Lattice Quantum Chromodynamics with FLIC Overlap Fermions

Waseem Kamleh

Submitted for the degree of Doctor of Philosophy.

Abstract

Quantum Chromodynamics (QCD) is the theory describing the strong interaction, felt by quarks and mediated by gluons. Due to the strong coupling of the theory at low energies the standard technique for dealing with the electroweak sector, perturbation theory, becomes intractable. One must therefore resort to non-perturbative methods to study the rich low energy structure of QCD.

Lattice Quantum Chromodynamics is the only known means of directly studying QCD non-perturbatively from first principles. By formulating QCD on a discrete hypercube, one can perform numerical simulations to calculate observable expectation values, and additionally perform studies of more fundamental quantities, such as the quark propagator.

Constructing QCD on the lattice is not without its difficulties. Aside from being extremely computationally intensive, maintaining all the relevant symmetries of the continuum theory is non-trivial. Chiral symmetry, and its spontaneous breaking in the QCD vacuum is a key feature of the continuum theory which, among other things, provides an explanation of the relative light mass of the pion and the other pseudo-Goldstone bosons.

The overlap-Dirac operator is a long sought after realisation of chiral symmetry on the lattice. However, it is relatively expensive to evaluate compared to more standard discretisations of the Dirac operator. The overlap is constructed as a function of a non-chiral fermion operator, the overlap kernel, typically taken to be the Wilson-Dirac operator. In this work we construct an overlap operator that is faster to evaluate computationally through the use of an alternative kernel, the Fat Link Irrelevant Clover (FLIC) action.

The properties of FLIC Overlap fermions are investigated through both the quark propagator in momentum space, via topology. These calculations are performed in both the quenched and the dynamical kernel approximation, a type of partial quenching. In the quenched approximation, the effects of fermion vacuum flucuations are neglected in the creation of gluon field configurations. In the partially quenched approximation, a different fermionic action is used in the sea and valence sectors. A means of including the effects of FLIC sea fermions in the gluonic background field is developed. An implementation of code optimised for cluster computing is also presented.

Statement of Originality

This work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text.

I give consent to this copy of my thesis, when deposited in the University Library, being available for loan and photocopying.

Acknowledgements

First and foremost, I would like to sincerely thank Tony Williams and Derek Leinweber for their supervision and support throughout my doctorate. I am additionally grateful to David Adams and Jianbo Zhang for their collaboration, valuable discussions and assistance on numerous occasions. Paul Coddington and Francis Vaughan have also been of great help with trying to squeeze the most juice out of the (occasionally temperamental) compute clusters, Orion and Hydra, that the majority of this work's computation was performed on.

There have been a number of people with whom I have had some very insightful discussions that inspired some of the work contained herein, so I would like to say thanks to Robert Edwards, Urs Heller, Tony Kennedy, Rajamani Narayanan and Herbert Neuberger. Gratitude also goes to Ben L. and James for their lattice spacing code.

Special thanks go to Pat, James, Olivier and Ross for sharing the Ph. D experience (and occasionally a conference hotel room). Finally, I am grateful to my family and friends for their understanding of the exclusion principle that often applies to the doctoral process and spare time.

Contents

1 Introduction **2**
- 1.1 Quantum Chromodynamics 3
- 1.2 Continuum QCD 3
  - 1.2.1 Gauge Invariance 4
  - 1.2.2 Gauge Field Geometry 5
- 1.3 Functional QFT 6
  - 1.3.1 Functional Integration 7
  - 1.3.2 Functional Differentiation 8
  - 1.3.3 Perturbation Theory in Functional QFT 9
  - 1.3.4 The Lattice as a Regulator 11
  - 1.3.5 Lattice Field Theory 12

2 Lattice QCD **14**
- 2.1 Lattice Fundamentals 14
- 2.2 Fermion Fields 15
  - 2.2.1 Fermion doublers 16
  - 2.2.2 Wilson Fermions 17
- 2.3 Gauge Fields 18
  - 2.3.1 Gauge action 20
  - 2.3.2 Lattice Field Strength Tensor 22
- 2.4 Quenched QCD 23
  - 2.4.1 Functional Integrals on the Lattice 24
- 2.5 Systematic Improvements 25

3 Improved Lattice Actions **28**
- 3.1 Mean Field Improvement 28
- 3.2 Improving the Gauge Action 29
- 3.3 Improving the Fermion Action 31
- 3.4 Improving the Field Strength Tensor 32
- 3.5 Smeared Link Actions 33
  - 3.5.1 Cooling and Smearing 34
  - 3.5.2 Fat Link Irrelevant Operator Actions 35
- 3.6 Lattice Symmetries 36

3.6.1 Chiral Symmetry 36
3.6.2 Ginsparg-Wilson Relation 38

4 Overlap Fermions **40**
4.1 The chiral determinant 40
4.2 The overlap-Dirac operator 42
4.3 Properties of the overlap 44
4.3.1 Exceptional Configurations and Topology 44
4.3.2 Locality 48

5 Accelerated Overlap Fermions **52**
5.1 Implementation 52
5.2 Fermion actions 55
5.3 Spectral Flow Comparison 57
5.4 Condition Number Analysis 60
5.5 Spin Projection 63
5.5.1 Standard Spin-Projection Trick 63
5.5.2 Generalised Spin-Projection Trick 64
5.5.3 Clover Trick 65
5.6 Computational Analysis 65
5.7 Further Improvements 67
5.8 Analytic Bounds 71

6 Dynamical FLIC Fermions **76**
6.1 Partial Quenching 76
6.2 Hybrid Monte Carlo 77
6.3 SU(3) Projection 79
6.4 Equations of Motion 81
6.4.1 Mathematical Preliminaries 81
6.4.2 Standard Derivatives 84
6.4.3 Smeared Link Derivatives 85
6.5 Simulation Results 88
6.6 Projection Analysis 90

7 FLIC Overlap Properties **96**
7.1 Quark Propagator in Momentum Space 96
7.1.1 The Overlap Propagator 97
7.1.2 Simulation Details 99
7.1.3 Results 100
7.2 Topology 116

8 Conclusion **120**

A Implementation 124
A.1 Fundamental Modules . . . 126
A.2 Communication Modules . . . 132
A.3 Common Modules . . . 139
A.4 Overlap Code . . . 165
A.5 Quark Propagator Code . . . 173
A.6 Accelerated Ritz Algorithm Code . . . 186
A.7 Hybrid Monte Carlo Code . . . 199

Bibliography 222

Chapter 1

Introduction

The last century has been a remarkable period in the history of physics. The evolution from the classical world to the quantum world has challenged our understanding of nature and has also revolutionised modern society. Quantum physics was proposed in the early 1900's by Einstein and Planck as a theory which resolved the UV catastrophe in the photo-electric effect, and two decades later quantum mechanics would be developed by Schrodinger, de Broglie and Heisenberg. The physicists of the time grappled with the conceptual difficulties presented by this strange new microscopic world, but their successors would overcome these difficulties to unite the theory of special relativity with quantum mechanics to form the foundation of modern physics, quantum field theory (QFT). In turn, the development of QFT would lead to the discovery of quantum gauge field theories. This discovery was a remarkable breakthrough, as the strong and weak nuclear force, along with the electromagnetic force were found to be mediated by gauge bosons, leading to a unified theory of these three forces, the Standard Model.

While the theoretical evolution from quantum mechanics onwards was taking place, there was also a technological evolution unfolding. Soon after the transistor was invented, quantum electrodynamics(QED) was formulated by Feynman, Schwinger and Tomonaga. Around the time that Yang and Mills were studying non-Abelian gauge invariance, quantum mechanics would give us the silicon chip, and the world would never be the same. In 1971 t'Hooft proved that the electroweak QFT of Weinberg, Salam and Glashow was renormalisable, signaling the start of the period when gauge field theories became the foundation of modern physics. In the same year, the first microprocessor was created, itself heralding the start of a new era, one which would see computers become a fundamental part of modern society. The formulators of quantum mechanics in the early part of the last century would never have conceived that this theory would one day lead to the development of the silicon chip and it subsequent technological hegemony. Nonetheless, in this modern age it seems apt that we use the technological fruits of quantum mechanics to study its theoretical flowers.

1.1 Quantum Chromodynamics

Quantum chromodynamics (QCD) is a gauge field theory which is believed to be the theory of strong interactions. It is the theory which primarily describes quarks (or partons), the constituent particles of the nucleon and all hadronic matter. Unlike the electron and the other leptons, quarks possess a colour charge and hence interact with the gauge bosons which mediate the strong interaction, the gluons. The gluons are themselves coloured, as QCD is a non-Abelian gauge theory. This simple fact means that, unlike their electromagnetic counterpart the photon, the gluons interact with themselves, and these self-interactions have some very non-trivial consequences.

Certainly not the least of these is colour confinement, that is, the inability to directly observe quarks in isolation (except at extreme energies). The binding force exerted by the gluons is so strong that we can only see quarks indirectly through the colour neutral composite particles that are comprised of them, the hadrons. Furthermore, the hadrons which compose stable matter, the proton and the neutron, are comprised of up and down quarks, yet the intrinsic mass of these quarks accounts for only three percent of the nucleon mass. The remaining mass is dynamically generated by the complex interactions between the quarks and the gluons, a clear indication that the QCD "vacuum" is anything but empty.

These fundamental properties of QCD that shape the world we see are inherently non-perturbative in nature - they are not present at high energies when the coupling constant α_{strong} becomes small, and QCD becomes perturbative with the onset of asymptotic freedom. Thus perturbation theory, which was so successful when applied to quantum electrodynamics, fails when applied to QCD at low to moderate energies. Various techniques have been developed to deal with the non-perturbative nature of the strong interaction at the energy scales that we are most interested in. However due to complexity of the non-perturbative theory either a model is used, or a truncation of the theory is performed (as in the Dyson-Schwinger equations) in order to study quantities of interest. Unfortunately, this means that no continuum based approach is derived solely from first principles. However, modern computational power made possible by the thoughts of physicists a century ago means that a first principles approach is available, that of Lattice QCD.

1.2 Continuum QCD

We begin by outlining some important concepts from continuum gauge field theories in order to provide a foundation for the formulation of Lattice QCD as well as providing some insight into the motivation of the lattice approach and the work presented in this thesis.

1.2.1 Gauge Invariance

The Lagrangian density for the free fermion field, $\psi(x)$, in Euclidean space is given by (suppressing the spinor and gauge indices as is standard)

$$\mathcal{L}_{\text{free}}(x) = \bar{\psi}(x)(\partial\!\!\!/ + m)\psi(x), \tag{1.1}$$

and is invariant under global gauge (phase) transformations,

$$\psi(x) \to G\psi(x), \;\; \bar{\psi}(x) \to \bar{\psi}(x)G^{\dagger}, \tag{1.2}$$

where $G \in SU(N)$. The principle of local gauge invariance states that $\mathcal{L}(x)$ should be invariant under local gauge transformations,

$$\psi(x) \to G(x)\psi(x), \;\; \bar{\psi}(x) \to \bar{\psi}(x)G^{\dagger}(x). \tag{1.3}$$

This is based upon the idea that the phase (an irrelevant quantity) of a particle should be able to be changed locally without affecting the physics. However, we can see that $\mathcal{L}_{\text{free}}(x)$ is not locally gauge invariant,

$$\mathcal{L}_{\text{free}}(x) \to \mathcal{L}_{\text{free}}{}'(x) = \bar{\psi}(x)(\partial\!\!\!/ + m)\psi(x) + \bar{\psi}(x)G^{\dagger}(x)\partial\!\!\!/ G(x)\psi(x). \tag{1.4}$$

The problem is that ∂_μ now acts non-trivially on $G(x)$. We can ameliorate this problem by using a gauge covariant derivative,

$$D_\mu = \partial_\mu + igA_\mu, \tag{1.5}$$

where we define the gauge field $A_\mu(x)$ to transform as

$$A_\mu(x) \to G(x)(A_\mu(x) + \frac{1}{ig}\partial_\mu)G^{\dagger}(x). \tag{1.6}$$

Then as $G(x)G^{\dagger}(x) = 1$ we have $\partial_\mu G(x)G^{\dagger}(x) = -G(x)\partial_\mu G^{\dagger}(x)$ and hence the covariant derivative transforms as

$$D_\mu\psi(x) \to G(x)D_\mu\psi(x), \tag{1.7}$$

and thus

$$\mathcal{L}(x) = \bar{\psi}(x)(D\!\!\!\!/ + m)\psi(x) \tag{1.8}$$

is locally gauge invariant under $SU(N)$ gauge transformations. In the case of QED, we consider the Abelian group $U(1)$ instead of (non-Abelian) $SU(N)$, and $A_\mu(x)$ is the photon field. For QCD, the gauge group is $SU(3)$, and $A_\mu(x) = A_\mu^a(x)T_a(x)$ represents the sum of the eight gluon fields $A_\mu^a(x)$, and T_a are the eight generators of $SU(3)$. The gluon field is given dynamics by the inclusion of a new term in the Lagrangian,

$$\mathcal{L}_{\text{QCD}} = \bar{\psi}(x)(D\!\!\!\!/ + m)\psi(x) - \frac{1}{2}\,\text{Tr}\, F_{\mu\nu}(x)F^{\mu\nu}(x), \tag{1.9}$$

where $F_{\mu\nu} = \partial_\mu A_\nu - \partial_\nu A_\mu + ig[A_\mu, A_\nu]$ is the antisymmetric field strength tensor, which satisfies $[D_\mu, D_\nu] = igF_{\mu\nu}$.

1.2.2 Gauge Field Geometry

In order to explain some geometrically oriented concepts related to the gauge field, we must first clarify precisely what we mean when we write down $\psi(x)$ and $A_\mu(x)$. With each point in space-time x we associate a vector space $V_x \cong \mathbb{C}^3$ that represents the colour (gauge) degrees of freedom of the fermion field $\psi(x) = \psi^i(x) \in V_x$. $A_\mu(x) = A_\mu^{ij}(x)$ is a matrix or linear operator on V_x. A gauge transformation $G(x)$ performs a change of basis of V_x. A global gauge transformation changes the basis in the same fashion at every point. The principle of local gauge invariance simply states that we can choose a different basis at each x.

As at each point x the colour part of the fermion field lies in a different vector space, one might ask how one can compare the field at two different points.What we wish to do is transport a vector $\psi(x) \in V_x$ to y along some curve $\mathcal{C}_{x\to y}$ to get $\psi(\mathcal{C}_{x\to y}) \in V_y$. We want to do this in a meaningful fashion, such that the inner product $\bar{\psi}(\mathcal{C}_{x\to y})\psi(y)$ is preserved under local gauge (basis) transformations. We clearly cannot simply set $\psi(\mathcal{C}_{x\to y}) = \psi(x)$ as the gauge transformation properties at the two points are different. We must transport $\psi(x)$ in a gauge parallel sense such that $\psi(\mathcal{C}_{x\to y})$ has the same gauge transformation properties as $\psi(y)$. This is the concept of parallel transport, well known to those familiar with differential geometry.

We can see that if we had a matrix $U_\mathcal{C}(y,x)$ that transformed as

$$U_\mathcal{C}(y,x) \to G(y)U_\mathcal{C}(y,x)G^\dagger(x), \tag{1.10}$$

then $\psi(\mathcal{C}_{x\to y}) = U_\mathcal{C}(y,x)\psi(x)$ would transform appropriately. Let us divide $\mathcal{C}_{x\to y}$ into n very small line segments. Consider transporting $\psi(x)$ a small distance to $x+\delta x$ along the first line segment. Now, set

$$U_\mathcal{C}^{(1)}(x+\delta x, x) = e^{igA_\mu(x)\delta x^\mu} = 1 + igA_\mu(x)\delta x^\mu + O(\delta x^2), \tag{1.11}$$

where it is understood that $\delta x^\mu = \delta x^\mu(\mathcal{C})$. We will show that this matrix has the desired transformation properties. Now, note that we can Taylor expand $G^\dagger(x+\delta x)$ to see that

$$G^\dagger(x+\delta x) = G^\dagger(x) + \partial_\mu G^\dagger(x)\delta x^\mu + O(\delta x^2). \tag{1.12}$$

Thus, under the gauge transformation $G(x)$ we have that

$$\begin{aligned}
&U_\mathcal{C}^{(1)}(x+\delta x, x) = 1 + igA_\mu(x)\delta x^\mu + O(\delta x^2)\\
&\to 1 + ig\big(G(x)(A_\mu(x) + \frac{1}{ig}\partial_\mu)G^\dagger(x)\big)\delta x^\mu + O(\delta x^2)\\
&= G(x)\big(1 + igA_\mu(x)\delta x^\mu\big)G^\dagger(x) + G(x)\partial_\mu G^\dagger(x)\delta x^\mu + O(\delta x^2)\\
&= G(x)\big(1 + igA_\mu(x)\delta x^\mu\big)\big(G^\dagger(x) + \partial_\mu G^\dagger(x)\delta x^\mu\big) + O(\delta x^2)\\
&= G(x)\big(1 + igA_\mu(x)\delta x^\mu\big)G^\dagger(x+\delta x) + O(\delta x^2)\\
&= G(x)U_\mathcal{C}^{(1)}(x+\delta x, x)G^\dagger(x+\delta x) + O(\delta x^2).
\end{aligned}$$

Having seen that $U_{\mathcal{C}}^{(1)}(x+\delta x, x)$ as defined above has the desired property, we construct a matrix $U_{\mathcal{C}}^{(n)}(y,x) = \prod_{i=1}^{n} U_{\mathcal{C}}(x_i, x_{i-1})$, where $x_0 = x, x_n = y$ and x_i is given by moving along the ith line segment from the point x_{i-1}, that is, $x_i = x_{i-1} + \delta x_{i-1}$. Now define $U_{\mathcal{C}}(y,x) = \lim_{n\to\infty} U_{\mathcal{C}}^{(n)}(y,x)$,

$$U_{\mathcal{C}}(y,x) = \lim_{n\to\infty} \prod_{i=1}^{n} e^{igA_\mu(x_{i-1})\delta x_{i-1}^\mu} \equiv \mathcal{P} e^{ig\int_{\mathcal{C}} A_\mu(x)dx^\mu}. \tag{1.13}$$

Thus the (gauge) parallel transporter from x to y along the curve $\mathcal{C}$ is simply the path ordered exponential of the gauge field along $\mathcal{C}$, more commonly known as the Wilson line. This matrix is unitary and satisfies $U_{-\mathcal{C}}(x,y) = U_{\mathcal{C}}(y,x)^\dagger$.

