of the array. Within the processor, a 1bit full adder along with the accumulator (Q) and carry (C) registers make up the arithmetic and logic unit (ALU). The adder adds Q, C and the input to the PE, giving the sum and the carry outputs in the Q and C registers respectively, unless an "add to store" instruction is being executed in which case the sum is written back to the location that the operand came from - this saves half a clock cycle over an "accumulator add" followed by an "accumulator store" and is used to speed up multi-bit arithmetic. There is also an activity register (A) which provides programmable control over

the action of the PE since certain store instructions are enabled only if it is set.

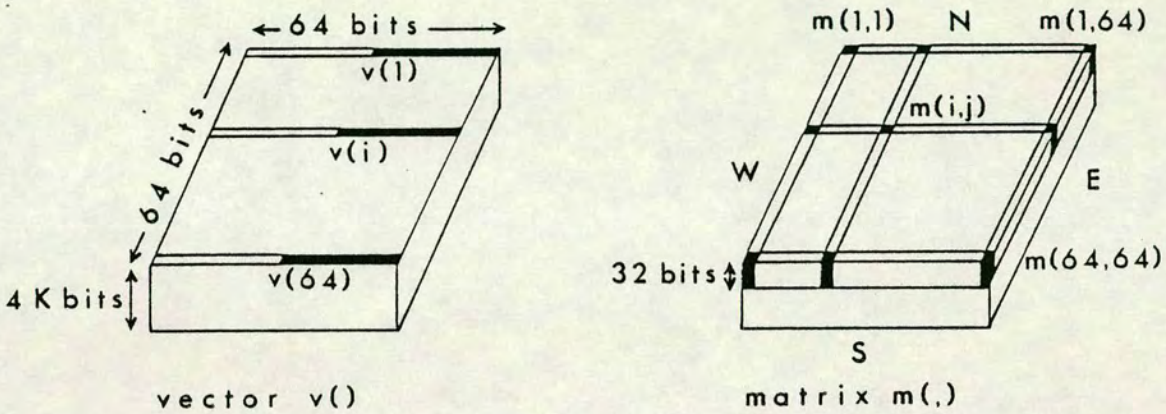
I.II. Software

Programs for the DAP (and the rest of the 2900 system) consist of two parts: a serial part (written in standard FORTRAN (77)) which executes on the host 2900 and a parallel part (written in a parallel extension of FORTRAN IV called DAP FORTRAN) for the DAP. Communication between DAP FORTRAN and FORTRAN routines is accomplished through the use of shared COMMON blocks, which are held in the DAP store. (An array processor assembly language (APAL) is also provided for the DAP with interfaces to both FORTRAN and DAP FORTRAN, but when using the highly optimised floating-point arithmetic and system routines there is little benefit to be gained from using APAL. It only comes into its own for algorithms which exploit the bit-serial nature of the PEs when orders of magnitude performance improvements may be achieved by coding at the assembler level.) The parallel extensions found in DAP FORTRAN can be discussed under four headings: declarations, expressions, conditional execution and intrinsic functions.

I.II.I. Declarations

In DAP FORTRAN there are three kinds, or modes, of data item: scalar, vector and matrix, which may be of any FORTRAN type (INTEGER, REAL, DOUBLE PRECISION, LOGICAL or CHARACTER). Scalar variables and arrays correspond to FORTRAN variables and arrays and are processed serially; vectors and matrices consist of a number of component values, or elements, and are processed in parallel. A vector is a one-dimensional set of 64 elements and a matrix is a two-dimensional set of 64x64 elements. A vector is stored (right-justified) horizontally along the rows of a single DAP store plane and a matrix is stored vertically under the PEs as a contiguous set of n DAP planes, where n is the number of bits in the internal representation of each matrix element. This is illustrated in Fig. I.4.

Fig. I.4 How vectors and matrices are stored in the DAP.



Vectors are declared with their first dimension null and matrices are declared with their first two dimensions null, for example,

```
REAL*8 V()                                    (I.1)

INTEGER M(,)                                  (I.2)
```

These are the constrained dimensions which take on the DAP size of 64. A matrix may also be regarded as a one-dimensional set of 4096 component values, obtained by placing successive columns of the matrix end to end, called a longvector (this feature is not present in the GRID - Appendix II).