Now let us consider an infinitesimal closed loop $\mathcal{C}_{x\to x}$ passing through the intermediate points $x_1 = x+\delta x, x_2 = x+\delta x+\delta y, x_3 = x+\delta y$. Making extensive use of the Taylor expansion $A_\mu(x+\delta x) = A_\mu(x) + \partial_\nu A_\mu(x)\delta x^\nu + O(\delta x^2)$ the parallel transporter around this infinitesimal loop will be given by

$$\begin{aligned}
U_{\mathcal{C}}(x,x) &= U_{\mathcal{C}}(x_1,x)U_{\mathcal{C}}(x_2,x_1)U_{\mathcal{C}}^\dagger(x_2,x_3)U_{\mathcal{C}}^\dagger(x_3,x) \\
&= (1+igA_\mu(x)\delta x^\mu)(1+igA_\nu(x_2)\delta y^\nu)(1-igA_\alpha(x_3)\delta x^\alpha)(1-igA_\beta(x)\delta y^\beta) \\
&\quad + O(\delta x^2, \delta y^2) \\
&= (1+igA_\mu(x)\delta x^\mu + igA_\nu(x_1)\delta y^\nu - g^2A_\mu(x)A_\nu(x_1)\delta x^\mu\delta y^\nu)\times \\
&\quad (1-igA_\alpha(x_3)\delta x^\alpha - igA_\beta(x)\delta y^\beta - g^2A_\alpha(x_3)A_\beta(x)\delta x^\alpha\delta y^\beta) + O(\delta x^2,\delta y^2) \\
&= 1 + ig\{(A_\mu(x) - A_\mu(x_3))\delta x^\mu + (A_\nu(x_1) - A_\nu(x))\delta y^\nu + ig(A_\mu(x)A_\nu(x_1) \\
&\quad - A_\mu(x)A_\nu(x) + A_\mu(x_3)A_\nu(x) - A_\nu(x_1)A_\mu(x_3))\delta x^\mu\delta y^\nu\} + O(\delta x^2,\delta y^2) \\
&= 1 + ig\Big(\partial_\mu A_\nu(x) - \partial_\nu A_\mu(x) + ig[A_\mu(x), A_\nu(x)]\Big)\delta x^\mu\delta y^\nu + O(\delta x^2,\delta y^2) \\
&= 1 + igF_{\mu\nu}(x)\delta x^\mu\delta y^\nu + O(\delta x^2,\delta y^2).
\end{aligned} \tag{1.14}$$

Recognising this, using a similar construction to that we used in deriving the Wilson line, we can see that the parallel transporter around a closed loop $\mathcal{C} = \partial\mathcal{S}$ which is the boundary of some surface $\mathcal{S}$ will be given by

$$U_{\mathcal{C}}(x,x) = \mathcal{P} e^{ig\int_{\mathcal{S}} F\cdot d\mathcal{S}}, \tag{1.15}$$

where we have made use of Stoke's theorem. It is easy to see that $F_{\mu\nu}(x) \to G(x)F_{\mu\nu}(x)G^\dagger(x)$ under a gauge transformation $G(x)$, and $U_{\mathcal{C}}(x,x)$ is similarly covariant. Hence $\mathrm{Tr}\, U_{\mathcal{C}}(x,x)$ is gauge invariant by the cyclic invariance of the trace, and this invariant quantity is known as the Wilson loop.

1.3 Functional QFT

Given the Lagrangian density $\mathcal{L}(x)$ of a quantum field theory, one can describe the physics of the theory in what is known as the functional integral formalism.

For a detailed description of this formalism we refer the reader elsewhere[1], and here only present the concepts relevant to this thesis. In particular, we will not mention any of the subtleties involved when one wishes to differentiate or integrate with respect to anti-commuting Grassmanian fields.

1.3.1 Functional Integration

A functional $F : \mathcal{C}^\infty(\mathbb{R}^4) \to \mathbb{C}$ is a mapping from the space of smooth functions to the complex numbers. Define the action to be the functional

$$S[A, \bar{\psi}, \psi] = \int d^4x \mathcal{L}(x), \tag{1.16}$$

where we consider $\mathcal{L}(x)$ to depend on the gauge fields A_μ and the fermion fields $\bar{\psi}$ and ψ. In Euclidean space, the generating functional is given by the following functional integral,

$$\mathcal{Z} = \int \mathcal{D}A\mathcal{D}\bar{\psi}\mathcal{D}\psi \; e^{-S[A,\bar{\psi},\psi]}. \tag{1.17}$$

The primary quantity of interest, the expectation value of some observable $\mathcal{O}$ is given by

$$\langle \mathcal{O} \rangle = \frac{1}{\mathcal{Z}} \int \mathcal{D}A\mathcal{D}\bar{\psi}\mathcal{D}\psi \; \mathcal{O}[A, \bar{\psi}, \psi] e^{-S[A,\bar{\psi},\psi]}. \tag{1.18}$$

In order to specify precisely what we mean by integrating a functional $F[f]$ over function space, we must define the measure $\mathcal{D}f$. Defining this measure is somewhat problematic, as function space is extremely large, in fact, an infinite-dimensional vector space. We define $\int \mathcal{D}f$ by considering the limit of the equivalent integral over a finite number of degrees of freedom.

Starting with Euclidean spacetime $\mathbb{R}^4$, define $\mathbb{H}^4 \subset \mathbb{R}^4$ to be a hypercube with sides of length l. Partition $\mathbb{H}^4$ into a hybercubic lattice of points $\mathbb{L} \subset a\mathbb{Z}^4$ with lattice spacing a and $N+1$ points on a side, where $N = \frac{l}{a}$. So, if x_0 denotes the origin of $\mathbb{H}^4$ then

$$\mathbb{L} = \{x_0 + an | n \in \mathbb{Z}^4, 0 \leq n_\mu \leq N\}. \tag{1.19}$$

Given a function $f : \mathbb{R}^4 \to \mathbb{C}$, define a function $f|_\mathbb{L} : \mathbb{L} \to \mathbb{C}$ by simply considering the restriction of f to $\mathbb{L}$. Now, $\mathbb{L}$ contains $(N+1)^4$ points, so we can completely specify any such function $f|_\mathbb{L}$ by specifying its value at each $x \in \mathbb{L}$. So the function space over $\mathbb{L}$ is simply a finite-dimensional complex vector space which we denote $\mathcal{V}^\mathbb{L}(\mathbb{C})$, and each function $f|_\mathbb{L}$ has a corresponding vector $\mathbf{f} \in \mathcal{V}^\mathbb{L}(\mathbb{C})$ with components $\mathbf{f}_x \equiv f(x), x \in \mathbb{L}$. Given some functional $F : \mathcal{C}^\infty(\mathbb{R}^4) \to \mathbb{C}$, let us assume we can write down a corresponding function $F|_\mathbb{L} : \mathcal{V}^\mathbb{L}(\mathbb{C}) \to \mathbb{C}$. Then we define the functional integral of $F[f]$ with respect to f to be

$$\int \mathcal{D}f \; F[f] = \lim_{l\to\infty} \lim_{a\to 0} \int \prod_{x\in\mathbb{L}} d\mathbf{f}_x \; F|_\mathbb{L}(\mathbf{f}), \tag{1.20}$$

where it is understood that when taking $\lim_{a\to 0}$ we hold $l = aN$ fixed.

1.3.2 Functional Differentiation

Given a functional $F[f]$ we define $\frac{\delta F}{\delta f}$, the functional derivative of F with respect to f to be the function defined by

$$\int d^4x\ \phi(x)\frac{\delta F}{\delta f(x)} = \lim_{a\to 0}\frac{1}{a}\big(F[f+a\phi] - F[f]\big), \tag{1.21}$$

where $\phi(x)$ is some (arbitrary) smooth test function. Naturally, we require that the limit defined above to be independent of the choice of ϕ in order for F to be differentiable. It is then straightforward to show that

$$F[f] = e^{\int d^4x\ f(x)g(x)} \Rightarrow \frac{\delta F}{\delta f(x)} = g(x)F[f], \tag{1.22}$$

and one can see that the functional derivative behaves in the expected fashion.

At this point we wish to note that we can write the gluonic term in the QCD Lagrangian in terms of the eight individual gluon fields $A^a_\mu(x)$. First, note that T_a, the generators of $SU(3)$ satisfy

$$\mathrm{Tr}\, T_a T_b = \frac{1}{2}\delta_{ab}, \tag{1.23}$$

$$[T_a, T_b] = i f_{abc} T_c, \tag{1.24}$$

where f_{abc} are the structure constants for $SU(3)$, which are antisymmetric. It is then straightforward that

$$\frac{1}{2}\,\mathrm{Tr}\, F_{\mu\nu}(x)F^{\mu\nu}(x) = \frac{1}{4}F^a_{\mu\nu}(x)F^a_{\mu\nu}(x), \tag{1.25}$$

where $F_{\mu\nu}(x) = F^a_{\mu\nu}(x)T_a$, and

$$F^a_{\mu\nu}(x) = \partial_\mu A^a_\nu(x) - \partial_\nu A^a_\mu(x) - g f_{abc} A^b_\mu(x) A^c_\nu(x). \tag{1.26}$$

Now, in the presence of a gluonic current $\mathcal{J}^\mu_a(x)$ and fermionic sources $\bar\chi(x), \chi(x)$, we may write the generating functional for QCD as

$$\mathcal{Z}[\mathcal{J}^\mu, \bar\chi, \chi] = \int \mathcal{D}A\mathcal{D}\bar\psi\mathcal{D}\psi\ e^{\int d^4x - \mathcal{L}(x) + A^a_\mu(x)\mathcal{J}^\mu_a(x) + \bar\chi(x)\psi(x) + \bar\psi(x)\chi(x)}. \tag{1.27}$$

Then it is clear that, for example, the two-point function can be obtained by differentiating the generating functional,

$$\langle\psi(x_1)\bar\psi(x_2)\rangle = \frac{1}{\mathcal{Z}[0]}\left(\frac{\delta}{\delta\bar\chi(x_1)}\right)\left(-\frac{\delta}{\delta\chi(x_2)}\right)\mathcal{Z}[\bar\chi,\chi]\Big|_{\bar\chi,\chi=0}. \tag{1.28}$$

Higher order n-point functions may be similarly constructed by inserting more derivatives.

1.3.3 Perturbation Theory in Functional QFT

Decomposing the Lagrangian for QCD into the following form

$$\mathcal{L}_{\text{QCD}}(x) = \mathcal{L}^{(0)}_{\text{quark}}(x) + \mathcal{L}^{(0)}_{\text{gluon}}(x) + \mathcal{L}_I(x), \tag{1.29}$$

we can now distinguish between the kinetic terms

$$\mathcal{L}^{(0)}_{\text{quark}}(x) = \bar{\psi}(x)(\partial\!\!\!/ + m)\psi(x), \tag{1.30}$$

$$\mathcal{L}^{(0)}_{\text{gluon}}(x) = -\frac{1}{4}(\partial_\mu A^a_\nu(x) - \partial_\nu A^a_\mu(x))(\partial_\mu A^a_\nu(x) - \partial_\nu A^a_\mu(x)), \tag{1.31}$$

and the interacting terms

$$\mathcal{L}_I(x) = \mathcal{L}_{\bar{\text{q}}\text{gq}}(x) + \mathcal{L}_{\text{ggg}}(x) + \mathcal{L}_{\text{gggg}}(x), \tag{1.32}$$

$$\mathcal{L}_{\bar{\text{q}}\text{qg}}(x) = ig\bar{\psi}(x)A\!\!\!/^a(x)T_a\psi(x), \tag{1.33}$$

$$\mathcal{L}_{\text{ggg}}(x) = \frac{1}{2}gf_{abc}\partial_\mu A^a_\nu(x)A^b_\mu(x)A^c_\nu(x), \tag{1.34}$$

$$\mathcal{L}_{\text{gggg}}(x) = -\frac{1}{4}g^2 f_{abc}f_{ade}A^b_\mu(x)A^c_\nu(x)A^d_\mu(x)A^e_\nu(x). \tag{1.35}$$

Now, define the following operators

$$\mathcal{V}_{\bar{\text{q}}\text{qg}}(x) = -ig\gamma^\mu T_a \frac{\delta}{\delta\chi(x)}\frac{\delta}{\delta\mathcal{J}^\mu_a(x)}\frac{\delta}{\delta\bar{\chi}(x)}, \tag{1.36}$$

$$\mathcal{V}_{\text{ggg}}(x) = \frac{1}{2}gf_{abc}\partial_\mu \frac{\delta}{\delta\mathcal{J}^\nu_a(x)}\frac{\delta}{\delta\mathcal{J}^\mu_b(x)}\frac{\delta}{\delta\mathcal{J}^\nu_c(x)}, \tag{1.37}$$

$$\mathcal{V}_{\text{gggg}}(x) = -\frac{1}{4}g^2 f_{abc}f_{ade}\frac{\delta}{\delta\mathcal{J}^\mu_b(x)}\frac{\delta}{\delta\mathcal{J}^\nu_c(x)}\frac{\delta}{\delta\mathcal{J}^\mu_d(x)}\frac{\delta}{\delta\mathcal{J}^\nu_e(x)} \tag{1.38}$$

and observe that for each s = q̄gq, ggg, gggg we have

$$\mathcal{V}_{\text{s}}(x)\mathcal{Z}[\mathcal{J}^\mu, \bar{\chi}, \chi] = \mathcal{L}_{\text{s}}(x)\mathcal{Z}[\mathcal{J}^\mu, \bar{\chi}, \chi]. \tag{1.39}$$

Using the fact that these operators do not depend upon $A_\mu, \bar{\psi}, \psi$ the generating functional may be rewritten in the following form

$$\mathcal{Z}[\mathcal{J}^\mu, \bar{\chi}, \chi] = e^{-V_I}\int \mathcal{D}A\mathcal{D}\bar{\psi}\mathcal{D}\psi\ e^{-S_Q}e^{-S_G}, \tag{1.40}$$

where we have set

$$V_I = \int d^4x\ \mathcal{V}_{\bar{\text{q}}\text{qg}}(x) + \mathcal{V}_{\text{ggg}}(x) + \mathcal{V}_{\text{gggg}}(x), \tag{1.41}$$

$$S_Q = \int d^4x\ \mathcal{L}^{(0)}_{\text{quark}}(x) - \bar{\chi}(x)\psi(x) - \bar{\psi}(x)\chi(x), \tag{1.42}$$

$$S_G = \int d^4x\ \mathcal{L}^{(0)}_{\text{gluon}}(x) - A_\mu(x)\mathcal{J}^\mu(x). \tag{1.43}$$

Define $\Delta_{\text{f}}(x,y)$ to be the fermionic Green's function

$$(\partial\!\!\!/ + m)\Delta_{\text{f}}(x,y) = \delta^4(x-y). \tag{1.44}$$

The functional integral is invariant under translations in function space, so if we make the change of variables

$$\psi(x) \to \psi(x) + \int d^4y\ \Delta_{\text{f}}(x,y)\chi(y), \tag{1.45}$$

$$\bar{\psi}(x) \to \bar{\psi}(x) + \int d^4y\ \bar{\chi}(y)\Delta_{\text{f}}(y,x), \tag{1.46}$$

then one can show that $\int d^4x\ \mathcal{L}^{(0)}_{\text{quark}}(x)$ will become

$$\int d^4x\ \bar{\psi}(x)(\partial\!\!\!/+m)\psi(x)+\bar{\psi}(x)\chi(x)+\bar{\chi}(x)\psi(x)+\int d^4y\bar{\chi}(x)\Delta_{\text{f}}(x,y)\chi(y). \tag{1.47}$$

Thus, setting $M(x,y) = \delta^4(x-y)(\partial\!\!\!/ + m)$ we can write

$$S_Q = \int d^4x \int d^4y\ \bar{\psi}(x)M(x,y)\psi(y) + \bar{\chi}(x)\Delta_{\text{f}}(x,y)\chi(y). \tag{1.48}$$

In order to do the equivalent for S_G we must derive the appropriate Green's function for the gluons. First, using integration by parts we observe that

$$\int d^4x\ \partial_\mu A^a_\nu(x)\partial_\alpha A^a_\beta(x) = -\int d^4x\ A^a_\nu(x)\partial_\mu\partial_\alpha A^a_\nu(x), \tag{1.49}$$

and hence

$$\int d^4x\ \mathcal{L}^{(0)}_{\text{gluon}}(x) = \frac{1}{2}\int d^4x\ A^a_\nu(x)(\delta^{\mu\nu}\partial\cdot\partial - \partial^\mu\partial^\nu)A^a_\mu(x). \tag{1.50}$$

In this form we can now see that if we define $\Delta_{\mu\nu}(x,y)$ to be the gauge bosonic Green's function

$$(\delta^{\mu\nu}\partial\cdot\partial - \partial^\mu\partial^\nu)\Delta_{\mu\nu}(x,y) = \delta^4(x-y), \tag{1.51}$$

then one can transform the gluonic fields according to

$$A^a_\mu(x) \to A^a_\mu(x) + \int d^4y\ \Delta_{\mu\nu}(x,y)\mathcal{J}^\nu_a(x), \tag{1.52}$$

in order to obtain that

$$S_G = \int d^4x \int d^4y\ A^a_\mu(x)D^{\mu\nu}(x,y)A^a_\nu(y) + \mathcal{J}^\mu_a(x)\Delta_{\mu\nu}(x,y)\mathcal{J}^\nu_a(y), \tag{1.53}$$

where $D^{\mu\nu}(x,y) = \delta^4(x-y)(\delta^{\mu\nu}\partial\cdot\partial - \partial^\mu\partial^\nu)$. We now make use if our expressions for Q and G, along with the following standard result, which states that given linear operators $B^{\mu\nu}(x,y)$ and $C(x,y)$ we have that

$$\prod_{a=1}^{8} \frac{1}{\det B^{\frac{1}{2}}} = \int \mathcal{D}A \; e^{-\frac{1}{2}\int d^4x \int d^4y \; A_\mu^a(x) B^{\mu\nu}(x,y) A_\nu^a(y)}, \tag{1.54}$$

$$\det C = \int \mathcal{D}\bar{\psi}\mathcal{D}\psi \; e^{-\int d^4x \int d^4y \; \bar{\psi}(x) C(x,y) \psi(y)}, \tag{1.55}$$

to write the generating functional as[1]

$$\mathcal{Z}[\mathcal{J}_a^\mu, \bar{\chi}, \chi] = \det D^{-4} \det M e^{-V_I} e^{-\int d^4x \int d^4y \; \bar{\chi}(x)\Delta_{\mathrm{f}}(x,y)\chi(y) + \mathcal{J}_a^\mu(x)\Delta_{\mu\nu}(x,y)\mathcal{J}_a^\nu(y)}. \tag{1.56}$$

Now, observing that in momentum space $\Delta_{\mathrm{f}}(x,y)$ is the bare quark propagator and $\Delta_{\mu\nu}(x,y)$ is the bare gluon propagator, and hence that $\mathcal{V}_{\bar{\mathrm{q}}\mathrm{qg}}$, $\mathcal{V}_{\mathrm{ggg}}$ and $\mathcal{V}_{\mathrm{gggg}}$ will correspond to the quark-gluon vertex, the 3-gluon vertex and the 4-gluon vertex respectively, we can construct a set of Feynman rules based upon the above form of the generating functional. Then, by expanding the exponential e^{-V_I} we can obtain a power series in g and hence a perturbative expansion for vacuum expectation values. For example, the perturbation series for the two-point function would be

$$\langle \psi(x_1)\bar{\psi}(x_2)\rangle = \frac{1}{\mathcal{Z}[0]} \det D^{-1} \det M \left(-\frac{\delta}{\delta\bar{\chi}(x_1)} \frac{\delta}{\delta\chi(x_2)} \right) (1 - V_I + \frac{1}{2}V_I^2 + \ldots) \\ \times e^{-\int d^4x \int d^4y \; \bar{\chi}(x)\Delta_{\mathrm{f}}(x,y)\chi(y) + \mathcal{J}_a^\mu(x)\Delta_{\mu\nu}(x,y)\mathcal{J}_a^\nu(y)} \Big|_{\mathcal{J},\bar{\chi},\chi=0}. \tag{1.57}$$

Using the correspondences stated above for the vertices and propagators, we could then construct a set of Feynman diagrams to represent the above expansion, and this would be equivalent to the standard set of diagrams derived from the usual perturbation theory, shown below.

1.3.4 The Lattice as a Regulator

In perturbation theory, loop diagrams involve an unconstrained momentum integral which is divergent in four dimensions. This problem is due to the fact

[1]Strictly speaking, in its gauge invariant form, $\det D = 0$ and hence is not invertible. This is due to the integration $\int \mathcal{D}A$ over all gauge fields, whereas what is desired is an integration over physically inequivalent gauge fields, that is, those that are not related by a gauge transformation. This is achieved by the process of gauge fixing, which will not be discussed here. Suffice it to say, after fixing the gauge $\det D^{-1}$ becomes well-defined (neglecting Gribov copies).

that perturbation theory is formulated in terms of bare parameters, and the (renormalised) physical theory is not plagued with these infinities. In order to perform a calculation using perturbation theory, one must regularise the theory in some fashion. There are many choices of regulator, two examples being a simple momentum cutoff and dimensional regularisation. When regulating a theory, often one cannot maintain all the symmetries of the theory, for example introducing a momentum cutoff violates translational invariance.

We have seen that in order to provide a definition of what we mean by functional integration, we discretised space-time into a lattice and considered the limit of a standard multi-dimensional integral. This procedure is suggestive of an alternative means of providing a regulator. We will see in the next chapter that the introduction of a minimum distance scale, the lattice spacing a, has the effect of introducing a maximum available momentum, $p = \frac{\pi}{a}$. Thus, the loop integrals formulated on the lattice are convergent, and one could then take $a \to 0$ to recover the desired result.

If one only considers doing perturbation theory, then one may consider discretising all of space-time to evaluate a loop integral somewhat drastic when there are simpler alternatives available. However, the strength of the lattice formulation does not lie in perturbation theory. The strength of the lattice formalism lies in the fact that it is not simply a mathematical nicety that defines the functional integral formalism. Rather, the power of the lattice regularisation is such that it renders the desired functional integrals directly calculable. That is, we can formulate our QFT on a finite lattice, calculate the desired observable defined by equation (1.18) using numerical techniques, and then derive the physical result by taking the infinite-volume and continuum ($a \to 0$) limits. This is the underlying principle behind Lattice QCD.

Furthermore, as we are directly calculating expectation values rather than truncating a perturbative expansion, it is possible to use Lattice QCD to explore the low-energy content of QCD, where the coupling constant g is large and perturbation theory fails. Thus in QCD we are in the desirable position of having two complementary powerful tools at our disposal. At high energies where Lattice QCD is least effective the coupling constant is small and the theory is tractable using perturbation theory. In the low to moderate energy region g is large and QCD no longer yields to the perturbative method, but it is in this region that Lattice QCD enters into its own, allowing non-perturbative calculations based solely on first principles. This makes Lattice QCD an exciting field as it is in this region that we live in, this region where the physics of QCD is richest and hence this region that many physicists are most interested in.

1.3.5 Lattice Field Theory

We have seen that the functional integral formulation of QCD gives us a non-perturbative tool to analyse our theory, through the lattice regularisation. By

formulating QCD on the lattice and calculating the physical content of the lattice field theory we are then able to deduce content of the theory in the real world by taking the continuum limit.

What we will see in the chapters to come is that the process of formulating QCD on the lattice is not a trivial one. In particular, as in other regularisation schemes, difficulties arise when we try to maintain all the symmetries that QCD possesses in the continuum. Chiral symmetry in particular will prove extremely problematic to maintain. Furthermore, we will also see that performing "ideal" numerical simulations within Lattice QCD is well beyond current computational power, and in particular increase exponentially when one wishes to simulate at light quark masses and small lattice spacings.

To date, various approximations have been made to reduce the numerical cost of Lattice QCD, but these approximations all have the effect of removing the lattice formulation of QCD further and further from its continuum counterpart. This in turn reduces the amount of QCD that we can explore through the lattice. It is the purpose of this thesis to try and overcome these analytic and numerical obstructions by improving the formulation of Lattice QCD in such a way as to bring it as close to the continuum formulation as possible and yet at the same time allow simulations to be performed using current computing power.

Chapter 2

Lattice QCD

In the previous chapter we saw that via the functional integral formalism one could non-perturbatively regularise QCD by introducing a space-time lattice. The lattice was constructed through the introduction of a minimum distance scale, the lattice spacing a. Our task then is to formulate QCD on the lattice such that when we take $a \to 0$ we recover standard QCD. As a starting point, we will restrict continuum QCD to the lattice in a straightforward fashion. Almost immediately we will see that this naive approach is afflicted with severe problems. Successive improvements will one by one ameliorate these problems, until we arrive at an essentially ideal formulation of lattice QCD, which will bear little resemblance to our initial guess.

2.1 Lattice Fundamentals

Given a lattice spacing a, define the set of available space-time points to be restricted to the hybercubic lattice,

$$\mathbb{L} \subset a\mathbb{Z}^4 = \{x | x^\mu = an^\mu, n \in \mathbb{Z}^4\}. \tag{2.1}$$

If we have a finite lattice we usually introduce periodic boundary conditions, which corresponds to formulating our theory on the 4-torus.

It is not possible to consider infinitesimal distances on the lattice. This means that when restricting our continuum theory to the lattice we must replace derivatives by finite difference operators, and integrals with sums. As might be expected when we introduce a minimum distance a, the corresponding generator of translations, momentum is also affected. Each component of 4-momentum is now restricted to the Brillouin zone, $p_\mu \in (-\frac{\pi}{a}, \frac{\pi}{a}]$.

Example 2.1.1. *Show this.*

Proof. We consider the momentum space representation of $f(x), x \in \mathbb{L}$,

$$\tilde{f}(p) = \int d^4x \; e^{-ip\cdot x} f(x) \simeq a^4 \sum_n e^{-iap\cdot n} f(an).$$

Then componentwise (no sum over μ), we have

$$e^{-ia(p_\mu+\frac{2\pi}{a})n^\mu} = e^{-iap_\mu n^\mu}e^{-2\pi i n^\mu} = e^{-iap_\mu n^\mu}.$$

Clearly $\tilde{f}(p)$ is a periodic function in p_μ with period $2\pi/a$, and hence without loss of generality we can restrict $p_\mu \in (-\frac{\pi}{a}, \frac{\pi}{a}]$, as required. □

The final point we consider before we provide the details of formulating QCD on the lattice is that we are restricted in the choice of operators that we can use. In order for an operator to be valid, its action on a function evaluated at at a lattice point must only depend upon the function values at lattice points. Later we will see that this also limits the available symmetry groups on the lattice, but at this point let us commence our construction of Lattice QCD.

2.2 Fermion Fields

We begin by considering free fermions on the lattice. This is done in a straightforward manner, by simply considering the fermionic fields $\psi(x), \bar{\psi}(x)$ as before, but now with the understanding that the space-time points x are restricted to lattice sites, $x \in \mathbb{L}$. Let us define the transport operators $T_\mu, T_\mu^\dagger$ that take us from site x to the site one step forward or backward in the μ direction, $x \pm ae_\mu$, by

$$(T_\mu\psi)(x) = \psi(x + ae_\mu), \ (T_\mu^\dagger\psi)(x) = \psi(x - ae_\mu). \tag{2.2}$$

Now, define the central finite difference operator as

$$\nabla_\mu = \frac{1}{2a}(T_\mu - T_\mu^\dagger). \tag{2.3}$$

This operator is obviously anti-Hermitian, and in the continuum limit corresponds to the partial derivative operator, ∂_μ (provided it is acting on continuously differentiable functions).

Example 2.2.1. *Verify this.*

Proof.

$$\begin{aligned}
\lim_{a\to 0}(\nabla_\mu\psi)(x) &= \lim_{a\to 0}\frac{1}{2a}\big((T_\mu\psi)(x) - (T_\mu^\dagger\psi)(x)\big) \\
&= \lim_{a\to 0}\frac{1}{2a}\big(\psi(x+ae_\mu) - \psi(x-ae_\mu)\big) \\
&= \lim_{a\to 0}\frac{1}{2}\Big(\frac{\psi(x+ae_\mu)-\psi(x)}{a} + \frac{\psi(x)-\psi(x-ae_\mu)}{a}\Big) \\
&= \frac{1}{2}\big(\partial_\mu\psi(x) + \partial_\mu\psi(x)\big) = \partial_\mu\psi(x). \quad \square
\end{aligned}$$

Now we can write down the Lagrangian density for free, massless fermions on the lattice (noting that the Dirac matrices $\gamma_\mu, \mu = 1\dots4$ are unchanged from the continuum, that is they are Hermitian, unitary and anti-commute),

$$\mathcal{L}_{\text{free}}(x) = \bar{\psi}(x)\nabla\!\!\!\!/\,\psi(x). \tag{2.4}$$

Having written down $\mathcal{L}_{\text{free}}$, we have already come across our first difficulty in formulating QCD on the lattice. A quick glance at Example 2.2 should convince you that ∇_μ only couples sites that are separated by $2a$. This has a very serious consequence, the infamous *fermion doubling problem.*

2.2.1 Fermion doublers

In one dimension, coupling sites spaced by $2a$ means that even sites are coupled only to even sites, and odd to odd. This is equivalent to having two lattice fermion fields ψ_{even} and ψ_{odd}. Unfortunately, this situation is not ameliorated by taking the continuum limit, and we find that while we discretised a theory with only one fermion species, when we extrapolate back to the continuum our results are contaminated by the additional fermions. This unwanted extra species is known as a fermion doubler, because the situation only arises when discretising a first-order derivative (and therefore is not a problem with bosons, as they obey the Klein-Gordon equation).

In d dimensions, we find that sites x and y are coupled only if

$$x_\mu - y_\mu = 0 \mod 2 \;\forall\; \mu = 1, 2, \dots d. \tag{2.5}$$

This yields 2^d fermion species, that is $2^d - 1$ doublers. To provide some physical insight into this problem, and its eventual solution, we consider the problem in momentum space. The momentum space representation of ∂_μ is ip_μ, and this function has only one zero, at $p_\mu = 0$. However, in momentum space the lattice finite difference $\nabla_\mu \to \frac{i}{a}\sin(ap_\mu)$.

Example 2.2.2. *Verify this.*

$$\begin{aligned}
(\widetilde{\nabla_\mu\psi})(p) &\equiv \int d^4x\; e^{-ip\cdot x}(\nabla_\mu\psi)(x) \\
&= \frac{1}{2a}\int d^4x\; e^{-ip\cdot x}\big(\psi(x+ae_\mu) - \psi(x-ae_\mu)\big) \\
&= \frac{1}{2a}\int d^4x\; e^{-ip\cdot x}\psi(x+ae_\mu) - \frac{1}{2a}\int d^4x\; e^{-ip\cdot x}\psi(x-ae_\mu) \\
&= \frac{1}{2a}\int d^4x\; e^{-ip\cdot(x-ae_\mu)}\psi(x) - \frac{1}{2a}\int d^4x\; e^{-ip\cdot(x+ae_\mu)}\psi(x) \\
&= \frac{1}{2a}\int d^4x\; (e^{ip\cdot ae_\mu} - e^{-ip\cdot ae_\mu})e^{-ip\cdot x}\psi(x) \\
&= \frac{1}{2a}(e^{iap_\mu} - e^{-iap_\mu})\int d^4x\; e^{-ip\cdot x}\psi(x)
\end{aligned}$$

$$= \frac{i}{a}\sin(ap_\mu)\tilde{\psi}(p) \quad \square$$

Now $\sin(ap_\mu)$ also has a zero at $p_\mu = 0$, but it has additional zeros in the Brillouin zone, fifteen of them in fact, at $p_\mu = \frac{1}{a}(\pi, 0, 0, 0), \frac{1}{a}(\pi, \pi, 0, 0)$ and so on. These extra zeros give rise to the unwanted doublers.

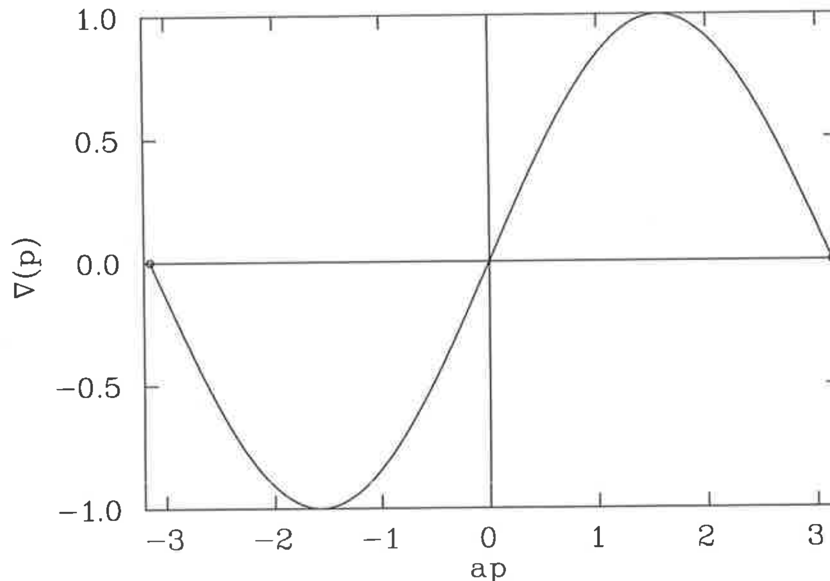

Figure 2.1: The structure of ∇ in (1D) momentum space.

2.2.2 Wilson Fermions

Fortunately, Wilson[2] saw a solution to this. When writing down our formulation of lattice QCD, we can modify it to our liking, so long as we recover the correct theory when taking the continuum limit. In particular this means that we are allowed to add irrelevant dimension five and higher operators to the Lagrangian as they will disappear as $a \to 0$. Now, define the lattice Laplacian operator as

$$\Delta = \frac{1}{a^2}\sum_{\mu=1}^{4} 2 - T_\mu - T_\mu^\dagger. \tag{2.6}$$

It is a simple exercise to see that $\lim_{a\to 0}\Delta = \partial\cdot\partial$, but more importantly, we can see that Δ couples sites that are only one lattice spacing apart. Alternatively, in momentum space $\Delta \to \frac{2}{a^2}\sum_\mu(1 - \cos(ap_\mu))$, which clearly only has a zero at $p_\mu = 0$ (see Fig. 2.2). So we can remove the fermion doublers by including a term in the Lagrangian proportional to Δ, and we refer to this term as the Wilson term. We can also incorporate a mass term m to arrive at the Wilson

action,

$$\mathcal{L}_{\text{Wilson}}(x) = \bar{\psi}(x)(\nabla\!\!\!/ + \frac{ra}{2}\Delta + m)\psi(x). \tag{2.7}$$

The Wilson coefficient r is canonically set to unity.

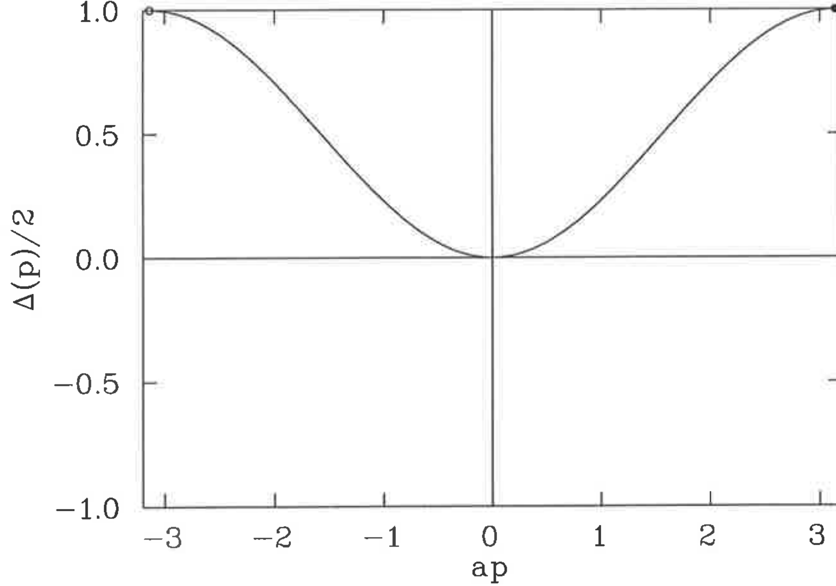

Figure 2.2: The structure of $\frac{1}{2}\Delta$ in (1D) momentum space.

2.3 Gauge Fields

Transcribing gauge fields to the lattice is not quite as straightforward as it was with fermion fields. Recall that in the continuum, the introduction of the gauge field A_μ was necessary to satisfy local gauge invariance. This was done by modifying ∂_μ to include the gauge field and become the covariant derivative $D_\mu = \partial_\mu + igA_\mu$. Now, observe that (in the continuum) we can write

$$T_\mu = e^{a\partial_\mu}, \; T^\dagger_\mu = e^{-a\partial_\mu}. \tag{2.8}$$

Example 2.3.1. *Verify this.*

$$\begin{aligned}
(T_\mu\psi)(x) &= \psi(x + ae_\mu) \\
&= \sum_n \frac{1}{n!}||(x + ae_\mu) - x||^n \partial^n_\mu \psi(x) \\
&= \sum_n \frac{1}{n!} a^n \partial^n_\mu \psi(x) \\
&= e^{a\partial_\mu}\psi(x) \quad \square
\end{aligned}$$

Let us redefine the continuum version of the transport operators by performing the same modification to the partial derivative,

$$T_\mu \equiv e^{aD_\mu}, \; T_\mu^\dagger = e^{-aD_\mu}. \tag{2.9}$$

Now Trotter's formula states that for operators A, B we have that

$$e^{a(A+B)} = \lim_{n\to\infty} (e^{\frac{a}{n}A} e^{\frac{a}{n}B})^n. \tag{2.10}$$

Noting that for a pointwise operator $A(x)$ we have that $A(x+ae_\mu) = e^{a\partial_\mu} A(x) e^{-a\partial_\mu}$, we can now see that [1]

$$\begin{aligned}
(T_\mu\psi)(x) &\equiv e^{aD_\mu}\psi(x) = e^{a(\partial_\mu + igA_\mu(x))}\psi(x) \\
&= \lim_{n\to\infty} (e^{\frac{a}{n}\partial_\mu} e^{\frac{a}{n}igA_\mu(x)})^n \psi(x) \\
&= \lim_{n\to\infty} e^{\frac{a}{n}\partial_\mu} e^{\frac{a}{n}igA_\mu(x)} e^{-\frac{a}{n}\partial_\mu} e^{\frac{a}{n}\partial_\mu} (e^{\frac{a}{n}\partial_\mu} e^{\frac{a}{n}igA_\mu(x)})^{n-1}\psi(x) \\
&= \lim_{n\to\infty} e^{\frac{a}{n}igA_\mu(x+\frac{a}{n}e_\mu)} e^{\frac{a}{n}\partial_\mu} (e^{\frac{a}{n}\partial_\mu} e^{\frac{a}{n}igA_\mu(x)})^{n-1}\psi(x) \\
&= \lim_{n\to\infty} e^{\frac{a}{n}igA_\mu(x+\frac{a}{n}e_\mu)} e^{\frac{2a}{n}\partial_\mu} e^{\frac{a}{n}igA_\mu(x)} e^{-\frac{2a}{n}\partial_\mu} e^{\frac{2a}{n}\partial_\mu} (e^{\frac{a}{n}\partial_\mu} e^{\frac{a}{n}igA_\mu(x)})^{n-2}\psi(x) \\
&= \lim_{n\to\infty} e^{\frac{a}{n}igA_\mu(x+\frac{a}{n}e_\mu)} e^{\frac{a}{n}igA_\mu(x+\frac{2a}{n}e_\mu)} e^{\frac{2a}{n}\partial_\mu} (e^{\frac{a}{n}\partial_\mu} e^{\frac{a}{n}igA_\mu(x)})^{n-2}\psi(x) \\
&\;\;\vdots \\
&= \lim_{n\to\infty} [\prod_{j=1}^{n} e^{\frac{a}{n}igA_\mu(x+\frac{ja}{n}e_\mu)}] (e^{\frac{a}{n}\partial_\mu})^n \psi(x) \\
&= \mathcal{P} e^{ig\int_x^{x+ae_\mu} dx_\mu A_\mu(x)} \psi(x+ae_\mu).
\end{aligned} \tag{2.11}$$

So we have shown that by replacing ∂_μ by D_μ in the definition of T_μ is equivalent to setting $T_\mu(x) = U_\mu(x)e^{a\partial_\mu}$, where $U_\mu(x) \equiv \mathcal{P}e^{ig\int_x^{x a e_\mu} dx_\mu A_\mu(x)}$ is the path-ordered exponential of the integral of the gauge field from x to $x + ae_\mu$. But this is just the Wilson line from x to $x + ae_\mu$, which means that T_μ is simply the (gauge) parallel transporter of the fermion field in the forward μ direction.