I.II.II. Expressions

Simple assignment and expression evaluation in DAP FORTRAN are basically the same as for FORTRAN, the only difference being that they may be vector or matrix mode as well as scalar mode. Hence a vector (or matrix) may be assigned

to another vector (or matrix) in a single assignment. Operators act on entire vectors (or matrices) combining corresponding elements. Four extra logical operators are provided in DAP FORTRAN: .NOR. and .NAND. (the logical converse of .OR. and .AND.) and .LEQ. and .LNEQ. (logical equivalence and non-equivalence). Scalars are automatically converted to the appropriate mode in vector and matrix mode expressions; however vectors are not expanded to matrices automatically since such an expansion can be made in two ways (intrinsic functions are provided for this - Sec. II.IV).

Vectors and matrices may be indexed in expressions. If the vector or matrix is on the right hand side of the expression then indexing selects a value; if on the left hand side then indexing identifies one or more vector or matrix elements to which the value of the right hand side is assigned. The latter case is a form of conditional execution and will be discussed in the next section. In the former case, selection is from constrained dimensions and is rank-reducing: selection from a matrix yields either a vector or a scalar and selection from a vector always gives a scalar. Examples are

V(I)	element I of vector V	
M(I,J)	element I,J of matrix M	(I.3a)
M(I,)	row I of matrix M	
M(,J)	column J of matrix M	(I.3b)

In addition, routing or shifts, by one place only, can also be applied as an indexing operation, using '+' or '-' in either of the constrained dimensions. (What happens at the edges depends on the geometry - planar or cyclic - set up by the GEOMETRY statement.) Some examples for cyclic geometry, along with the equivalent intrinsic function calls, are

V(+)	SHLC(V)	shift vector left
M(+,)	SHNC(M)	shift matrix north
M(,-)	SHEC(M)	shift matrix east
M(-)	SHRC(M)	shift matrix, treated as longvector, right (I.4)

Shifts of greater than one place are performed by the intrinsic functions only, see Sec. II.IV. Note that null indices, like no indices, select the whole vector or matrix.

I.II.III. Conditional execution

Selection of which component values of a vector or matrix are affected by an operation is achieved through the use of left hand side indexing or the intrinsic function MERGE (the latter alternative is discussed in the next section). In left hand side indexing, a logical vector (or matrix) L is used as the index for another vector (or matrix) A so that wherever L is true the corresponding element of A is assigned the value of the right hand side of the expression and wherever L is false the element of A retains its original value. For example

```
A(L) = 0.0                                                    (I.5)
```

I.II.IV. Intrinsic functions

Firstly, we have the intrinsic function MERGE which is used for conditional execution. MERGE returns a vector (or matrix) whose elements are selected from elements of the first and second arguments depending on whether the corresponding element of the third argument is true or false respectively. For example

```
L = A .GT. 0.0
B = SQRT( MERGE(A, 0.0, L) )                                  (I.6)
```

will set B equal to the square root of A wherever A is greater than 0.0 and to 0.0 otherwise.

The other intrinsic functions include:

1) Routing

Shifts by one place only can be written as indexed expressions, as detailed in Sec. II.II, but more general shifts require the following intrinsic functions

```
SHLg( vector or matrix (longvector), count )   shift left
SHRg( vector or matrix (longvector), count )   shift right    (I.7)
```

where g can be 'P' for planar edge connections or 'C' for cyclic edge connections. The effect of SHLg is that element(i) := element(i + count) and the effect of SHRg

is that element(i) := element(i - count).

SHNg(matrix, count)	shift north	
SHEg(matrix, count)	shift east	
SHSg(matrix, count)	shift south	
SHWg(matrix, count)	shift west	(I.8)

2) Matrix to vector

SUMR(matrix)	sums rows of matrix into a vector .	
SUMC(matrix)	sums columns of matrix into a vector	
ANDROWS(logical matrix)	ANDs rows into a logical vector	
ANDCOLS(logical matrix)	ANDs columns into a logical vector	
ORROWS(logical matrix)	ORs rows into a logical vector	
ORCOLS(logical matrix)	ORs columns into a logical vector	(I.9)

3) Vector to matrix

MATR(vector)	returns a matrix of identical rows	
MATC(vector)	returns a matrix of identical columns	(I.10)

4) Array to scalar

SUM(array)	returns sum of all elements	
MAXV(array)	returns maximum element	
MINV(array)	returns minimum element	(I.11)
ALL(logical array)	ANDs all the elements	
ANY(logical array)	ORs all the elements	(I.12)

5) Conversion

CONVFMt(matrix)	FORTRAN array to DAP FORTRAN matrix	
CONVMFt(matrix)	DAP FORTRAN matrix to FORTRAN array	(I.13a)
CONVFVt(vector, e, v)	FORTRAN array to DAP FORTRAN vector	
CONVVFt(vector, e, v)	DAP FORTRAN vector to FORTRAN array	(I.13b)
CONVFSt(scalar, s)	FORTRAN scalar to DAP FORTRAN scalar	
CONVSFt(scalar, s)	DAP FORTRAN scalar to FORTRAN scalar	(I.13c)
CONVVMt(vector)	vector to matrix	
CONVMVt(matrix)	matrix to vector	(I.13d)

where t is the size of the element in bytes, e is the number of elements in each vector to be converted, v is the number of vectors in the conversion and s is the number of scalars in the conversion.