As the $A_\mu(x)$ field is Hermitian and the sum of generators of $SU(3)$, we can see that the matrices $U_\mu(x) \in SU(3)$. Thus in order to put our gauge fields on the lattice, without any reference to integrals, we can simply define our transporters to be the unitary operators

$$(T_\mu\psi)(x) \equiv U_\mu(x)\psi(x+ae_\mu), \; (T_\mu^\dagger\psi)(x) \equiv U_\mu^\dagger(x-ae_\mu)\psi(x-ae_\mu), \tag{2.12}$$

where we consider the set of special unitary matrices $U_\mu(x)$ to be bi-local objects, or links, which connect the sites x and $x+ae_\mu$. Where it is necessary to consider a lattice gauge field A_μ^{L} directly, rather than through the links, we can let

$$U_\mu(x) = \exp igaA_\mu^{\mathrm{L}}(x) \tag{2.13}$$

be generated by an "average" gauge field that is naturally associated with the mid-points of the links on the lattice, with $\lim_{a\to 0} A_\mu^{\mathrm{L}}(x) = A_\mu(x)$.

[1] This derivation is essentially one given by Neuberger[3].

2.3.1 Gauge action

Recall that in the continuum the gauge fields were given a dynamic by adding the following term to the Lagrangian density,

$$\mathcal{L}_{\text{gluon}} = -\frac{1}{2} \operatorname{Tr} F_{\mu\nu}(x) F^{\mu\nu}(x). \tag{2.14}$$

In order to construct the lattice equivalent, we first consider a simple discretisation of the field strength tensor. For arbitrary $\psi(x)$ the continuum field strength tensor satisfies

$$[D_\mu, D_\nu]\psi(x) = igF_{\mu\nu}(x)\psi(x). \tag{2.15}$$

as D_μ is anti-Hermitian, we then have that $[D_\mu, D_\nu]^\dagger = -[D_\mu, D_\nu]$ and hence

$$[D_\mu, D_\nu]^\dagger [D_\mu, D_\nu]\psi(x) = g^2 F_{\mu\nu}(x) F_{\mu\nu}(x)\psi(x). \tag{2.16}$$

We then by analogy consider the equation

$$[\nabla^+_\mu, \nabla^+_\nu]^\dagger [\nabla^+_\mu, \nabla^+_\nu]\psi(x) \equiv g^2 F^+_{\mu\nu}(x) F^+_{\mu\nu}(x)\psi(x), \tag{2.17}$$

where

$$\nabla^+_\mu = \frac{1}{a}(T_\mu - 1) \tag{2.18}$$

is the forward covariant finite difference operator, and we wish to solve for the right hand side. Define $a_\mu = ae_\mu$ to be the vector of length a pointing in the direction of e_μ. Then clearly $[\nabla^+_\mu, \nabla^+_\nu] = [T_\mu, T_\nu]$, and as T_μ is a unitary operator we then have

$$(T_\mu T_\nu - T_\nu T_\mu)(T^\dagger_\nu T^\dagger_\mu - T^\dagger_\mu T^\dagger_\nu)\psi(x) = (2 - P_{\mu\nu} - P^\dagger_{\mu\nu})\psi(x), \tag{2.19}$$

where

$$P_{\mu\nu} = T_\mu T_\nu T^\dagger_\mu T^\dagger_\nu, \tag{2.20}$$

is also unitary. Furthermore,

$$P_{\mu\nu}\psi(x) = U_{\mu\nu}(x)\psi(x), \tag{2.21}$$

where $U_{\mu\nu}(x)$ is the elementary plaquette in the $\mu - \nu$ plane at the point x. That is, $U_{\mu\nu}(x)$ is the product of the links starting at the point x and going first in the μ and then ν directions around a 1×1 square on the lattice (see Fig. 2.3),

$$U_{\mu\nu}(x) = U_\mu(x) U_\nu(x + a_\mu) U^\dagger_\mu(x + a_\nu) U^\dagger_\nu(x). \tag{2.22}$$

So, we can now see that the solution to Eq. (2.17) is

$$g^2 F^+_{\mu\nu}(x) F^+_{\mu\nu}(x) = \frac{1}{a^4}(1 - U^\dagger_{\mu\nu}(x))(1 - U_{\mu\nu}(x)). \tag{2.23}$$

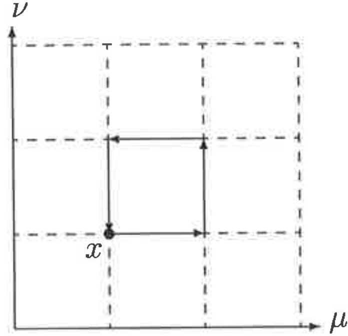

Figure 2.3: The product of links that form the plaquette $U_{\mu\nu}(x)$ on the lattice.

Hence, define the action due to the lattice gauge field to be

$$S_{\text{gauge}} = a^4 \sum_{x \in \mathbb{L}} \frac{1}{2} \operatorname{Tr} F^{+}_{\mu\nu}(x) F^{+}_{\mu\nu}(x), \tag{2.24}$$

in order to obtain

$$S_{\text{gauge}} = \beta \sum_{x \in \mathbb{L}} \frac{1}{4}(2 - \frac{1}{3} \operatorname{Tr}(U_{\mu\nu}(x) + U^{\dagger}_{\mu\nu}(x))) \tag{2.25}$$

where to obtain the correct continuum limit we set

$$\beta = \frac{6}{g^2}. \tag{2.26}$$

We can then write the action in a simpler fashion, as the sum on the action of all the unique plaquettes on the lattice,

$$S_{\text{gauge}} = \beta \sum_{x \in \mathbb{L}} \sum_{\mu < \nu} (1 - \frac{1}{3} \operatorname{Re} \operatorname{Tr} U_{\mu\nu}(x)). \tag{2.27}$$

The fact that we constructed the gauge action using plaquettes is quite intuitive, as we saw in the previous chapter that the field strength tensor is related to the Wilson line for a closed loop, and in particular for a square loop with sides of length a we have that

$$1 - U_{\square}(x, x) = 1 - \mathcal{P} e^{ig \int_{\square} F \cdot dS} = igF_{\mu\nu}(x) a^{\mu} a^{\nu} + \mathcal{O}(a^4). \tag{2.28}$$

The analogous result on the lattice is[4]

$$1 - U_{\mu\nu}(x) = iga^2 F^{+}_{\mu\nu}(x) + \mathcal{O}(a^3), \tag{2.29}$$

using Eq. (2.13) and the Baker-Campbell-Haussdorf formula, where we have set

$$F^{+}_{\mu\nu}(x) = \nabla^{+}_{\mu} A^{\mathrm{L}}_{\nu}(x) - \nabla^{+}_{\nu} A^{\mathrm{L}}_{\mu}(x) + ig[A^{\mathrm{L}}_{\mu}(x), A^{\mathrm{L}}_{\nu}(x)]. \tag{2.30}$$

The plaquette gauge action proposed above is the simplest such action that is also gauge invariant, and is known as the Wilson gauge action. Thus, we can now at this point write down the gauge invariant Lagrangian density for Lattice QCD with Wilson fermions and the Wilson gauge action,

$$\mathcal{L}_{\text{LQCD}}(x) = \bar{\psi}(x)D_{\text{w}}(m)\psi(x) - \sum_{\mu<\nu}(1 - \frac{1}{3}\operatorname{Re}\operatorname{Tr} U_{\mu\nu}(x)), \tag{2.31}$$

where

$$D_{\text{w}}(m) = \nabla\!\!\!/ + \frac{1}{2}\Delta + m. \tag{2.32}$$

2.3.2 Lattice Field Strength Tensor

The simple discretisation $F^{+}_{\mu\nu}$ proposed above, while valid, does not share the property of Hermiticity that the continuum field strength tensor possesses. Hence, although it served well as an auxiliary for constructing the gauge action, when we need to refer to the field strength tensor directly we desire a discretisation that bears more resemblance to its continuum counterpart. As the central covariant finite difference operator ∇_μ is anti-hermitian, it is then trivial to see if we consider the following equation

$$[\nabla_\mu, \nabla_\nu]\psi(x) \equiv igF^{\text{cl}}_{\mu\nu}(x)\psi(x) + \mathcal{O}(a^2), \tag{2.33}$$

then the solution $F^{\text{cl}}_{\mu\nu}(x)$ will be more "continuum-like". We first show the left hand side is free of $O(a)$ errors. Consider

$$\nabla_\mu = \frac{1}{2a}(T_\mu - T^\dagger_\mu) = \frac{1}{2a}(e^{aD_\mu} - e^{-aD_\mu}) = D_\mu + \frac{1}{3}a^2 D^3_\mu + O(a^4). \tag{2.34}$$

Then clearly

$$[\nabla_\mu, \nabla_\nu]\psi(x) = [D_\mu, D_\nu]\psi(x) + O(a^2) = igF_{\mu\nu}(x)\psi(x) + O(a^2). \tag{2.35}$$

Now, expanding the left hand side of Eq. (2.33) we see that

$$\begin{aligned}
[\nabla_\mu, \nabla_\nu]\psi(x) = \frac{1}{4a^2}\Big(&(U_\mu(x)U_\nu(x+a_\mu) - U_\nu(x)U_\mu(x+a_\nu))\psi(x+a_\mu+a_\nu) \\
&+ (U_\mu(x)U^\dagger_\nu(x+a_\mu-a_\nu) - U^\dagger_\nu(x-a_\nu)U_\mu(x-a_\nu))\psi(x+a_\mu-a_\nu) \\
&+ (U^\dagger_\mu(x-a_\mu)U_\nu(x-a_\mu) - U_\nu(x)U^\dagger_\mu(x-a_\mu+a_\nu))\psi(x-a_\mu+a_\nu) \\
+ (U^\dagger_\mu(x-a_\mu)&U^\dagger_\nu(x-a_\mu-a_\nu) - U^\dagger_\nu(x-a_\nu)U^\dagger_\mu(x-a_\mu-a_\nu))\psi(x-a_\mu-a_\nu)\Big).
\end{aligned} \tag{2.36}$$

By the unitarity of the links we have that

$$\begin{aligned}
U_\mu(x)U_\nu(x+a_\mu) - U_\nu(x)U_\mu(x+a_\nu) = \frac{1}{2}\big[&(U_{\mu\nu}(x) - 1)U_\nu(x)U_\mu(x+a_\nu) + \\
&(1 - U^\dagger_{\mu\nu}(x))U_\mu(x)U_\nu(x+a_\mu)\big],
\end{aligned} \tag{2.37}$$

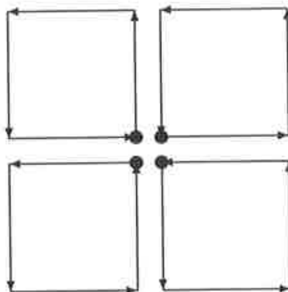

Figure 2.4: The four terms that make up $C_{\mu\nu}(x)$.

and similar equations also hold for the other terms in Eq. (2.36). Noting this, we expand the exponential in Eq. (2.13) to obtain

$$U_\mu(x) = 1 + \mathcal{O}(a), \tag{2.38}$$

and perform the Taylor expansion

$$\psi(x + a_\mu) = \psi(x) + \mathcal{O}(a) \tag{2.39}$$

in order to show

$$[\nabla_\mu, \nabla_\nu]\psi(x) = \frac{1}{8a^2}\Big((U_{\mu\nu}(x) - U^\dagger_{\mu\nu}(x))\psi(x) + (U_{-\nu\mu}(x) - U^\dagger_{-\nu\mu}(x))\psi(x)$$
$$+ (U_{\nu-\mu}(x) - U^\dagger_{\nu-\mu}(x))\psi(x) + (U_{-\mu-\nu}(x) - U^\dagger_{-\mu-\nu}(x))\psi(x)\Big) + \mathcal{O}(a^2). \tag{2.40}$$

We have made use of Eq. (2.35) to see that the leading $O(a)$ error terms must cancel. Hence, if we set

$$C_{\mu\nu}(x) = \frac{1}{4}(U_{\mu\nu}(x) + U_{-\nu\mu}(x) + U_{\nu-\mu}(x) + U_{-\mu-\nu}(x)), \tag{2.41}$$

then in order to satisfy Eq. (2.33) we define the lattice field strength tensor to be

$$F^{\mathrm{cl}}_{\mu\nu}(x) = \frac{1}{2iga^2}(C_{\mu\nu}(x) - C^\dagger_{\mu\nu}(x)). \tag{2.42}$$

This is known as the clover discretisation of $F_{\mu\nu}$, due to the shape of $C_{\mu\nu}$ resembling (with only a little imagination) that of a four-leaf clover (see Fig. 2.4). The field strength tensor defined in this way has reduced discretisation errors, $F^{\mathrm{cl}}_{\mu\nu}(x) = F_{\mu\nu}(x) + O(a^2)$.

2.4 Quenched QCD

In terms of the gauge links U and the fermion fields $\bar{\psi}, \psi$ the expectation value of an observable in the functional integral formalism is

$$\langle \mathcal{O} \rangle = \frac{1}{\mathcal{Z}} \int \mathcal{D}U\mathcal{D}\bar{\psi}\mathcal{D}\psi \; \mathcal{O}[U, \bar{\psi}, \psi]e^{-S[U,\bar{\psi},\psi]}, \tag{2.43}$$

where the action is the sum of the gauge action S_g and the fermion action S_f,

$$S[U,\bar{\psi},\psi] = S_\mathrm{g}[U] + S_\mathrm{f}[U,\bar{\psi},\psi], \tag{2.44}$$

where the fermion action depends on the fermion Dirac matrix D_f,

$$S_\mathrm{f} = \int d^4x \bar{\psi}(x) D_\mathrm{f}[U]\psi(x). \tag{2.45}$$

As the fermionic fields are Grassmannian, we can make use of the fact that

$$\det D_\mathrm{f} = \int \mathcal{D}\bar{\psi}\mathcal{D}\psi \; e^{-\int d^4x \bar{\psi}(x) D_\mathrm{f} \psi(x)} \tag{2.46}$$

to integrate out the fermionic degrees of freedom and obtain

$$\langle \mathcal{O} \rangle = \frac{1}{\mathcal{Z}} \int \mathcal{D}U \; \mathcal{O}_\mathrm{eff}[U] e^{-S_\mathrm{eff}[U]}, \tag{2.47}$$

where

$$\mathcal{O}_\mathrm{eff} = \int \mathcal{D}\bar{\psi}\mathcal{D}\psi \; \mathcal{O}[U,\bar{\psi},\psi] e^{-S_\mathrm{f}[U,\bar{\psi},\psi]}, \tag{2.48}$$

and the effective action becomes

$$S_\mathrm{eff}[U] = S_\mathrm{g}[U] - \ln \det D_\mathrm{f}[U]. \tag{2.49}$$

As we shall see later, it is computationally expensive to include the fermionic determinant in lattice simulations, and for this reason lattice QCD has made extensive use of the *quenched approximation,* where $\det D_\mathrm{f}[U]$ is set to unity. This is equivalent to neglecting the contribution of fermionic vacuum fluctuations. The effects of this approximation can be quite severe and will be discussed further on, but nonetheless quenched QCD shares many qualitative features with full (unquenched) QCD, and over the last two decades has provided valuable insight into the properties of non-Abelian gauge field theories.

2.4.1 Functional Integrals on the Lattice

After applying the lattice regularisation, the path integral above (in quenched QCD) becomes

$$\langle \mathcal{O} \rangle = \frac{1}{\mathcal{Z}} \int \prod_{x \in \mathbb{L}} \prod_{\mu=1}^{4} dU_\mu(x) \; \mathcal{O}[U] e^{-S[U]}. \tag{2.50}$$

To calculate this, we must evaluate a multi dimensional integral, where for each point x we have to integrate over the available degrees of freedom. The gauge field lies in $SU(3)$ and hence has eight degrees of freedom per link, and four links per site. Thus, for a lattice with L^4 sites our integration space has a dimension

$d = 32L^4$. If we sample N points per dimension to evaluate the integral, then the complexity of the functional integral is $O(N^d)$, that is, if we double N the time taken to calculate $\langle \mathcal{O} \rangle$ increases by a factor of 2^d[5]. Even with the most powerful modern computers a calculation of such magnitude is impossible.

As we have formulated our theory in Euclidean space, the generating functional plays an identical role to that of the partition function in statistical mechanics. This similarity between lattice QCD and statistical mechanics allows us to apply powerful statistical methods to the problem of simulating QCD. One such technique is importance sampling. The weighting of $e^{-S[U]}$ in the integrand above means we are only interested in a small portion of the available configuration space (that with small action), as the contribution of the remainder to the integral is exponentially suppressed. Importance sampling is a Monte Carlo technique which makes use of the fact that given a set of n representative bosonic field configurations U_i distributed according to $\exp(-S[U])$, the functional integral

$$\langle \mathcal{O} \rangle = \frac{1}{\mathcal{Z}} \int \mathcal{D}U \; \mathcal{O}[U] e^{-S[U]} \tag{2.51}$$

will be approximated by

$$\langle \mathcal{O} \rangle \approx \frac{1}{n} \sum_{i=1}^{n} \mathcal{O}[U_i], \tag{2.52}$$

with statistical errors that decrease as $1/\sqrt{n}$. Thus, in practice when we calculate observables on the lattice, we do so by generating a set of gauge field configurations randomly chosen with probability e^{-S}, and then evaluating the desired quantity on each of these configurations and calculating the ensemble average. The details of how appropriate configurations are chosen is covered in chapter 6, but we will note at this point that there are algorithms which only have a complexity of $O(\alpha L^4)$ for generating background gauge fields (where α depends on the desired physical properties of the gauge fields).

2.5 Systematic Improvements

We have constructed a basic formulation of QCD on the lattice, which is minimal in the sense that we started with the standard formulation of continuum QCD and performed only the necessary modifications required to restrict it to the lattice. While it is true that if we take $a \to 0$ we recover the desired limit, at finite lattice spacing the physical properties of our theory may differ markedly from the continuum. This poses some difficulties, not the least of which is that the dependence of an observable on a may be non-trivial, and hence extrapolating from a large finite a to the continuum may not be done reliably. In order to avoid this quandary, it is desirable to work in the *scaling region*, that is, at sufficiently small values of a such that the predicted physics does not vary with a, and hence may be considered to be continuum like and trivially extrapolated.

Unfortunately, the computational power required for a lattice simulation generally increases as some inverse power of a, and as such the scaling region may not be numerically accessible to lattice actions with large discretisation errors, such as the simple formulation we have constructed so far. However, so long as we obtain the correct continuum limit, we are free to modify our lattice formulation as we see fit. This is the essential spirit behind the process of improving lattice QCD, that is removing lattice artifacts from the theory so that the scaling region is extended to larger a and hence reducing the computational power required to conduct lattice simulations.

Chapter 3

Improved Lattice Actions

One of the key features of lattice field theory is that there are many formulations of QCD on the lattice which will reproduce the desired theory in the continuum limit. This enables us to perform many modifications to the discretised theory, and in particular one can choose to systematically eliminate the discretisation errors of a particular order in a. This is done by adding additional higher order terms to the basic formulation of the theory. However, there are also other less obvious improvements that can be applied that improve the properties of a given lattice action, ranging from filtering out ultraviolet lattice artifacts to restoring a symmetry that evaded the lattice community for two decades, chiral symmetry.

3.1 Mean Field Improvement

When one wishes to compare lattice quantities with those in the continuum, one frequently performs expansions in powers of a, such as expanding Eq.(2.13)

$$U_\mu(x) = 1 + igaA_\mu(x) - \frac{1}{2}g^2a^2A_\mu^2(x) + \mathcal{O}(a^3). \tag{3.1}$$

The difficulty in such expansions comes from the higher order or tadpole terms, as A_μ^2 looks like a gluonic loop propagator,

$$A_\mu^2(x) \sim \int d^4q \frac{1}{q^2} \sim q^2 \sim \frac{\pi^2}{a^2}, \tag{3.2}$$

which goes like the cutoff. This means that the expansion is really only in powers of g, and in the region we are interested in g is not small. Thus the higher order terms may contribute significantly, causing lattice perturbation theory to fail. However, Lepage and Mackenzie[6] proposed the use of a standard tool from statistical mechanics, mean field improvement, in order to divide out the contribution from the so-called "tadpole" terms. The mean link u_0 can be defined

in several ways, and in this work we choose to define it as the fourth root of the average plaquette,

$$u_0 = \langle \frac{1}{3}\mathrm{ReTr}U_{\mu\nu}(x)\rangle^{\frac{1}{4}}_{x,\mu<\nu}. \tag{3.3}$$

The process of mean field improvement then, simply consists of performing the replacement

$$U_\mu(x) \to \frac{U_\mu(x)}{u_0} \tag{3.4}$$

in the lattice action and the desired lattice operators. This process has the effect of compensating for much of the renormalisation of the bare lattice parameters that occurs in the presence of gauge field interactions.

3.2 Improving the Gauge Action

The Wilson (plaquette) gauge action was constructed by considering

$$S_{\text{gauge}} = a^4 \sum_{x\in\mathbb{L}} \frac{1}{2}\,\mathrm{Tr}\,F^{+}_{\mu\nu}(x)F^{+}_{\mu\nu}(x). \tag{3.5}$$

Through Eq. (2.28) we may see that expanding the plaquette Wilson loop

$$\frac{\beta}{6}\,\mathrm{Tr}(1-U_\Box(x,x))(1-U^\dagger_\Box(x,x)) = a^4\frac{1}{2}\,\mathrm{Tr}\,F_{\mu\nu}(x)F_{\mu\nu}(x) + \mathcal{O}(a^6), \tag{3.6}$$

and hence the plaquette gauge action differs from the continuum gauge action at $\mathcal{O}(a^2)$. If instead, one chooses to use the clover discretisation of the field strength tensor,

$$S'_{\text{gauge}} = a^4 \sum_{x\in\mathbb{L}} \frac{1}{2}\,\mathrm{Tr}\,F^{\text{cl}}_{\mu\nu}(x)F^{\text{cl}}_{\mu\nu}(x), \tag{3.7}$$

and then expands the right hand side as we did before, one obtains contributions not only from the plaquette loops $U_{\mu\nu}(x)$, but also horizontal rectangle $R^{(2\times1)}_{\mu\nu}$, vertical rectangle $R^{(1\times2)}_{\mu\nu}$ and "figure of eight" $R^{(8)}_{\mu\nu}$ loops (see Fig. 3.1). One can incorporate these "higher" loops into the definition of the gauge action in order to construct an improved action. For computational efficiency one generally only considers the horizontal and vertical rectangles, and defines the plaquette plus rectangle gauge action to be

$$S^{P+R}_{\text{gauge}} = \beta \sum_{x\in\mathbb{L}} \sum_{\mu<\nu} \frac{1}{3}\,\mathrm{Re}\,\mathrm{Tr}[c_P(1-U_{\mu\nu}(x)) + c_R(1-R^{(2\times1)}_{\mu\nu}(x)) + c_R(1-R^{(1\times2)}_{\mu\nu}(x))], \tag{3.8}$$

where c_P and c_R are coefficients that determine the relative contribution of the plaquette and rectangle terms.

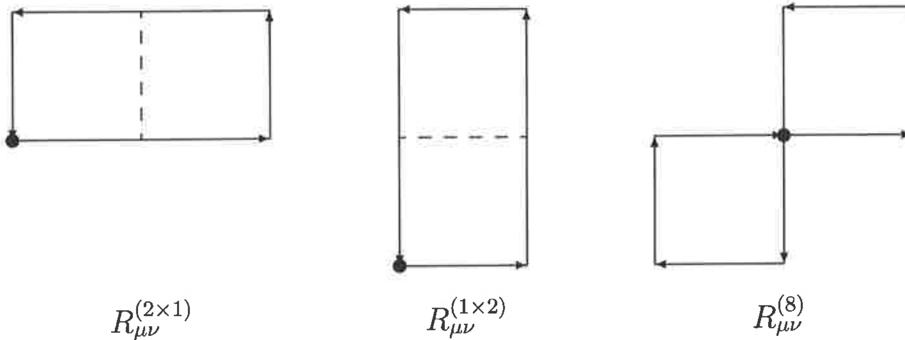

Figure 3.1: The horizontal rectangle, vertical rectangle and "figure of eight" terms that arise in the expansion of $F_{\mu\nu}^{\rm cl}(x)F_{\mu\nu}^{\rm cl}(x)$.

One can derive a perturbatively improved action by expanding the Wilson loop defined by Eq. (1.15) corresponding to the plaquette and rectangle loops. One finds that the leading term of the different loops all agree with the continuum gauge action, and the sub-leading $\mathcal{O}(a^2)$ error term is also the same for each, but the coefficients for each of the terms differs between the loops. If one then makes the choice

$$c_P = \frac{5}{3}, \quad c_R = -\frac{1}{12}, \tag{3.9}$$

one obtains an action that (at tree-level) has only $\mathcal{O}(a^4)$ errors. To compensate for the effects of non-trivial gauge fields, one introduces a relative factor of u_0 into the coefficients, with a power determined according to the difference in the number of links that make up each loop. Thus we arrive at the tadpole-improved Lüscher-Weisz action[7],

$$S_{\rm gauge}^{\rm imp} = \frac{5}{9}\beta \sum_{x\in\mathbb{L}} \sum_{\mu<\nu} {\rm Re\,Tr}[(1-U_{\mu\nu}(x)) - \frac{1}{20u_0^2}(1-R_{\mu\nu}^{(2\times1)}(x)) - \frac{1}{20u_0^2}(1-R_{\mu\nu}^{(1\times2)}(x))]. \tag{3.10}$$

Alternatively, one may use Monte-Carlo Renormalisation Group (MCRG) flow techniques in order to derive a non-perturbatively improved gauge action. One such action is the doubly-blocked Wilson 2 action, which is derived by creating large lattices with Wilson glue at a small a, and then performing two steps of blocking (averaging) the links to produce a smoother lattice which has a lattice spacing $a' = 4a$, but has a quarter the number of points on each side. One then attempts to recreate the properties of this smoothed lattice by exploring the 2-parameter space (c_P, c_R) of gauge fields which have the same (lattice) size, in

order to obtain the DBW2 action[8, 9, 10, 11],

$$S^{\rm dbw2}_{\rm gauge} = \frac{1}{3}\beta \sum_{x\in\mathbb{L}} \sum_{\mu<\nu} {\rm Re\,Tr}[c_P(1-U_{\mu\nu}(x)) - c_R(1-R^{(2\times1)}_{\mu\nu}(x)) - c_R(1-R^{(1\times2)}_{\mu\nu}(x))]. \tag{3.11}$$

3.3 Improving the Fermion Action

If one considers the action of the central covariant finite difference operator, ∇_μ, on $\psi(x)$ and performs a Taylor expansion, it is straightforward to show that

$$\nabla\!\!\!/ = D\!\!\!/ + \mathcal{O}(a^2). \tag{3.12}$$

However, the addition of the Wilson term $\frac{a}{2}\Delta$ to form the Wilson-Dirac operator $D_{\rm w}$ introduces $\mathcal{O}(a)$ errors,

$$D_{\rm w} \equiv \nabla\!\!\!/ + \frac{a}{2}\Delta = D\!\!\!/ + \mathcal{O}(a). \tag{3.13}$$

We calculate the leading discretisation error in $D_{\rm w}$ by writing it in terms of the T_μ,

$$D_{\rm w} = \frac{1}{a}\sum_{\mu=1}^{4} 1 - \frac{1}{2}[(1-\gamma_\mu)T_\mu + (1+\gamma_\mu)T^\dagger_\mu]. \tag{3.14}$$

Then we use $T_\mu = e^{aD_\mu}$ to see that

$$D_{\rm w} = \frac{1}{a}\sum_{\mu=1}^{4} 1 - \frac{1}{2}[(1-\gamma_\mu)(1 + aD_\mu + \frac{1}{2}a^2D^2_\mu + \mathcal{O}(a^3)) + (1+\gamma_\mu)(1 - aD_\mu + \frac{1}{2}a^2D^2_\mu + \mathcal{O}(a^3))]. \tag{3.15}$$

Thus,

$$D_{\rm w} = \sum_{\mu=1}^{4} \gamma_\mu D_\mu - a\frac{1}{2}D^2_\mu + \mathcal{O}(a^2). \tag{3.16}$$

It is simple to show that

$$D\!\!\!/^2 = D\cdot D + \frac{1}{4}[\gamma^\mu, \gamma^\nu][D_\mu, D_\nu], \tag{3.17}$$

and hence that

$$D_{\rm w} = D\!\!\!/ - a\frac{1}{2}(D\!\!\!/^2 - g\frac{1}{2}\sigma\cdot F) + \mathcal{O}(a^2), \tag{3.18}$$

where we have set

$$\sigma_{\mu\nu} = \frac{i}{2}[\gamma_\mu, \gamma_\nu]. \tag{3.19}$$

Now, any field which satisfies the Dirac equation,

$$\not{D}\psi(x) = -m\psi(x), \tag{3.20}$$

will also satisfy

$$\not{D}^2\psi(x) = m^2\psi(x). \tag{3.21}$$

Admitting a redefinition of the bare lattice mass $m' = m + \frac{a}{2}m^2$ then gives

$$D_{\text{w}} + m' = \not{D} + m + \frac{1}{4}ag\sigma \cdot F + \mathcal{O}(a^2). \tag{3.22}$$

Hence if we subtract the so-called clover term (taking its name from the presence of the lattice field strength tensor) from the Wilson fermion action we obtain the Sheikholeslami-Wohlert action[12],

$$S_{\text{sw}} = \bar{\psi}(x)D_{\text{w}}\psi(x) - \frac{1}{4}ag\bar{\psi}(x)\sigma \cdot F^{\text{cl}}(x)\psi(x). \tag{3.23}$$

At tree-level this lattice action differs from the continuum fermion action at $\mathcal{O}(a^2)$. In the presence of a non-trivial gauge background the coefficient of the clover term c_{sw} gets renormalised away from unity, and so our clover improved discretisation of the Dirac operator becomes

$$D_{\text{cl}} = \not{\nabla} + \frac{a}{2}(\Delta - \frac{1}{2}c_{\text{sw}}\sigma \cdot F^{\text{cl}}). \tag{3.24}$$

A mean field estimate of the clover coefficient yields

$$c_{\text{sw}} = \frac{1}{u_0^3}, \tag{3.25}$$

calculated by counting the difference in the number of link products between F^{cl} and $\not{\nabla}$. Alternatively, one may estimate the coefficient non-perturbatively using numerical techniques[13]. The non-perturbatively improved clover action has been shown to have leading $\mathcal{O}(a^2)$ discretisation errors, whereas the mean field improved clover action still possesses some residual $\mathcal{O}(a)$ errors[14]. In this work we (by convention) set $c_{\text{sw}} = 1$ and absorb the mean field improvement into the definition of $F_{\mu\nu}$.

3.4 Improving the Field Strength Tensor

The clover discretisation of the field strength tensor was defined as

$$F^{\text{cl}}_{\mu\nu}(x) = \frac{1}{2iga^2}(C_{\mu\nu}(x) - C^{\dagger}_{\mu\nu}(x)), \tag{3.26}$$

where $C_{\mu\nu}(x)$ was the sum of four plaquette terms shown in Fig. 2.4. Making use of the expansion of the Wilson line corresponding to the plaquette,

$$1 - U_{\mu\nu}(x) = iga^2 F_{\mu\nu}(x) + \mathcal{O}(a^4), \tag{3.27}$$

one can show that

$$F^{\text{cl}}_{\mu\nu}(x) = F_{\mu\nu}(x) + \mathcal{O}(a^2). \tag{3.28}$$

In the same way that the addition of higher loops to the gauge action allowed us to reduce the discretisation errors, we can add additional higher "clover" loops to $F^{\text{cl}}_{\mu\nu}$ to produce an improved field strength tensor[15]. In general, define the improved field strength tensor to be

$$F^{\text{imp}}_{\mu\nu}(x) = k_1 F^{1\times1}_{\mu\nu} + k_2 F^{2\times2}_{\mu\nu} + \frac{k_3}{2}(F^{2\times1}_{\mu\nu} + F^{1\times2}_{\mu\nu}) + \frac{k_4}{2}(F^{3\times1}_{\mu\nu} + F^{1\times3}_{\mu\nu}) + k_5 F^{3\times3}_{\mu\nu}, \tag{3.29}$$

where

$$F^{m\times n}_{\mu\nu}(x) = \frac{1}{2iga^2}(C^{m\times n}_{\mu\nu}(x) - C^{\dagger m\times n}_{\mu\nu}(x)), \tag{3.30}$$

and $C^{m\times n}_{\mu\nu}(x)$ corresponds to the sum of the four $m \times n$ loops at the point x in the clover formation.

Demanding that $F^{\text{imp}}_{\mu\nu}(x)$ be free of $\mathcal{O}(a^2)$ and $\mathcal{O}(a^4)$ errors gives an under-constrained set of linear equations for the coefficients k_i, in which k_5 can be considered a free parameter. Different choices of k_5 give different versions of $F^{\text{imp}}_{\mu\nu}$ that contain three ($k_3 = k_4 = 0$), four (one $k_i = 0, i \in \{1, 2, 5\}$) or five (all $k_i \neq 0$) different loop terms (see Fig. 3.2). For computational efficiency, in this work we make use of the 3-loop improved field strength tensor with mean field improved coefficients,

$$F^{\text{3L}}_{\mu\nu}(x) = \frac{3}{2u_0^4} F^{1\times1}_{\mu\nu} - \frac{3}{20u_0^8} F^{2\times2}_{\mu\nu} + \frac{1}{90u_0^{12}} F^{3\times3}_{\mu\nu}. \tag{3.31}$$

3.5 Smeared Link Actions

For typical gauge field configurations the mean link $u_0 \approx 0.9$, which means $u_0^4 \approx 0.67$ and $u_0^{12} \approx 0.28$. This implies that the perturbative corrections based upon mean field improvement are large, and therefore are unlikely to provide an accurate estimate of the real value of the coefficients needed for highly improved objects such as $F^{\text{imp}}_{\mu\nu}$ in the presence of fluctuating gauge backgrounds. The deviation of u_0 from one is largely due to the ultra-violet, short distance fluctuations that mask the infrared, long distance structures present in equilibrium gauge field configurations.

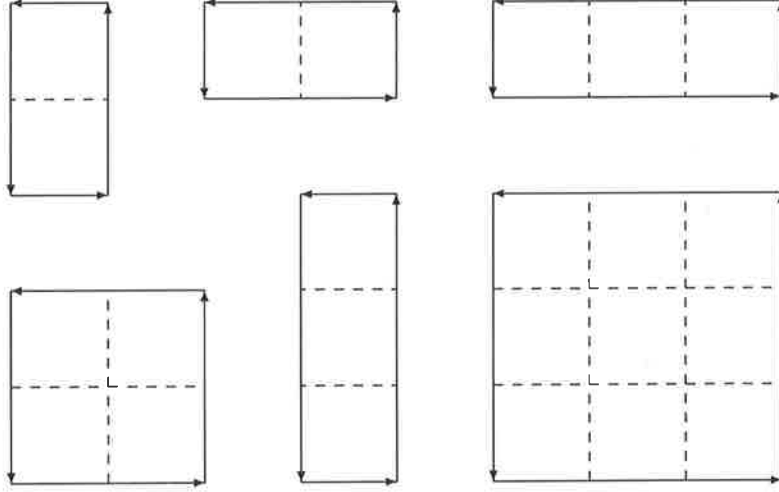

Figure 3.2: The additional loops that are make up $F_{\mu\nu}^{\mathrm{imp}}$. Along the top are the 1×2 and 2×1 rectangles and the 3×1 ladder. Along the bottom are the 2×2 window, 1×3 ladder and the 3×3 picture window.

3.5.1 Cooling and Smearing

It is possible to filter out the short distance fluctuations present in the gauge field by various smoothing processes. Cooling [16, 17, 18, 19, 20] is one method where the long distance topological structure of the underlying gauge field is revealed by iteratively updating each link on the lattice in such a way as to remove local fluctuations in the action. APE-smearing[21, 22, 23, 24] is a process where each link is smoothed or "fattened" through a gauge covariant averaging process with the neighbouring links. Specifically, in a single sweep of APE-smearing the "staples" which are the next shortest path between the link's endpoints are combined with the original "thin" link to produce the smeared links,

$$\begin{aligned} U_\mu^{(\alpha)}(x) &= \mathcal{P}\left((1-\alpha)U_\mu(x) + \frac{\alpha}{6}\sum_{\pm\nu\neq\mu} U_\nu(x)U_\mu(x+a_\nu)U_\nu^\dagger(x+a_\mu)\right) \\ &= \mathcal{P}\left((1-\alpha)\longrightarrow + \frac{\alpha}{6}\sum_{\nu\neq\mu} \sqcap + \sqcup\right). \end{aligned} \tag{3.32}$$

Here $\mathcal{P}$ denotes projection of the RHS of Eq. (3.32) back to the SU(3) gauge group, which is necessary as the group is not closed under addition. The means by which one projects back to SU(3) is not unique, but one should choose a method which is gauge covariant. The specifics of $SU(3)$ projection are discussed elsewhere in section 6.3.

The APE smearing process may be applied multiple times, producing a set of smeared or "fat" links which depend upon the smearing fraction α and the number of sweeps n_{ape}. Although there are two parameters available, it has been shown that the resultant smeared links effectively only depend upon the product αn_{ape}[25]. Thus, unless otherwise stated we choose $\alpha = 0.7$ in this work, which is close to the maximal valid choice $\alpha = 0.75$, and only vary the number of sweeps n_{ape}.

3.5.2 Fat Link Irrelevant Operator Actions

Both smearing and cooling smooth out the gauge field such that the mean link becomes very close to unity. The major advantage of APE-smearing compared to cooling is that the former can be represented in the form of higher-dimension operators applied to the gauge field, which disappear in the continuum limit. One can then formulate one's fermion action of choice in terms of APE smeared links[26, 27], and the coefficients for the irrelevant operators (such as c_{sw}) will be well-estimated in terms of the (smoothed) mean link, u_0^{fl}. Such actions will possess the correct continuum limit, and avoid the need for non-perturbative tuning of action coefficients. However, the process of APE-smearing removes short-distance physics, and therefore such actions will not be able to reproduce correctly quantities which are sensitive to such physics. An alternative is to divide the fermionic action into two parts, the first consisting of the naive lattice Dirac operator, and the second consisting of the remaining higher-dimensional irrelevant operators, such as the Wilson term and the clover term[3, 28]. These Fat Link Irrelevant (FLI) actions will still have knowledge of all the pertinent short-distance physics and at the same time will only need perturbative estimates for the coefficients of the irrelevant terms.

One such action which we will focus on in this work is the Fat Link Irrelevant Clover (FLIC) action[28],

$$D_{\mathrm{flic}} = \nabla\!\!\!/_{\mathrm{mfi}} + \frac{1}{2}(\Delta^{\mathrm{fl}}_{\mathrm{mfi}} - \frac{1}{2}\sigma \cdot F^{\mathrm{fl}}_{\mathrm{mfi}}) + m, \tag{3.33}$$

where the Wilson and clover terms are constructed of fat links, and all links have been mean field improved, that is

$$U_\mu(x) \to \frac{U_\mu(x)}{u_0}, \quad U^{\mathrm{fl}}_\mu(x) \to \frac{U^{\mathrm{fl}}_\mu(x)}{u^{\mathrm{fl}}_0}. \tag{3.34}$$

In the absence of a clover term, we refer to the action as the Fat Link Irrelevant Wilson (FLIW) action.