6) Masking

```
ALTR( i )
ALTC( i )
ALT( i )                                                    (I.14)
```

ALTR (ALTC) returns a logical matrix which has its first i rows (columns) false, the next i rows (columns) true and so on in alternation until all the elements of the matrix have a value. (If i is zero then all the elements are false.) ALT does the same for vectors.

```
ROW( i )
COL( i )                                                    (I.15)
```

ROW (COL) returns a logical matrix with false values everywhere except for row (column) i where they are true.

```
ROWS( i, j )
COLS( i, j )                                                (I.16)
```

ROWS (COLS) returns a logical matrix with elements in rows (columns) i to j inclusive given the value true and all others set false.

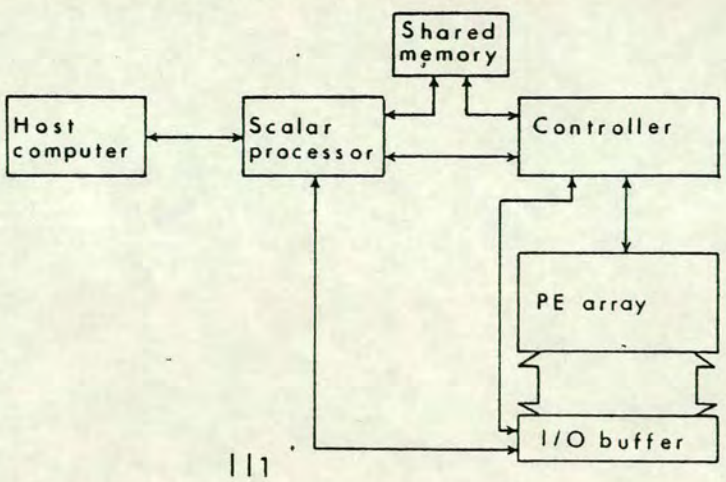
Appendix II
The GRID

The General Electric Company (GEC) Rectangular Image and Data processor (GRID) was begun in 1982. It is superficially like the DAP but differs both in gross architecture (in particular, it does not form part of a host computer) and in PE design and connectivity; moreover, it is implemented in very large scale integration (VLSI) integrated circuit (chip) technology (Arvind, Robinson and Parker, 1983; Pass, 1984). In this appendix we shall describe the hardware and software of the GRID.

II.1. Hardware

The GEC Rectangular Image and Data processor (GRID) is a SIMD computer with an architecture similar to that of the DAP - it contains a 64x64 square array of bit-serial processing elements (PEs). Each PE has 8Kbits of local memory and connections to all eight neighbouring PEs. A central controller broadcasts a sequence of instructions to the array so that each PE performs the same operation simultaneously on its own local data. The instruction sequences are supplied to the controller by a scalar processor via shared memory. On completion of one sequence the controller interrupts the scalar processor to request another. The scalar processor can also perform calculations while the array is functioning. The scalar processor and controller are 16bit processors; they, along with the PE array, form the GRID system which is hosted by a multi-user, (mini-)computer. The general layout is shown in Fig. II.1.

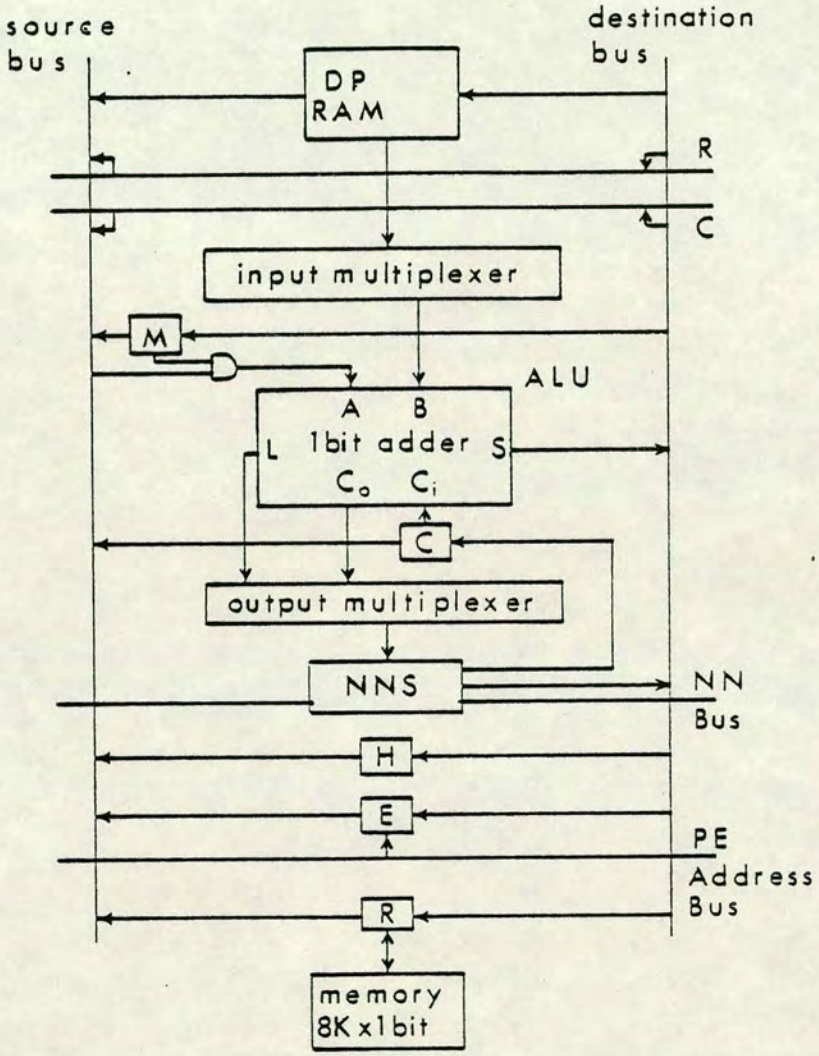
Fig. II.1 Schematic diagram of the GRID system.



II.1.1. PE array

The complete 64x64 array of PEs is composed of 128 VLSI GRID chips; each GRID chip containing an 8x4 sub-array. The structure of the bit-serial PE is shown schematically in Fig. II.2.

Fig. II.2 The GRID PE.



It has a two-operand structure. The ALU can produce any of the sixteen possible logical combinations L of its inputs A and B. It can also perform addition or subtraction using the carry register (C): A, B and C_i are combined to yield the sum or difference in S and the carry in C_o. The multiply register (M) is gated (ANDed) with whatever comes in on the source bus so that a multiply can be performed by repeated addition, or a division by repeated subtraction. There is a histogram register (H) which is used to form sums across the array, that is, count the number of 1's in a bit-plane. Each PE has 64bits of dual-ported (DP) on-chip

random access memory (RAM) which provides a cache for storing, for example, partial results during multiplication and division. The 8Kbits of local memory is off-chip and is accessed via the RAM register (R). The enable register (E) allows the operation of the PE to be controlled independently (just like the activity register does in the DAP - Appendix I.I.III). E can be set either as the result of some calculation or from the PE address bus - this gives a mechanism for geometric control in which selected chips, rows, columns or single PEs can be enabled. The PEs are interconnected (within and across GRID chips) by a nearest neighbour switching network (NNS) which can connect a PE to any of the eight PEs nearest to it. The orthogonal north, south, east and west connections are made directly (forming the NN bus). Diagonal moves are achieved by routing through the NNS of the intermediate orthogonal neighbour. For example, to access data in the neighbouring PE to the north-east the NNS is set up to pass local data south, to transfer input from the north to the west and to accept input from the east as the neighbour's data. This compounded routing takes less time than two orthogonal moves. The PEs are also connected by row (R) and column (C) buses to form a square mesh. There is a single 64bit edge register which may be connected to either of the buses and can be read and written by the controller. This provides the means for the controller to broadcast data to, or extract data from, the array.

II.I.II. Controller

The controller is a microprogrammed processor which provides the low-level interface to the PE array. It reads GRID controller assembler (GCA) instructions from the shared memory. These instructions may cause branching to occur, or may perform scalar arithmetic, or may be broadcast in a decoded form to the PE array. The latter possibility entails rather complex operations (given the bit-serial nature of the array) such as arithmetic, comparison and routing. These are implemented in microcode for maximum efficiency.

The controller also includes special hardware which supports the mapping of larger data arrays onto a smaller PE array, using so-called pyramidal mapping (see Pass, 1984 for a detailed description of this).

II.I.III. Scalar processor

The scalar processor is a standard MC68000 microprocessor. It executes the high-level programming language for the GRID, called GRID extended C (GEC), which is an extension of the language C to include parallel array operations in addition to the usual serial operations (see Sec. II). Serial code executes on the scalar processor whilst parallel sections of code run on the controller/PE array. Instructions to the controller are placed in queues in the shared memory. When the controller reaches the end of an instruction queue, it interrupts the scalar processor to request another. This is a much better arrangement than that found in the DAP where instructions are actually stored in the array memory, thus wasting data space.