3.6 Lattice Symmetries

It was mentioned in section 1.3.4 that when regularising loop integrals in perturbation theory one often violates one symmetry or another of the theory in question, most often translational invariance. Similarly, on the lattice it is difficult to maintain all of the symmetries of continuum QCD. As stated in section 2.1, the available operators on the lattice are those whose action on a function evaluated on the lattice depends only upon the value of the function at lattice points. So, for example, in the continuum the available translation operators $e^{r\cdot\partial}$ are parameterised by $r \in \mathbb{R}^4$, and subsequently continuum QCD possesses an $\mathbb{R}^4$ translation symmetry group. On the (infinite or periodic) lattice, however, the available translation operators $e^{an\cdot\partial}$ are parameterised by $n \in \mathbb{Z}^4$ and so lattice QCD only possesses a discrete $\mathbb{Z}^4$ translational symmetry group, which is a subgroup of $\mathbb{R}^4$.

If we consider a field $\phi(an_\mu)$ evaluated on the lattice, and then apply an arbitrary translation on the x-axis say, $e^{r\partial_x}$, then the resultant field $\phi(an_\mu + re_x)$ will not in general lie on the lattice. However, setting $m = \lceil r/a \rceil$ we can perform a Taylor expansion, $\phi(an_\mu + re_x) = \phi(an_\mu + ma_x) + (ma - r)\partial_x\phi((an_\mu + ma_x) + O(ma - r)^2$ to see that the continuous symmetry is only violated at $O(a)$ and hence will be restored in the continuum limit. Similarly the $SO(4)$ Lorentz group reduces to the discrete subgroup of rotations by $\frac{\pi}{2}$, but is also restored in the continuum limit.

3.6.1 Chiral Symmetry

In the case of massless quarks, the fermionic part of the QCD Lagrangian density in the continuum is

$$\mathcal{L}_{m=0} = \bar{\psi} \not{D} \psi. \tag{3.35}$$

This Lagrangian is invariant under axial rotations,

$$\psi(x) \rightarrow e^{i\alpha\gamma_5}\psi(x), \quad \bar{\psi}(x) \rightarrow \bar{\psi}(x)e^{i\alpha\gamma_5}. \tag{3.36}$$

Example 3.6.1. *Verify this.*

Proof. For infinitesimal α, we have up to $O(\alpha^2)$ that

$$\psi \rightarrow \psi' = (1 + i\alpha\gamma_5)\psi, \quad \bar{\psi} \rightarrow \bar{\psi}' = \bar{\psi}(1 + i\alpha\gamma_5),$$

and hence

$$\begin{aligned} \mathcal{L}'_{m=0} &= \bar{\psi}(1 + i\alpha\gamma_5)\not{D}(1 + i\alpha\gamma_5)\psi \\ &= \bar{\psi}\not{D}\psi + i\alpha(\bar{\psi}\gamma_5\not{D}\psi + \bar{\psi}\not{D}\gamma_5\psi) + O(\alpha^2) \\ &= \mathcal{L}_{m=0}, \end{aligned}$$

as $\{\gamma_\mu, \gamma_5\} = 0$. □

This symmetry of the massless Lagrangian is called chiral symmetry, which is one of the most important symmetries of QCD, as it is the spontaneous breaking of chiral symmetry which is thought to be responsible for the dynamical generation of nearly all of mass of the nucleon. From the previous example we can see that chiral symmetry is signified by the relation

$$\not{D}\gamma_5 + \gamma_5\not{D} = 0. \tag{3.37}$$

On the lattice, it is immediately obvious that the naive Dirac operator is chirally symmetric,

$$\not{\nabla}\gamma_5 + \gamma_5\not{\nabla} = 0. \tag{3.38}$$

Thus it is possible to define the left and right handed chiral projectors

$$\Gamma_5^- = \frac{1}{2}(1-\gamma_5), \qquad \Gamma_5^+ = \frac{1}{2}(1+\gamma_5), \tag{3.39}$$

in order to decompose the fermion fields in terms of their helicity $\psi = \psi_{\text{left}} + \psi_{\text{right}}$,

$$\begin{aligned} \psi_{\text{left}} &= \Gamma_5^-\psi, \quad & \psi_{\text{right}} &= \Gamma_5^+\psi, \\ \bar{\psi}_{\text{left}} &= \bar{\psi}\Gamma_5^-, \quad & \bar{\psi}_{\text{right}} &= \bar{\psi}\Gamma_5^+ \end{aligned} \tag{3.40}$$

and hence provide a corresponding decomposition of the massless Lagrangian into chiral sectors (using $\gamma_\mu\Gamma_5^\pm = \Gamma_5^\mp$, and $\Gamma_5^\pm\Gamma_5^\mp = 0$),

$$\mathcal{L} = \bar{\psi}_{\text{right}}\not{\nabla}\psi_{\text{left}} + \bar{\psi}_{\text{left}}\not{\nabla}\psi_{\text{right}}. \tag{3.41}$$

However, the lattice Laplacian Δ does not anti-commute with γ_5, and hence when we introduce the Wilson term to eliminate the fermion doublers we explicitly break chiral symmetry even in the massless case,

$$D_{\text{w}}\gamma_5 + \gamma_5 D_{\text{w}} = a\gamma_5\Delta. \tag{3.42}$$

This leads us to the (somewhat notorious) problem of finding a lattice fermion action which is both chirally symmetric and free of fermion doublers. The difficulty in solving this problem was made concrete by a no-go theorem derived by Nielsen and Ninomiya[29, 30, 31],

The Nielsen-Ninomiya No-Go Theorem. *It is not possible to find a lattice Dirac operator D_a that simultaneously satisfies the following four conditions:*

1. **Correct continuum limit.** *In the limit $a \to 0$, $D_a \to \not{D}$, where D_μ is the covariant derivative in the continuum, giving rise to a single fermion species of zero or finite mass.*

2. **No Doublers.** *All other modes of D_a are of order $1/a$, that is, all other fermion species decouple in the continuum limit (grow infinitely heavy).*

3. **Locality.** *D_a is exponentially local, that is, the norm of the matrix elements of D_a decays exponentially as $|x-y|$ grows large.*

4. **Chirality.** *D_a does not explicitly break chiral symmetry, that is, $D_a\gamma_5 + \gamma_5 D_a = 0$.*

The importance of the first two conditions is clear, and the third condition is also important for several reasons, not the least of which is that the less local an operator is the more computationally expensive it is to evaluate. Presented with such a situation, chiral symmetry was deemed the condition that was the least unpalatable to sacrifice, arguing that although it was explicitly broken, this was done at $O(a)$ and hence should be restored when one takes the continuum limit.

Unfortunately, this choice was not without its drawbacks. Violating chiral symmetry at the massless level removes the protection of the lattice fermion action against additive quark mass renormalisation. This has several consequences, such as the introduction of exceptional gauge field configurations, and a loss of the connection between exact fermionic zero modes and topology (see §4.3.1). Furthermore, the absence of chiral symmetry at finite lattice spacing restricted the phenomenology that could be explored on the lattice, excluding the possibility of calculating chirally sensitive physical quantities.

3.6.2 Ginsparg-Wilson Relation

A means of circumventing the powerful no-go theorem was identified early on by Ginsparg and Wilson[32]. They saw that if instead of demanding chiral symmetry as it was defined in the continuum, one instead required a lattice deformed version of chiral symmetry,

$$D_a\gamma_5 + \gamma_5 D_a = 2aD_a\gamma_5 D_a, \tag{3.43}$$

then the no-go theorem would not present an obstruction to finding such a D_a, with the advantage that continuum chiral symmetry is only "softly" broken. This definition of lattice chiral symmetry is known as the Ginsparg-Wilson relation, and is equivalent to allowing the fermion and anti-fermion fields to transform differently under deformed axial rotations. To highlight this deformed symmetry, define

$$\hat{\gamma}_5 = \gamma_5(1-2aD_a), \tag{3.44}$$

then

$$D_a\hat{\gamma}_5 + \gamma_5 D_a = 0. \tag{3.45}$$

It is immediate then that the Lagrangian

$$\mathcal{L}_a = \bar{\psi} D_a \psi \tag{3.46}$$

is invariant under lattice deformed axial rotations,

$$\psi(x) \to e^{i\alpha\gamma_5}\psi(x), \quad \bar{\psi}(x) \to \bar{\psi}(x)e^{i\alpha\hat{\gamma}_5}. \tag{3.47}$$

Hence, defining the corresponding deformed chiral projectors

$$\hat{\Gamma}_5^- = \frac{1}{2}(1 - \hat{\gamma}_5), \qquad \hat{\Gamma}_5^+ = \frac{1}{2}(1 + \hat{\gamma}_5),$$

if we decompose the fermion fields according to

$$\begin{aligned} \psi_{\text{left}} &= \Gamma_5^- \psi, \quad \psi_{\text{right}} = \Gamma_5^+ \psi, \\ \bar{\psi}_{\text{left}} &= \bar{\psi}\hat{\Gamma}_5^-, \quad \bar{\psi}_{\text{right}} = \bar{\psi}\hat{\Gamma}_5^+, \end{aligned} \tag{3.48}$$

then $\mathcal{L}_a$ decomposes into chiral sectors as desired,

$$\mathcal{L}_a = \bar{\psi}_{\text{right}} D_a \psi_{\text{left}} + \bar{\psi}_{\text{left}} D_a \psi_{\text{right}}. \tag{3.49}$$

The formulation of chiral symmetry on the lattice via the Ginsparg-Wilson relation is enough to eliminate additive mass renormalisation, and other such undesired consequences of breaking chiral symmetry. Unfortunately, an explicit solution to this relation evaded the lattice community for well over a decade, until recently when a solution was developed by Neuberger and Naryanan, that of the overlap formalism.

Chapter 4

Overlap Fermions

The long standing problem of constructing a satisfactory lattice regularisation of chiral gauge theories was solved by extensive work conducted during the nineties. Sparked by the domain wall formulation[33] and earlier work[34, 35], Neuberger and Naryanan developed the overlap formalism[36, 37, 38, 39] and subsequently the overlap Dirac operator[40, 41] as a realisation of chiral fermions on the lattice. Not only did this open the way for exploring the effects of dynamical chiral symmetry breaking within QCD, it also allowed for the formulation of massless fermions on the lattice, such as the neutrino sector of the Standard Model. The obstructions to a formulation of lattice chiral fermions are subtle, and construction of the overlap is correspondingly subtle. It is not possible to do full justice to the overlap formalism within the scope of this work, but rather we will endeavour to represent here some of the key points in its formulation, directing the reader elsewhere for a detailed review[42].

4.1 The chiral determinant

Recall from §2.4 that the fermionic degrees of freedom can be integrated out of the QCD path integral, to give the determinant of the Dirac operator (or its discretisation). We wish to write down an effective Lagrangian $\mathcal{L}_{\text{eff}}$ which, when placed in a path integral, will give a fermionic determinant that corresponds to a single species of massless fermions (of definite chirality).

Henceforth, we will work in lattice units ($a = 1$), except where explicit a dependence is necessary or instructive. Dimensional considerations can always be used to restore the a dependence. In the previous chapter, we showed that the naive massless fermionic Lagrangian could be chirally decomposed,

$$\mathcal{L} = \bar{\psi}(\Gamma_5^+ \nabla\!\!\!\!/ + \Gamma_5^- \nabla\!\!\!\!/)\psi = \bar{\psi}_{\text{right}} \nabla\!\!\!\!/ \psi_{\text{left}} + \bar{\psi}_{\text{left}} \nabla\!\!\!\!/ \psi_{\text{right}}. \tag{4.1}$$

Now, let $\mathcal{M}$ be a matrix in flavour space, and consider the following,

$$\mathcal{L} = \bar{\psi} \nabla\!\!\!\!/ \psi + \bar{\psi}(\Gamma_5^+ \mathcal{M} + \Gamma_5^- \mathcal{M}^\dagger)\psi. \tag{4.2}$$

If the above expression were in the continuum, then it would represent n_{left} left-handed and n_{right} right-handed massless fermions, where n_{left} is the number of zero eigenmodes of $\mathcal{M}$ and n_{right} is the number of zero eigenmodes of $\mathcal{M}^\dagger$. If $\mathcal{M}$ is finite dimensional then we must have that $n_{\text{left}} = n_{\text{right}}$ and hence it is not possible to represent a single fermion species of a given chirality. However, if we make $\mathcal{M}$ infinite dimensional then we can avoid this and have $|n_{\text{left}} - n_{\text{right}}| = 1$, and in particular $n_{\text{left}} = 1, n_{\text{right}} = 0$.

This can be accomplished by choosing $\mathcal{M}$ to be an operator on a (continuous) one-dimensional flavour space, which we parameterise by s (with $\partial_s = \frac{\partial}{\partial s}$),

$$\mathcal{M} = -\partial_s - m_+\theta(s) + m_-\theta(-s), \tag{4.3}$$

where $m_\pm$ are constants with the dimensions of mass, interpreted as regulators, and $\theta(s)$ is the Heaviside function. Then what we have constructed so far can be interpreted as a one-parameter family of (naive) fermion species on the lattice, where the mass is subject to a discontinuity at $s = 0$, but otherwise is independent of s on either side of this defect. We have started from a naive fermion discretisation, and $\mathcal{M}$ as given does nothing to remove the fermion doublers. We can avoid this problem in the usual way, through the addition of a Wilson term,

$$\mathcal{L}_{\text{eff}} = \bar{\psi}(\nabla\!\!\!\!/ + \frac{1}{2}\Delta + \Gamma_5^+\mathcal{M} + \Gamma_5^-\mathcal{M}^\dagger)\psi. \tag{4.4}$$

If we interpret s as an "evolution" parameter, we can define Hamiltonians on either side of the defect,

$$\mathcal{H}_\pm = \hat{a}^\dagger H_\pm \hat{a}, \tag{4.5}$$

where

$$H_\pm = \gamma_5(\nabla\!\!\!\!/ + \frac{1}{2}\Delta \mp m_\pm), \tag{4.6}$$

and $\hat{a}^\dagger_{x,a,\alpha}, \hat{a}_{x,a,\alpha}$ are single-particle creation and annihilation operators (one for each site, colour and spin index value) obeying canonical anti-commutation relations. Then the corresponding "evolution" operators are,

$$\hat{T}_\pm = e^{-\mathcal{H}_\pm}, \tag{4.7}$$

and we denote the ground states of $\hat{T}_\pm$ by $|\pm\rangle$. Our effective Lagrangian describes two five-dimensional many body systems. The effect of the path integral,

$$\int \mathcal{D}\bar{\psi}\mathcal{D}\psi\; e^{-S_{\text{eff}}}, \tag{4.8}$$

where

$$S_{\text{eff}} = \int ds \sum_{x\in\mathbb{L}} \mathcal{L}_{\text{eff}}(x, s), \tag{4.9}$$

is simply to evolve each system on either side of the defect to infinite "time" $s = \pm\infty$, that is, project out the ground states of each system. After an appropriate

regularisation, the chiral determinant is given by the overlap of the two ground states of $T_{\pm}$,

$$\langle +|-\rangle. \tag{4.10}$$

Now, the overlap is insensitive to the overall rescaling of the Hamiltonians, so we can redefine

$$H_{\pm} = \epsilon(H_{\mathrm{w}}(\mp m_{\pm})), \tag{4.11}$$

where ϵ is the matrix sign function,

$$\epsilon(H) = \frac{H}{\sqrt{H^2}}, \tag{4.12}$$

and we have identified

$$H_{\mathrm{w}}(m) = \gamma_5(\nabla\!\!\!/ + \frac{1}{2}\Delta + m) \tag{4.13}$$

as the Hermitian version of the Wilson-Dirac operator, with a negative mass for H^{+} (i.e. $s > 0$) and a positive mass for H^{-} (i.e. $s < 0$). An analysis of the free-field spectrum of H_w shows[3] that it is doubler free for $m > -2$, with doublers appearing at $m = -2, -4, -6, -8$. This means that we require that $0 < m_{+} < 2$, and $m_{-} > 0$. To simplify the situation somewhat, we can take $m_{-} \to \infty$, and hence $H_{-} \to \gamma_5$. The same is not possible for H_{+}, instead we define $\hat{\gamma}_5 = \epsilon(H_{+})$.

4.2 The overlap-Dirac operator

The overlap formula for the chiral determinant leads to the following expression for the vector-like (Dirac) fermionic determinant[37, 40],

$$|\langle +|-\rangle|^2 = \det(\frac{1}{2}(1 + \gamma_5\hat{\gamma}_5)). \tag{4.14}$$

This then gives us an explicit formula for the overlap-Dirac operator (restoring a here),

$$D_{\mathrm{o}} = \frac{1}{2a}(1 + \gamma_5\hat{\gamma}_5). \tag{4.15}$$

One arrives at the above formula through the structure of the ground states $|\pm\rangle$. For simplicity, we do not reproduce this derivation here, but instead motivate D_{o} as a lattice Dirac operator with the chiral symmetry we need by showing that it is a solution to the Ginsparg-Wilson relation[43],

$$D_{\Lambda}\gamma_5 + \gamma_5 D_{\Lambda} = 2D_{\Lambda}\gamma_5 D_{\Lambda}. \tag{4.16}$$

We wish to find the class of solutions D_{Λ}, where Λ is a matrix that parameterises the solutions. Define

$$D_{\Lambda} = \frac{1}{2}(1 + \gamma_5\Lambda), \tag{4.17}$$

Then if D_{Λ} satisfies the Ginsparg-Wilson relation we must have that $\Lambda^2 = 1$.

Example 4.2.1. *Verify this.*

$$\begin{aligned}\Lambda^2 &= \gamma_5(2D_\Lambda - 1)\gamma_5(2D_\Lambda - 1)\\ &= \gamma_5(4D_\Lambda\gamma_5 D_\Lambda - 2D_\Lambda\gamma_5 - 2\gamma_5 D_\Lambda + \gamma_5)\\ &= \gamma_5^2 = 1. \quad \square\end{aligned}$$

Furthermore, if we require that D_Λ satisfies γ_5-Hermiticity, $D_\Lambda^\dagger = \gamma_5 D_\Lambda \gamma_5$, then Λ must be Hermitian. So, then our solution space to the Ginsparg-Wilson relation requires Λ to be unitary and Hermitian. Immediately we note that $\hat{\gamma}_5 = \epsilon(H^+)$ satisfies these conditions, and thus D_o is a Ginsparg-Wilson operator. Furthermore, using $H^+ = H_\text{w}(-m_+) = \gamma_5 D_\text{w}$ it is easily shown that D_o is a valid discretisation of the continuum Dirac operator.

Example 4.2.2. *Prove this.*

First, we define $m_\text{w} = am_+$ to be a dimensionless parameter. Then,

$$\begin{aligned}H_\text{w}^2(m_+) &= D_\text{w}^\dagger D_\text{w} = (-\nabla\!\!\!/ + \frac{a}{2}\Delta - \frac{m_\text{w}}{a})(\nabla\!\!\!/ + \frac{a}{2}\Delta - \frac{m_\text{w}}{a})\\ &= \frac{m_\text{w}^2}{a^2}(1 - \frac{a}{m_\text{w}}(\nabla\!\!\!/ - \frac{a}{2}\Delta))(1 + \frac{a}{m_\text{w}}(\nabla\!\!\!/ - \frac{a}{2}\Delta))\\ &= \frac{m_\text{w}^2}{a^2}(1 + O(a^2)).\end{aligned} \tag{4.18}$$

Now, $(1+x)^{-\frac{1}{2}} = 1 - \frac{1}{2}x + O(x^2)$, so

$$\begin{aligned}D_\text{o} &= \frac{1}{2a}(1 + \gamma_5\epsilon(H_\text{w}(-m_+)))\\ &= \frac{1}{2a}(1 + \gamma_5\frac{H_\text{w}(-m_+)}{\sqrt{H_\text{w}^2(-m_+)}})\\ &= \frac{1}{2a}(1 + \gamma_5\frac{\frac{1}{m_+}H_\text{w}(-m_+)}{\sqrt{\frac{1}{m_+^2}H_\text{w}^2(-m_+)}})\\ &= \frac{1}{2a}(1 - \frac{1 - \frac{a}{m_\text{w}}(\nabla\!\!\!/ + \frac{a}{2}\Delta)}{\sqrt{1 + O(a^2)}})\\ &= \frac{1}{2a}(1 - 1 + \frac{a}{m_\text{w}}(\nabla\!\!\!/ + \frac{a}{2}\Delta)(1 - O(a^2)))\\ &= \frac{1}{2m_\text{w}}\nabla\!\!\!/ + O(a)\\ &\to \frac{1}{2m_\text{w}}D\!\!\!/, \text{ as } a \to 0. \quad \square\end{aligned} \tag{4.19}$$

This simple derivation does not really give the full picture. All possible $O(a)$ error terms violate chiral symmetry, so D_o only has $O(a^2)$ errors. In the next section we will see that the lattice formulation based on the overlap construction (which we simply refer to as the overlap) reproduces all the desired continuum physics, and is thus essentially an ideal lattice regularisation of Dirac fermions. While there are other choices for Λ that also satisfy hermiticity and unitarity, we strongly stress that D_o is the *only* known explicit solution which correctly reproduces the desired physics.