II.I.IV. I/O buffer

Most real-time devices (for example, TV cameras and monitors) and mass-storage units (for example, disks) handle data in bit-parallel, word-serial form; whereas the PE array operates in a word-parallel, bit-serial fashion. The transformation from one format to the other is effected by the "corner-turning" I/O buffer which is capable of buffering lines of up to 512 16bit words. The I/O buffer works concurrently with the PE array - only interrupting it when a whole line has been read in, or written out.

II.I.V. Host computer

The host computer provides a multi-user environment suitable for the development and maintenance of programs for the GRID. It can be any (mini-)computer running the Unix operating system, since it must only interface to the scalar processor which is a standard microprocessor - not the specialized controller.

II.II. Software

As the GRID system contains three processors - PE array, controller and scalar processor - it is programmed at three levels: microcode, assembler and high-level language.

Instructions broadcast to the PE array by the controller are microcoded for maximum efficiency. This is the very lowest software level and is in the realm of the system implementer.

The controller is programmed in GRID controller assembler (GCA). Frequently used functions for image, signal and numerical processing will be coded in GCA to form part of the system software library. The specialist user will program at this level when execution performance is critical.

The scalar processor is programmed in the high-level GRID extended C (GEC) programming language which is essentially the language C with appropriate parallel extensions. Serial code is compiled and executed on the scalar processor as for C; parallel sections are complied into GCA and run on the controller/PE array. The parallel extensions fall into four main categories, namely, declarations, expressions, conditional execution and intrinsic functions.

II.II.I. Declarations

Type specifiers are provided in GEC for the declaration of parallel array types. (This is simpler than in DAP FORTRAN where modes are introduced - Appendix I.II.I.) Most of these type specifiers are parallel extensions of the usual types found in C, viz, **char_array**, **short_array**, **int_array**, **long_array**, **float_array** and **double_array**; but there is also a new 1bit data type specified by **bool_array**. Two-dimensional parallel arrays, denoted by the term matrix, are declared as follows

```
array_type_spec identifer [ row_spec, col_spec ];          (II.1)
```

where row_spec and col_spec are constant integer expressions defining respectively the number of rows and columns in the matrix. These must be

power of two multiples of the corresponding GRID PE array dimensions. One-dimensional parallel arrays, denoted by the term vector, are declared as follows

```
array_type_spec identifer [ dim_spec ];
```
(II.2)

where dim_spec is the number of elements in the vector and a power of two multiple of the GRID array row dimension. A vector can also be declared as **packed** which advises the compiler to store the vector with several elements packed into each GRID array row (rather than the default situation of only one per row, as is done on the DAP - Appendix I.II.I), saving on memory and increasing performance through greater parallelism. For example, for

```
packed char_array x[1024];
```

on a 64x64 GRID it is possible to store eight elements of x in each row.

II.II.II. Expressions

Parallel array expressions are written in a very similar manner to standard C expressions. All of the binary operations (except the shift operators >>, <<, >>=, <<= of course) can be used to combine either an array with an array, or an array with a scalar. In the latter case the scalar is (conceptually) expanded into an array of identical elements. Each operator is applied on a pointwise basis, combining corresponding array elements. A parallel array expression may contain mixed types but the arrays must have the same dimensionality. Arrays appear without their dimension specifier(s) in expressions since all elements are dealt with simultaneously. We note that there are no special indexing expressions like those found in DAP FORTRAN - Appendix I.II.II. Such operations are performed solely by intrinsic functions in GEC, these are discussed in Sec. II.IV.

II.II.III. Conditional execution

Control over the operations applied to the individual elements of a parallel array is exercised with the **where** construct. Its format is as follows

```
where ( parallel_array_expr )
    statement_1
else                                                    (II.3)
    statement_2
```

Statements 1 and 2 can be simple or compound and the **else** clause is optional, as for standard C. The parallel array expression is evaluated to yield a true/false parallel predicate (mask) which controls parallel operations within statements 1 and 2. Within statement_1, where the parallel array expression is true (that is, non-zero) assignment to corresponding elements is enabled; within statement_2, assignment is enabled where the mask is false.

II.II.IV. Intrinsic functions

A number of intrinsic functions are provided for manipulating parallel array expressions (compare these with the intrinsic functions in DAP FORTRAN - Appendix I.II.IV). They include:

1) Routing

```
vshftg( vector, count )        vector shift                 (II.4)
```

where g can be 'p' for planar edge connections or 'c' for cyclic edge connections. The effect of this instruction is that element(i + count) := element(i).

```
shng( matrix, count )          shift north
shneg( matrix, count )         shift north-east
sheg( matrix, count )          shift east
shseg( matrix, count )         shift south-east
shsg( matrix, count )          shift south
shswg( matrix, count )         shift south-west
shwg( matrix, count )          shift west
shnwg( matrix, count )         shift north-west
shiftg( matrix, rel_row, rel_col )                      (II.5)
```

where shiftg shifts the matrix by the relative row and column values, for example,

```
shnp( matrix, 2) ≡ shiftp( matrix, -2, 0 )
shsec( matrix, 1 ) ≡ shiftc( matrix, 1, 1 ).
```

2) Matrix to vector

row(matrix, i)	returns row i of matrix as a vector	
col(matrix, j)	returns column j of matrix as a vector	(II.6)

3) Vector to matrix

matr(vector)	returns a matrix of identical rows	
matc(vector)	returns a matrix of identical columns	(II.7)

4) Array to scalar

sum(array)	returns sum of all elements	
max(array)	returns maximum element	
min(array)	returns minimum element	(II.8)
element(matrix, i, j)	returns element at i,j	
element(vector, i)	returns element at i	(II.9)

5) Conversion

ptos(array, pointer)	parallel array to scalar array	
stop(pointer, array)	scalar array to parallel array	(II.10)

where pointer is assumed to be the address of a buffer in scalar processor memory which is at least as large as the parallel array.

6) Masking

rowset(nrow, ncol, row, width, period)	
colset(nrow, ncol, col, width, period)	
vecset(ndim, dim, width, period)	(II.11)

rowset returns a bool_array of nrow rows by ncol columns holding a pattern of horizontal stripes (background has value zero; stripes have value one) starting at row row (where the top of the array is row zero), being width elements wide and repeated at intervals of period elements. colset operates similarly for columns, creating vertical stripes. vecset does the same for vectors.

7) Resampling

Since matrices can be any power of two multiple of the size of the GRID PE array there is an intrinsic function

```
sample( matrix, i, j, ni, nj, si, sj )
```
(II.12)

which extracts a sub-array from the matrix and maps it across the PE array. (i,j) specifies the top left hand corner of the sub-array, ni and nj specify the number of rows and columns in the sub-array and si and sj specify the sample interval. (This is not found in the DAP because matrices there are all the same size as the PE array.)

8) I/O

Unlike the DAP, the GRID can communicate with its host via files using the intrinsic functions

```
input( fd, word-length, array )
output( fd, array )
```
(II.13)

where fd is a file descriptor (returned by a call to open in C) and word-length is the number of bits per element of the incoming data.

References

Aho A. V. and Ullman J. D., 1977, Principles of Compiler Design (Addison-Wesley, Massachusetts)

Arnison G. *et al* [UA1 collaboration], 1983a, Phys. Lett. 122B 103

Arnison G. *et al* [UA1 collaboration], 1983b, Phys. Lett. 126B 398

Arvind D. K., Robinson I. N. and Parker I. N., 1983, Int. Symp. on Circuits and Systems (IEEE) 405

Bagnaia P. *et al* [UA2 collaboration], 1983, Phys. Lett. 129B 130

Baillie C. F., 1986a, Proc. 6th Summer School on Computing Techniques in Physics (to appear in Comput. Phys. Commun.)

Baillie C. F., 1986b, (DAP) FORTRAN to (GRID extended) C translator: Users Manual, Edinburgh Preprint 86/373

Baillie C. F., 1986c, (DAP) FORTRAN to (GRID extended) C translator: Maintainers Manual, Edinburgh Preprint 86/374

Bander M., 1976, Phys. Rev. D13 1566

Banner M. *et al* [UA2 collaboration], 1983, Phys. Lett. 122B 476

Barbour I. M., Behilil N.-E., Dagotto E., Karsch F., Moreo A., Stone M. and Wyld H. W., 1986, Problems with Finite Density Simulations of Lattice QCD, Illinois Preprint ILL-TH-86-23

Barbour I. M., Behilil N.-E., Gibbs P. E., Rafiq M., Moriarty K. J. M. and Schierholz G., 1985a, Updating Fermions with the Lanczos Method, DESY Preprint 85-141

Barbour I. M., Behilil N.-E., Gibbs P. E., Schierholz G. and Teper M., 1985b, in Lecture Notes in Physics, The Recursion Method and its Applications (Springer-Verlag, Berlin, Heidelberg, New York, Tokyo)