4.3 Properties of the overlap

It was mentioned earlier that the obstructions to a satisfactory lattice regularisation were subtle. This is because the physics of chiral symmetry in the continuum is non-trivial, and reproducing this physics on the lattice is difficult. The triumph of the overlap is that it gives us a non-perturbative (lattice) regularisation of chiral physics that does not sacrifice any of these physical non-trivialities.

4.3.1 Exceptional Configurations and Topology

In the previous chapter we saw that in the massless case, the (vector-like) fermionic Lagrangian was invariant under the $U(1)$ symmetry group corresponding to axial rotations,

$$\psi(x) \rightarrow e^{i\alpha\gamma_5}\psi(x). \tag{4.20}$$

A famous result in the continuum is that this symmetry is anomalous, that is, the symmetry satisfied at the classical (Lagrangian) level is broken by quantum effects. Gauge field topology is connected to the axial anomaly, as Noether's theorem states the divergence of the axial current should be zero. However, axial current conservation is broken by the topological charge density,

$$\partial_\mu \mathcal{J}^\mu_A(x) \propto \frac{g^2}{32\pi}\epsilon^{\mu\nu\lambda\omega}F_{\mu\nu}(x)F_{\lambda\omega}(x) = \mathcal{Q}(x). \tag{4.21}$$

The topological charge Q is obtained by integration,

$$Q = \int d^4x\mathcal{Q}(x). \tag{4.22}$$

In the continuum (with appropriate boundary conditions) the topological charge is an integer valued functional of the gauge field. Furthermore, the fermions are sensitive to topology in this situation. The zero eigenmodes of the continuum Dirac operator have a definite chirality, that is they are also eigenmodes of γ_5. The connection between fermions and topology is manifested in the Atiyah-Singer index theorem, which equates the gauge field topological charge with the

chirality of the zero eigenmodes of the Dirac operator,

$$Q = n_- - n_+, \tag{4.23}$$

where n_- is the number of zero modes of $\not{D}$ with negative (right-handed) chirality, and n_+ is the number of eigenmodes with positive (left-handed) chirality. The space of continuum gauge fields is divided into disconnected components, and with each component we associate a single (positive or negative) value of Q.

On the lattice however, the space of gauge fields is connected (that is, any gauge field can be smoothly deformed into any other). Furthermore, the topological charge (as defined by the gauge field) is only integer valued for sufficiently smooth lattice gauge fields. This means that we must rethink what we mean by topology on the lattice.

Let us consider the Wilson dirac operator $D_{\rm w}(-m)$ with a negative mass term. Now, the Wilson fermion action (or any action which explicitly breaks chiral symmetry) suffers from additive mass renormalisation in the presence of gauge field interactions. This means that $D_{\rm w}(0)$ no longer represents massless quarks, but rather masslessness occurs at some other value of m, the critical mass, which we denote m_c. Typically we have that $0 < m_c < 2$. Furthermore, the zero eigenmodes of $H_{\rm w}(-m)$ are associated with topology. Zero modes, which must occur at $m = 0$ if we were in the continuum, may now occur at some other m, with $m_c < m < m_d$. Here, m_d is the point where doublers begin to appear. At tree-level, $m_d = 2$, but in the interacting theory, m_d gets renormalised, and typically $2 < m_d < 4$. We refer to the topological low modes in this case as zero crossings. Modes which cross zero with positive (negative) slope are associated with positive (negative) chirality, and the net number of crossings between m_c and m_d give the topological charge of the gauge field (as defined by the fermions). This definition of the topological charge is somewhat imprecise, as one could equally choose to stop counting crossings at some $m < m_d$. Furthermore, the occurence of many crossings close together can make counting difficult.

Recall from the first chapter that the quark propagator is the Green's function of the Dirac equation. The quark propagator is a key quantity on the lattice, and may be calculated non perturbatively by inverting the lattice Dirac operator. Additive mass renormalisation in this situation has an extremely undesirable effect, as the value of m_c may vary somewhat between different gauge field configurations in the same ensemble. This means that on some configurations $D_{\rm w}$ (or any non-chiral fermion operator) may have a zero eigenvalue even at positive quark mass. Such configurations are called "exceptional". Attempts to invert $D_{\rm w}$ on such configurations encounter singular behaviour. The exceptional configuration problem (at least in the quenched approximation) is worsened as one attempts to go to lighter quark masses as the fermion determininant which suppresses such singularities is absent. The numerical cost of inverting $D_{\rm w}$ increases with its condition number, which is the ratio of its largest eigenvalue to

it smallest,

$$\kappa = \frac{|\lambda_{\text{max}}|}{|\lambda_{\text{min}}|}. \tag{4.24}$$

Light quark mass simulations with non-chiral fermions are further impeded by the critical slowing down of this inversion procedure. The slowing down is caused by the occurrence of very small eigenvalues of D_{w}, even at positive quark mass. In the continuum $\not{D}$ is protected from this behaviour. Any Dirac operator which obeys γ_5-hermiticity and exact (continuum) chiral symmetry must be skew-Hermitian, which is enough to give a gap away from zero.

Example 4.3.1. *Show this.*

Proof. Let D_c be such that $D_c^\dagger = \gamma_5 D_c \gamma_5$, and $\{D_c, \gamma_5\} = 0$. Then it is immediate that $D_c = -D_c^\dagger$. Hence $(D_c + m)^\dagger (D_c + m) = D_c^\dagger D_c + m^2$, and as $M^\dagger M$ is non-negative for any matrix M, we have that $|\lambda_{\text{min}}(D_c + m)| \geq m$. □

The situation with overlap fermions is far more elegant. The (lattice deformed) chiral symmetry that the overlap enjoys is enough to eliminate additive mass renormalisation. This eliminates the usual exceptional configuration problem. Instead, a new kind of exceptional configuration arises. As mentioned above, $H_{\text{w}}(-m)$ may have zero eigenvalues for $m_c < m < m_d$. But the regulator parameter m_+ used in definition of the overlap is within this region. So for a given m_+, $H_+ = H_{\text{w}}(-m_+)$ may have zero modes in certain gauge backgrounds. On these new exceptional configurations then, $\epsilon(H_+) = H_+/\sqrt{H_+^2}$ is undefined. However, the set of configurations for which the overlap is undefined is of measure zero and is not encountered in numerical simulations.

Furthermore, the overlap's exceptional configurations are intricately connected to topology. The zero crossings of H_{w} in the region $m_c < m < m_d$ are associated with localised topological objects, with the size of the topological object decreasing as the crossing value increases[44]. Given a configuration U and a particular choice of m_+, as we change the configuration's topology by (smoothly) creating or destroying an instanton, say, then the crossing must at some stage go through our chosen value of m_+. Therefore this deformation must necessarily pass though a point where the fermionic action is undefined. In this way the space of lattice gauge fields are partitioned according to topological sectors.

As is the case with $\not{D}$, the zero eigenmodes of D_{o} are associated with topology, and have a definite chirality[45, 46].

Example 4.3.2. *Show this.*

Proof. Let $\psi \in Ker(D_o)$, that is the kernel (nullspace), $D_o\psi = 0$. Then

$$\begin{aligned} &\frac{1}{2}(1+\gamma_5\hat{\gamma}_5)\psi = 0 \\ &\Rightarrow (\gamma_5 + \hat{\gamma}_5)\psi = 0 \\ &\Rightarrow \gamma_5\psi = -\hat{\gamma}_5\psi. \end{aligned} \tag{4.25}$$

Multiply by γ_5 to obtain

$$-\psi = \gamma_5\hat{\gamma}_5\psi.$$

Then ψ is an eigenvector of $\gamma_5\hat{\gamma}_5$ with eigenvalue -1. Multiply (4.25) by $\hat{\gamma}_5$ to obtain

$$\hat{\gamma}_5\gamma_5\psi = -\psi.$$

Thus,

$$\hat{\gamma}_5\gamma_5\psi = \gamma_5\hat{\gamma}_5\psi.$$

Hence $[\gamma_5, \hat{\gamma}_5] = 0$ on the nullspace of D_o, and we can construct a basis for the nullspace in terms of their mutual eigenvectors. The same is also true of the subspace spanned by vectors with $\gamma_5\hat{\gamma}_5\psi = \psi$, that is $D_o\psi = \psi$. Both γ_5 and $\hat{\gamma}_5$ are hermitian and unitary, and thus they have eigenvalues ± 1. Let $v_{i,j}$ be an a mutual eigenvector of $\gamma_5, \hat{\gamma}_5$ with respective eigenvalue of sign $i, j \in \{+,-\}$. Construct four subspaces $V_{+,+}, V_{-,+}, V_{+,-}$ and $V_{-,-}$ spanned by the corresponding eigenvectors. Clearly all $v_{+,+}$ and $v_{-,-}$ are eigenvectors of D_o with eigenvalue 1. It is also clear that $Ker(D_o) = V_{-,+} \oplus V_{+,-}$, and all $v_{-,+}, v_{+,-}$ are zeromodes of D_o with definite chirality. Furthermore, it is straightforward to see that for every zero mode of a given chirality there is a corresponding eigenmode with eigenvalue unity of opposite chirality. □

Thus overlap fermions leads us to associate with each topological sector an integer

$$Q_F = n_- - n_+, \tag{4.26}$$

where n_- is the number of zero modes of the overlap-Dirac operator with negative (right-handed) chirality, and n_+ is the number of eigenmodes with positive (left-handed) chirality. In practice, we find that only one of $n_\pm$ is non-zero[46, 47]. In fact, $n_\pm$ counts the net number of zero crossings of H_+ between m_c and m_+ with chirality ± 1, and so Q_F is a natural definition of topological charge on the lattice (as defined by the fermions). In rough gauge backrounds, Q_F can depend upon the regulator mass m_+. Q_F may not always agree with the topogical charge as defined by the gauge field, $Q_G = \sum_{x\in\mathbb{L}} \mathcal{Q}(x)$ (which is not in general an integer). However, highly improved definitions of Q_G closely approach integer values on smooth configurations.

The massive overlap-Dirac operator $D_o(\mu)$ is given in terms of the massless one $D_o \equiv D_o(0)$ by[48]

$$D_o(\mu) = (1-\mu)D_o + \mu. \tag{4.27}$$

The mass parameter μ satisfies $|\mu| < 1$ and represents a mass of $\frac{\mu}{1-\mu}$. A major advantage of overlap fermions is that they do not suffer from critical slowing down as one attempts to invert $D_o(\mu)$ at light quark masses. This is because the exact (lattice) chiral symmetry the overlap possesses is enough to provide a gap away from zero for any positive quark mass[49].

Example 4.3.3. *Show this*

Proof. The Ginsparg-Wilson relation implies

$$D_o^\dagger + D_o = 2D_o^\dagger D_o. \tag{4.28}$$

Then consider

$$\begin{aligned} D_o^\dagger(\mu)D_o(\mu) &= (1-\mu)^2(D_o^\dagger + \frac{\mu}{1-\mu})(D_o + \frac{\mu}{1-\mu}) \\ &= (1-\mu)^2(D_o^\dagger D_o + \frac{\mu}{1-\mu}(D_o^\dagger + D_o) + \frac{\mu^2}{(1-\mu)^2}) \\ &= (1-\mu)^2((1+\frac{2\mu}{1-\mu})D_o^\dagger D_o + \frac{\mu^2}{(1-\mu)^2}) \\ &= (1-\mu^2)D_o^\dagger D_o + \mu^2. \end{aligned} \tag{4.29}$$

As $D_o^\dagger D_o \geq 0$ is non negative then we have that for $\mu > 0$ that $|\lambda_{\min}(D_o(\mu))| \geq \mu$. □

4.3.2 Locality

A lattice operator D_a is said to be ultra-local if

$$\exists r \geq 0 \text{ such that } |D_a(x,y)| = 0 \quad \forall |x-y| > r. \tag{4.30}$$

It is easily seen that the Wilson-Dirac operator is ultra-local. The overlap-Dirac operator is not ultra-local. For a lattice operator to be local in the continuum limit, it is sufficient for it to satisfy exponential locality. We say that D_a is (exponentially) local, if

$$\exists r, c, A \geq 0 \text{ such that } |D_a(x,y)| < Ae^{-cr} \quad \forall |x-y| > r. \tag{4.31}$$

It is not immediately obvious that D_o is local, as the inverse square root present in the sign function could be a potential source of non-locality. However, if the spectrum of $H_w^2(-m)$ is bounded and the lower bound is well separated from zero, with the exception of isolated low-lying modes, then it can be shown that the overlap Dirac operator is local[50]. It is therefore natural to be interested in finding bounds on H_w, and both upper and lower bounds have been derived[50, 3]. As all lattice operators are bounded from above, it is the lower bound which

is of chief interest. The lower bounds on H_w were derived in the presence of a plaquette smoothness condition,

$$||1 - U_{\mu\nu}(x)|| < \epsilon_{\mu\nu} \quad \forall\ x. \tag{4.32}$$

It is not unreasonable to expect such a condition to hold near the continuum limit. In particular, the lower bounds derived in [50, 3] state that for small enough ϵ the spectrum of $H_w(-m)$ has a gap away from zero, near $m = 1$. We will be interested in constructing similar bounds for a more complicated operator than H_w later, so it will be instructive to first review the derivation for the simpler case. The bounds in [3] allow for larger ϵ, so we will represent the derivation given there.

Example 4.3.4. *Assuming $||1 - U_{\mu\nu}(x)|| < \epsilon\ \forall\ x, \mu, \nu$, derive a lower bound on the spectrum of $H_w(-m)$.*

Proof. First we need some identities.

$$\begin{aligned}
\not{\nabla}^2 &= \gamma_\mu\gamma_\nu\nabla_\mu\nabla_\nu \\
&= \nabla_\mu\nabla_\mu + \frac{1}{4}\sum_{\mu\neq\nu}([\gamma_\mu,\gamma_\nu] + \{\gamma_\mu,\gamma_\nu\})([\nabla_\mu,\nabla_\nu] + \{\nabla_\mu,\nabla_\nu\}) \\
&= \nabla_\mu\nabla_\mu + \frac{1}{4}\sum_{\mu\neq\nu}[\gamma_\mu,\gamma_\nu][\nabla_\mu,\nabla_\nu] \ (\text{as } \{\gamma_\mu,\gamma_\nu\} = 0) \\
&= \nabla\cdot\nabla + \frac{1}{4}[\gamma,\gamma]\cdot[\nabla,\nabla]
\end{aligned} \tag{4.33}$$

Define $\Delta_\mu = 2 - T_\mu - T_\mu^\dagger$, thus $\Delta = \sum_\mu \Delta_\mu$. Then

$$\begin{aligned}
\Delta^2 &= \sum_{\mu,\nu}(2 - T_\mu - T_\mu^\dagger)(2 - T_\nu - T_\nu^\dagger) \\
&= \sum_{\mu}(2 - T_\mu - T_\mu^\dagger)(2 - T_\mu - T_\mu^\dagger) + \sum_{\mu\neq\nu}(2 - T_\mu - T_\mu^\dagger)(2 - T_\nu - T_\nu^\dagger) \\
&= \sum_{\mu}(4 - 4T_\mu - 4T_\mu^\dagger + T_\mu^2 + T_\mu^{\dagger 2} + 2) + \sum_{\mu\neq\nu}\Delta_\mu\Delta_\nu \\
&= \sum_{\mu}(8 - 4T_\mu - 4T_\mu^\dagger + (T_\mu - T_\mu^\dagger)^2) + \sum_{\mu\neq\nu}\Delta_\mu\Delta_\nu \\
&= 4\Delta + 4\nabla\cdot\nabla + \sum_{\mu\neq\nu}\Delta_\mu\Delta_\nu
\end{aligned} \tag{4.34}$$

So,

$$-\not{\nabla}^2 + \frac{1}{4}\Delta^2 = \Delta + \sum_{\mu\neq\nu}\Delta_\mu\Delta_\nu - \frac{1}{4}[\gamma,\gamma]\cdot[\nabla,\nabla]. \tag{4.35}$$

Then, choosing $m = 1$,

$$H_{\mathrm{w}}^2(-1) = D_{\mathrm{w}}^\dagger(-1)D_{\mathrm{w}}(-1) = (-\nabla\!\!\!/ + \frac{1}{2}\Delta - 1)(\nabla\!\!\!/ + \frac{1}{2}\Delta - 1)$$
$$= 1 - \nabla\!\!\!/^2 + \frac{1}{4}\Delta^2 - \Delta - \frac{1}{2}[\nabla\!\!\!/, \Delta]$$
$$= 1 + \sum_{\mu\neq\nu} \Delta_\mu\Delta_\nu - \frac{1}{4}[\gamma,\gamma]\cdot[\nabla,\nabla] - \frac{1}{2}[\nabla\!\!\!/, \Delta]. \tag{4.36}$$

Now, define $\nabla_\mu^+ = T_\mu - 1$ and $\nabla_\mu^- = -\nabla_\mu^{+\dagger}$. Then

$$\Delta_\mu\Delta_\nu = (2 - T_\mu - T_\mu^\dagger)(2 - T_\nu - T_\nu^\dagger)$$
$$= (1 - T_\mu)(1 - T_\mu^\dagger)(1 - T_\nu)(1 - T_\nu^\dagger)$$
$$= \frac{1}{2}(\nabla_\mu^+\nabla_\mu^-\nabla_\nu^+\nabla_\nu^- + \nabla_\mu^-\nabla_\mu^+\nabla_\nu^+\nabla_\nu^-)$$
$$= W_{\mu\nu} + X_{\mu\nu}, \tag{4.37}$$

where

$$W_{\mu\nu} = \frac{1}{2}\big((\nabla_\mu^+\nabla_\nu^+)(\nabla_\mu^+\nabla_\nu^+)^\dagger + (\nabla_\mu^-\nabla_\nu^+)(\nabla_\mu^-\nabla_\nu^+)^\dagger\big) \tag{4.38}$$

is non-negative and

$$X_{\mu\nu} = \frac{1}{2}(\nabla_\mu^+[\nabla_\mu^-, \nabla_\nu^+\nabla_\nu^-] + \nabla_\mu^-[\nabla_\mu^+, \nabla_\nu^+\nabla_\nu^-])$$
$$= \frac{1}{2}\big((1 - T_\mu)[T_\mu^\dagger, T_\nu + T_\nu^\dagger] + (1 - T_\mu^\dagger)[T_\mu, T_\nu + T_\nu^\dagger]\big). \tag{4.39}$$

Let $X = \frac{1}{8}\sum_{\mu\neq\nu} X_{\mu\nu}$, thus

$$X = \frac{1}{8}\sum_{\mu\neq\nu}[T_\mu + T_\mu^\dagger, T_\nu + T_\nu^\dagger] - T_\mu[T_\mu^\dagger, T_\nu + T_\nu^\dagger] - T_\mu^\dagger[T_\mu, T_\nu + T_\nu^\dagger]$$
$$= -\frac{1}{8}\sum_{\mu\neq\nu} T_\mu[T_\mu^\dagger, T_\nu + T_\nu^\dagger] + T_\mu^\dagger[T_\mu, T_\nu + T_\nu^\dagger] \text{ (by anti-symmetry)}. \tag{4.40}$$