Barbour I. M., Gibbs P., Bowler K. C. and Roweth D., 1985, Phys. Lett. 158B 61

Barbour I. M., Gibbs P., Gilchrist J. P., Schneider H., Schierholz G. and Teper M., 1984, Phys. Lett. 136B 80

Barbour I. M., Gilchrist J. P., Schneider H., Schierholz G. and Teper M., 1983, Phys. Lett. 127B 433

Batrouni G. G., Kutz G. R., Kronfeld A. S., Lepage G. P., Svetitsky B. and Wilson K. G., 1985, Phys. Rev. D32 2736

Bernard C., Draper T., Olynyk K. and Rushton M., 1983, Nucl. Phys. B220 [FS8] 508

Bilic N. and Gavai R. V., 1984, Z. Phys. C23 77

Bjorken J. D. and Paschos E. A., 1969, Phys. Rev. 185 1975

Bowler K. C., 1983, Three Day In-depth Review on the Impact of Specialized

Processors in Elementary Particle Physics (Padova) 119
Bowler K. C., Chalmers D. L., Kenway A., Kenway R. D., Pawley G. S. and Wallace D. J., 1984, Nucl. Phys. B240 [FS12] 213
Bowler K. C. and Pawley G. S., 1984, Proc. IEEE 72 42
Burkitt A. N., Kenway A. and Kenway R. D., 1983, Phys. Lett. 128B 83
Cabibbo N. and Marinari E., 1982, Phys. Lett. 119B 387
Callaway D. J. E. and Rahman A., 1982, Phys. Rev. Lett. 49 613
Callaway D. J. E. and Rahman A., 1983, Phys. Rev. D28 1506
Carpenter D. B. and Baillie C. F., 1985, Nucl. Phys. B260 103
Carson S. R. and Kenway R. D., 1986, Ann. Phys. 166 364
Casher A., Kogut J. and Susskind L., 1974, Phys. Rev. D10 732
Cleymans J., Gavai R. V. and Suhonen E., 1986, Phys. Rep. 130 217
Coleman S., 1976, Ann. Phys. 101 239
Coleman S., Jackiw R. and Susskind L., 1975, Ann. Phys. 93 267
Creutz M., 1979, Phys. Rev. Lett. 43 553
Creutz M., 1980a, Phys. Rev. Lett. 45 313
Creutz M., 1980b, Phys. Rev. D21 2308
Creutz M., 1983, Quarks, gluons and lattices (Cambridge University Press, Cambridge)
Creutz M., Jacobs L. and Rebbi C., 1979, Phys. Rev. D20 1915
Cullum J. and Willoughby R. A., 1979, in Sparse Matrix Proc. 1978, eds. I. Duff and G. Stewart (SIAM Press)
Dagotto E., Karsch F. and Moreo A., 1986, Phys. Lett. 169B 421
Dagotto E., Moreo A. and Wolff U., 1986, Study of Lattice SU(N) QCD at Finite Baryon Density, Illinois Preprint ILL-TH-86-12
Damgaard P. H., Hochberg D. and Kawamoto N., 1985, Phys. Lett. 158B 239
DEC, 1982, VAX-11 FORTRAN Language Reference Manual (Maynard, Massachusetts)
Drell S. D., Weinstein M. and Yankielowicz S., 1976, Phys. Rev. D14 1627
Duane S., 1985, Nucl. Phys. B257 [FS14] 652
Duncan A. and Furman M., 1981, Nucl. Phys. B190 [FS3] 767
Engels J. and Satz H., 1985, Phys. Lett. 159B 151
Feynman R. P., 1948, Rev. Mod. Phys. 20 367
Feynman R. P., 1972, Photon-Hadron Interaction (Benjamin, Reading, Massachusetts)
Flanders P. M., Hunt D. J., Reddaway S. F. and Parkinson D., 1977, High Speed Computer and Algorithm Organisation (Academic Press, London) 113
Fucito F., Marinari E., Parisi G. and Rebbi C., 1981, Nucl. Phys. B180 [FS2] 369
Gavai R. V., 1985, Phys. Rev. D32 519

Gell-Mann M., 1964, Phys. Lett. 8 214

Gibbs P. E., 1986, Understanding Finite Baryonic Density Simulations in Lattice QCD, Glasgow Preprint 86-389

Glashow S. L., 1961, Nucl. Phys. 22 579

Gross D. and Wilczek F., 1973, Phys. Rev. Lett. 30 1343; Phys. Rev. D8 3633

Guerin F. and Fried H. M., 1986, Phys. Rev. D33 3039

Gupta R. and Patel A., 1983, Phys. Lett. 124B 94; Nucl. Phys. B226 152

Guth A. H., 1980, Phys. Rev. D21 2291

Hamber H. W., 1981, Phys. Rev. D24 951

Hasenfratz A. and Hasenfratz P., 1980, Phys. Lett. 93B 165

Hasenfratz A. and Hasenfratz P., 1981, Phys. Lett. 104B 489

Hasenfratz P. and Karsch F., 1983, Phys. Lett. 125B 308

Haydock R., 1983, Consequences of rounding errors in the recursion and Lanczos methods, Cavendish Preprint

Hockney R. W. and Jesshope C. R., 1981, Parallel Computers (Adam Hilger, Bristol)

Karsten L. H. and Smit J., 1978, Nucl. Phys. B144 536

Karsten L. H. and Smit J., 1979, Phys. Lett. 85B 100

Kawamoto N. and Smit J., 1981, Nucl. Phys. B192 100

Kernighan B. W. and Ritchie D. M., 1978, The C Programming Language (Prentice-Hall, New Jersey)

Kluberg-Stern H., Morel A., Napoly O. and Petersson B., 1983, Nucl. Phys. B220 [FS8] 447

Kogut J., Matsuoka H., Stone M., Wyld H. W., Shenker S., Shigemitsu J. and Sinclair D. K., 1983, Nucl. Phys. B225 [FS9] 93

Kogut J. and Susskind L., 1975, Phys. Rev. D11 395

Kuti J., 1982, Phys. Rev. Lett. 49 183

Lanczos C., 1950, J. Res. Nat. Bur. Stand. 45 255

Lang C. B. and Nicolai H., 1982, Nucl. Phys. B200 [FS4] 135

Langguth W. and Montvay I., 1984, Phys. Lett. 145B 261

Lautrup B. and Nauenberg M., 1980, Phys. Lett. 95B 63

Lowenstein J. and Swieca A., 1971, Ann. Phys. 68 172

Marinari E., Parisi G. and Rebbi C., 1981, Nucl. Phys. B190 [FS3] 734

Matthews P. T. and Salam A., 1954, Nuovo Cim. 12 563

Matthews P. T. and Salam A., 1955, Nuovo Cim. 2 120

Metropolis N., Rosenbluth A. W., Rosenbluth M. N., Teller A. H. and Teller E., 1953, J. Chem. Phys. 21 1087

Montvay I., 1984, Phys. Lett. 139B 70

Nakamura A., 1984, Phys. Lett. 149B 391

Nielsen H. B. and Ninomiya M., 1981, Nucl. Phys. B185 20

Parisi G. and Wu Y.-S., 1981, Sci. Sin. 24 483

Parkinson D., 1983, Comput. Phys. Commun. 28 325

Pass S. D., 1984, The GRID Project, GEC Hirst Research Centre

Pawley G. S. and Thomas G. W., 1982, J. Comp. Phys. 47 165

Politzer H. D., 1973, Phys. Rev. Lett. 30 1346

Polonyi J. and Wyld H. W., 1983, Phys. Rev. Lett. 51 2257

Reddaway S. F., 1973, 1st Annual Symp. on Comput. Arch. (IEEE/ACM) 61

Reddaway S. F., 1979, Infotech State of the Art Report: Supercomputers 2 311

Salam A., 1968, in Elementary Particle Theory, ed. N. Svartholm (Almquist, Forlag AB, Stockholm) 367

Scalapino D. J. and Sugar R. L., 1981, Phys. Rev. Lett. 46 519

Schwinger J., 1962, Phys. Rev. 128 2425

Scott D. S., 1981, in Sparse Matrices and Thier Uses, ed. I. S. Duff (Academic Press, London)

Sharatchandra H. S., Thun H. J. and Weisz P., 1981, Nucl. Phys. B192 205

Stamatescu I. O., 1982, Phys. Rev. D25 1130

Susskind L., 1977, Phys. Rev. D16 3031

Ukawa A. and Fukugita M., 1985, Phys. Rev. Lett. 55 1854

van den Doel C. P., 1984, Phys. Lett. 143B 210

Wallace D. J., 1984, Phys. Rep. 103 191

Weinberg S., 1967, Phys. Rev. Lett. 19 1264

Weingarten D. H. and Petcher D. N., 1981, Phys. Lett. 99B 333

Wilson K. G., 1974, Phys. Rev. D10 2445

Wilson K. G., 1977, in New Phenomena in Subnuclear Physics, ed. A. Zichichi (Plenum, New York) 69

Yang C.-P., 1963, Proc. Symposia in Applied Mathematics, Vol. XV (Amer. Math. Soc., Providence, R. I.) 351

Yang C. N. and Mills R., 1954, Phys. Rev. 96 191