Now, set $Y = \frac{1}{2}[\nabla\!\!\!/, \Delta]$. Then

$$Y = -\frac{1}{4}\sum_{\mu\neq\nu}\gamma_\mu[T_\mu - T_\mu^\dagger, T_\nu + T_\nu^\dagger]$$
$$= -\frac{1}{8}\sum_{\mu\neq\nu}\gamma_\mu[T_\mu - T_\mu^\dagger, T_\nu + T_\nu^\dagger] + \gamma_\nu[T_\nu - T_\nu^\dagger, T_\mu + T_\mu^\dagger]$$
$$= -\frac{1}{8}\sum_{\mu\neq\nu}(\gamma_\mu - \gamma_\nu)([T_\mu, T_\nu] + [T_\nu^\dagger, T_\mu^\dagger]) + (\gamma_\mu + \gamma_\nu)([T_\mu, T_\nu^\dagger] + [T_\nu, T_\mu^\dagger]). \tag{4.41}$$

Finally, letting $Z = -\frac{1}{4}[\gamma,\gamma]\cdot[\nabla,\nabla]$, we have

$$Z = -\frac{1}{8}\sum_{\mu\neq\nu}\gamma_\mu\gamma_\nu(\frac{1}{4}[T_\mu - T_\mu^\dagger, T_\nu - T_\nu^\dagger]). \tag{4.42}$$

So X, Y, Z have been written in terms of commutators between parallel transport operators. Now,

$$\begin{aligned}[T_\mu, T_\nu][T_\mu, T_\nu]^\dagger &= (T_\mu T_\nu - T_\nu T_\mu)(T_\nu^\dagger T_\mu^\dagger - T_\mu^\dagger T_\nu^\dagger) \\ &= 2 - T_\mu T_\nu T_\mu^\dagger T_\nu^\dagger - T_\nu T_\mu T_\nu^\dagger T_\mu^\dagger \\ &= (1 - P_{\mu\nu})(1 - P_{\mu\nu}^\dagger), \end{aligned} \tag{4.43}$$

where $P_{\mu\nu}\psi(x) = U_{\mu\nu}(x)\psi(x)$ is the sitewise plaquette operator. Under our assumption of

$$||1 - U_{\mu\nu}(x)|| < \epsilon \;\forall\; x, \mu, \nu \tag{4.44}$$

we then have

$$||[T_\mu, T_\nu]|| < \epsilon. \tag{4.45}$$

The same inequality holds if one or both of T_μ, T_ν is replaced by their hermitian conjugate. As $W_{\mu\nu}$ is positive semi-definite, eq. (4.36) implies

$$\lambda_{\min}(H_{\mathrm{w}}^2(-1)) \geq 1 - ||X|| - ||Y|| - ||Z||. \tag{4.46}$$

Noting that for $\mu \neq \nu$, $||\gamma_\mu \pm \gamma_\nu|| = \sqrt{2}$, and that T_μ is unitary we obtain

$$||X|| \leq 6\epsilon, \qquad ||Y|| \leq 6\sqrt{2}\epsilon, \qquad ||Z|| \leq 6\epsilon. \tag{4.47}$$

Thus,

$$\lambda_{\min}(H_{\mathrm{w}}^2(-1)) \geq 1 - 6(2+\sqrt{2})\epsilon. \tag{4.48}$$

This bound is only meaningful when $\epsilon < \frac{1}{6(2+\sqrt{2})}$. Now, we have that $\frac{\partial H_{\mathrm{w}}}{\partial m} = \gamma_5$ which gives us the flow inequality for the eigenvalues of H_{w},

$$|\frac{\partial\lambda}{\partial m}| \leq 1, \tag{4.49}$$

and thus

$$\lambda_{\min}(|H_{\mathrm{w}}(-m)|) \geq \sqrt{1 - 6(2+\sqrt{2})\epsilon} - |1-m|. \tag{4.50}$$

□

This completes our review of the background material necessary for the main body of this work. Although the overlap has ameliorated the numerical cost of going to light quark masses, a new computational bottleneck appears. In the next chapter we will see that in numerical simulations the evaluation of the matrix sign function incurs significant computational expense. It will be the focus of our first investigation to reduce the computational cost of the sign function, thereby speeding up the evaluation of the overlap Dirac operator.

Chapter 5

Accelerated Overlap Fermions

In the last chapter we saw that the massless overlap-Dirac operator was given by

$$D_o = \frac{1}{2}\big(1 + \gamma_5 \epsilon(H)\big). \tag{5.1}$$

The argument H to the sign function $\epsilon(H)$ is referred to as the overlap kernel and is usually taken to the be the hermitian Wilson-Dirac operator, H_w. However, any valid discretisation of the Dirac operator that represents a fermion of negative mass and is doubler free could equally well be used. In this chapter we turn to the question of practical implementations of the overlap, and investigate whether alternative choices for H can reduce the numerical cost of simulating overlap fermions.

5.1 Implementation

The matrix sign function is defined through the spectrum of its argument,

$$\epsilon(H) = \sum_\lambda |\lambda\rangle \epsilon(\lambda) \langle\lambda|, \tag{5.2}$$

where $\epsilon(\lambda) = \lambda/\sqrt{\lambda^2}$. For small to moderate size matrices, this suggests a means to implement the sign function, by diagonalising H, calculating the above and then returning to the original basis via a unitary transformation. Unfortunately, for lattice QCD in four dimensions, matrices such as H_w are too large for this to be feasible.

Instead, one can use approximations to the (matrix) sign function. In lattice simulations we only need the action of the Dirac operator on a vector. This allows us to take advantage of the fact that H_w is a sparse matrix, so we do not need to store it in its entirety. For this reason we prefer approximations that also only need the action of H_w on a vector. Both polynomial and rational polynomial approximations satisfy this criterion. Due to the discontinuity at

the origin, polynomial approximations to the sign function are not particularly efficient. Studies indicate that rational polynomial approximations are the best choice[46].

The choice of rational polynomial is not unique. The first choice that was studied in the context of the overlap was the so-called polar decomposition[51],

$$\epsilon(x) \approx \epsilon_{\rm pd}^{(n)}(x) = x \sum_{k=1}^{n} \frac{b_k}{x^2 + c_k}, \tag{5.3}$$

where the coefficients for order n are

$$b_k = \frac{1}{n \cos^2(\frac{\pi}{2n}(k-\frac{1}{2}))}, \qquad c_k = \tan^2(\frac{\pi}{2n}(k-\frac{1}{2})). \tag{5.4}$$

As the sign function is scale-invariant, we can rescale H by a factor η to get the best approximation. The approximation is good over a finite range of $x, |x| \in [x_{\rm min}, x_{\rm max}]$. We want the spectrum of H to lie within this range, $x_{\rm min} \leq \eta\lambda(H) \leq x_{\rm max}$. Let $|\lambda(H)|$ be bounded by $\lambda_{\rm min}, \lambda_{\rm max}$. For the polar decomposition the optimal choice is $\eta = \frac{1}{\sqrt{\lambda_{\rm min}\lambda_{\rm max}}}$. Let

$$\kappa = \frac{\lambda_{\rm max}}{\lambda_{\rm min}} \tag{5.5}$$

be the condition number of H. Then for a given accuracy

$$\delta = \max_{\lambda \in [\lambda_{\rm min}, \lambda_{\rm max}]} |\epsilon(\lambda) - \epsilon_{\rm pd}^{(n)}(\lambda)| \tag{5.6}$$

the required approximation order is

$$n = \frac{1}{4}\sqrt{\kappa}\ln(\frac{\delta}{2}). \tag{5.7}$$

That is, as the condition number of H increases, the order of our approximation must increase to maintain a fixed accuracy.

The second rational approximation that has been used is the Zolotarev or optimal rational polynomial[52, 46]. It has the form

$$\epsilon(x) \approx \epsilon_{\rm z}^{(n)}(x) = d_0 x(x^2 + c_{2n}) \sum_{l=1}^{n} \frac{b_l}{x^2 + c_{2l-1}}, \tag{5.8}$$

where the coefficents are given in [52] and depend upon the condition number κ and the order of the polynomial. The advantage of the Zolotarev approximation is that it is optimal in the minimax (Chebyshev) sense, that is, for a given n and κ it gives the minimal δ. Furthermore, the approximation is uniformly good over the applicable range $x_{\rm min} = 1, x_{\rm max} = \kappa$, with $2n + 2$ points at which

the approximation is exact and the maximum deviations between these points oscillating in sign and equal in magnitude. Naturally, we can rescale H in this case also, with $\eta = 1/\lambda_{\text{min}}$.

As matrix functions, both approximations are implemented in the same fashion. The action of the sign function on a vector

$$\chi = \epsilon(H)\psi \tag{5.9}$$

is evaluated by first using an iterative conjugate gradient (CG) solver for the n linear systems

$$(H^2 + c_l)\phi_l = \psi, \tag{5.10}$$

which gives us

$$\phi_l = \frac{1}{H^2 + c_l}\psi. \tag{5.11}$$

As the systems are related by a simple linear shift, one can use a multi-shift CG solver[53] which simultaneously solves all the systems for basically the cost of solving the hardest system (the smallest shift). The weighted sum over poles $\phi = \sum_l b_l \phi_l$ can either be taken afterwards or simultaneously with the CG solve. Evaluating χ and subsequently $D_o\psi$ is then straightforward. As each iteration of the CG solver requires evaluating the action of H^2 on a vector, it is by far the major overhead in evaluating the sign function and hence the overlap. We therefore seek ways to reduce the cost of this step.

As with all conjugate gradient routines, the number of iterations required to reach a solution for a given precision depends upon the condition number κ of the matrix being inverted. Considering for the moment $H_w(-m)$ with $0 < m < 2$, in typical equilibrium background gauge field configurations. Using the bounds from [3], the upper bound $\lambda_{\text{max}} \leq 8 - m$ is typically saturated, and the lower bound is zero. While the set of expectional configurations for the overlap $\lambda_{\text{min}} = 0$ is of measure zero, and is not encountered in simulations, practical experience suggets λ_{min} is often as small as 10^{-8}. This results in an unacceptably large value for the condition number $\kappa(H)$. There is a way to get around this problem though [54]. The typical spectrum of H_w is characterised by a handful of isolated low-lying eigenmodes, so one can project these out and deal with them explicitly. The condition number for the remaining part of the spectrum is then small enough that evaluating the sign function approximation becomes feasible. In practical simulations, after projecting out the isolated low-lying modes, $\epsilon^{(n)}(H_w)$ takes roughly speaking $O(100-300)$ iterations to converge, meaning that overlap fermions with the standard H_w are about $\mathcal{O}(200 - 600)$ times more expensive than standard Wilson fermions[1].

[1]This does not take into account any savings at light quark masses due to the absence of the critical slowing down problem. However, light quark masses require large volumes, and the number of CG iterations required also increases with lattice volume, so overlap simulations are still relatively expensive regardless.

Obviously it is desirable to improve upon this situation in order to make simulations with overlap fermions more feasible. The overlap approach does not specifically require the use of $H_{\rm w}$. Rather any lattice Dirac fermion operator with a large negative mass that is free of doublers can be used. Therefore we investigate ways to speed up the overlap by modifying the overlap kernel so that its spectral properties are improved. The improvements we seek are twofold: (i) An upward shift in the magnitude of the low-lying eigenvalues of $H_{\rm w}$ so as to decrease the condition number, and (ii) a reduction in the *density* of low-lying eigenvalues, so as to make the projection method of Ref. [54] more efficient. Furthermore, our aim is to produce an implementation of the overlap formalism that will perform efficiently on large-scale parallel computing architectures. On such architectures, the cost of internode communication is typically high compared to the cost of intranode computation. We therefore demand that our candidate H be no less sparse than the Hermitian Wilson-Dirac operator. That is, it possesses nearest neighbour couplings at most.

5.2 Fermion actions

Recall from the previous chapter the form of the lower bound we derived,

$$||H_{\rm w}^2(-1)|| \geq 1 - \frac{1}{2}||[\Delta, \not\nabla]|| - \frac{1}{4}||[\gamma,\gamma]\cdot[\nabla,\nabla]|| - \frac{1}{4}||\sum_{\mu\neq\nu}\Delta_\mu\Delta_\nu||. \qquad (5.12)$$

Noting the presence of the commutator term $[\Delta, \not\nabla]$ we seek to improve our lower bound by replacing the Wilson term with something that commutes with $\not\nabla$. If we consider

$$H_{\rm cw}(-m) = \gamma_5(\not\nabla - \frac{1}{2}\not\nabla^2 - m), \qquad (5.13)$$

then $H^2_{\rm cw}(-m)$ has a qualitative lower bound of

$$||H^2_{\rm cw}(m)|| \;=\; ||m^2 - (1-m)\not\nabla^2 + \frac{1}{4}\not\nabla^4|| \qquad (5.14)$$

$$\geq\; m^2 - ||(1-m)\not\nabla^2||. \qquad (5.15)$$

At $m = 1$ this is bounded from below by 1, which is clearly better than $H^2_{\rm cw}(-1)$. As the spectral flow depends smoothly on m, by the flow inequality ($|\frac{d\lambda}{dm}| \leq 1$) we expect that $H_{\rm cw}(m)$ will possess a better low-lying spectrum than $H_{\rm w}(-m)$ for values of m around 1. However, $H_{\rm cw}(-m)$ possesses doublers, so we modify it. Noting once again that

$$\not\nabla^2 = \nabla\cdot\nabla + \frac{1}{4}[\gamma^\mu,\gamma^\nu][\nabla_\mu,\nabla_\nu], \qquad (5.16)$$

and $\lim_{a\to 0}\nabla\cdot\nabla = \lim_{a\to 0} -\Delta$, we make the replacement

$$-\frac{1}{2}\not\nabla^2 \to \frac{1}{2}\Big(\Delta - \frac{1}{4}[\gamma,\gamma]\cdot[\nabla,\nabla]\Big), \qquad (5.17)$$

which eliminates the doublers due to the presence of the Wilson term. To maintain the sparsity of our operator we must still make a further modification as $[\nabla_\mu, \nabla_\nu]$ has diagonal couplings. Recall from §2.3.2 that (in lattice units)

$$[\nabla_\mu, \nabla_\nu]\psi(x) \equiv iF^{\rm cl}_{\mu\nu}(x)\psi(x) + \mathcal{O}(a^2), \tag{5.18}$$

and hence

$$\frac{1}{4}[\gamma,\gamma]\cdot[\nabla,\nabla]\psi(x) = \frac{1}{2}\sigma\cdot F^{\rm cl}_{\mu\nu}(x)\psi(x) + O(a^2). \tag{5.19}$$

Thus we have essentially "derived" the clover term (with tree-level coefficient $c_{\rm sw} = 1$) in an effort to improve our low-lying spectrum. The use of a clover-improved fermion action (with either tree-level or perturbatively (mean field) improved coefficient) therefore seems like a sensible alternative for the overlap kernel.

The quantitative bounds in Eq. (4.48) imply that the smoother the gauge field, the better the lower bound. This then suggests the use of APE-smeared links is likely to improve the low-lying spectrum, although as discussed in §3.5.1 we prefer to do this only in the irrelevant operators of the fermion action, in order to preserve short distance physics[3, 28].

Motivated by the preceding discussion, we wish to compare the low-lying spectra of the Hermitian Wilson-Dirac operator (with negative mass),

$$H_{\rm w} = \gamma_5\big(\nabla\!\!\!/ + \frac{1}{2}\Delta - m\big), \tag{5.20}$$

with the following variants. First, the standard (one loop) clover-improved fermion action, both with and without mean field improvement,

$$H_{\rm cl} = \gamma_5\big(\nabla\!\!\!/ + \frac{1}{2}(\Delta - \frac{1}{2}\sigma\cdot F) - m\big), \tag{5.21}$$

$$H_{\rm mficl} = \gamma_5\big(\nabla\!\!\!/_{\rm mfi} + \frac{1}{2}(\Delta_{\rm mfi} - \frac{1}{2}\sigma\cdot F_{\rm mfi}) - m\big). \tag{5.22}$$

Secondly, we have the Wilson-Dirac operator, with fat link irrelevant terms,

$$H_{\rm flw} = \gamma_5\big(\nabla\!\!\!/ + \frac{1}{2}\Delta^{\rm fl} - m\big). \tag{5.23}$$

Finally, we have the clover-improved fermion actions, with fat link irrelevant terms, both with and without mean field improvement,

$$H_{\rm flcl} = \gamma_5\big(\nabla\!\!\!/ + \frac{1}{2}(\Delta^{\rm fl} - \frac{1}{2}\sigma\cdot F^{\rm fl}) - m\big), \tag{5.24}$$

$$H_{\rm flic} = \gamma_5\big(\nabla\!\!\!/_{\rm mfi} + \frac{1}{2}(\Delta^{\rm fl}_{\rm mfi} - \frac{1}{2}\sigma\cdot F^{\rm fl}_{\rm mfi}) - m\big). \tag{5.25}$$

The clover action with both fat links and mean field improvement is what we defined in §3.5.2 as the FLIC action. If followed by a number (e.g. FLIC12) this denotes the number of APE-smearing sweeps (at $\alpha = 0.7$) used in the action.

We note that in the case of $H_{\rm w}$ and $H_{\rm flw}$, mean-field improvement has little effect, entering in only as a single power in both cases. For these actions, mean field improvement only changes the effective value of m (for $H_{\rm flw}$ the effective value of the Wilson parameter r is also changed slightly), so we do not consider these variants.

Before proceeding to the numerical results it is worth pointing out that the previous analytical results on the locality [50] and continuum limit of the axial anomaly [55, 56, 57, 58] and index [59] of the overlap Dirac operator continue to hold when $H_{\rm w}$ is replaced by any of the variants given above in the overlap formula. In the case of the axial anomaly and index, this is essentially because the leading order term in the expansion of commutators of the covariant finite difference operators in powers of the lattice spacing is unchanged, and the variants of $H_{\rm w}$ all coincide with $H_{\rm w}$ in the free field case.

We also mention that more general variants of the overlap Dirac operator have been considered where one starts with an approximate solution to the Ginsparg-Wilson relation and then gets an exact solution by substituting into the overlap formula [60, 61, 62, 63, 64]. While these variants can have improved properties, such as locality, they generally have hypercubic rather than nearest neighbour couplings and are therefore not considered here.

5.3 Spectral Flow Comparison

In order to test the merits of each of our proposed actions, we first calculate the spectral flow of each of them to see if our reasoning regarding their low-lying spectra is valid. From the quadratic form of the lower bounds as a function of m, and based upon results given in Ref. [65], we expect there to be some peak value of m for which the gap around zero is the largest. We calculated the flow of the lowest 15 eigenvalues as a function of m for an ensemble of 10 mean-field improved Symanzik configurations at $\beta = 4.38$ and size $8^3 \times 16$. In particular, we use a 2-loop tadpole-improved Luscher-Wiesz action. The following flow graphs allow us to see the m value for the biggest gap, and also allow us to compare the different actions. As we are interested in the magnitude of the low-lying values rather than their sign, we plot $|\lambda|$ vs m.

We begin by examining the flow of the Wilson and clover action in Figure 5.1. We see the Wilson spectrum is very poor, with a high density of very small eigenmodes and no gap away from zero. The addition of the clover term (at $c_{\rm sw} = 1$) provides some improvement, shifting the flow upwards and moving the peak values towards $m = 1$ as expected. The presence of many small eigenmodes persists however, although their density is clearly reduced.

In Figure 5.2 we examine the MFI clover and fat link Wilson actions. Mean field improvement assists the basic clover action somewhat, spreading the spectrum upwards, although the lowest modes are not raised significantly. The mass

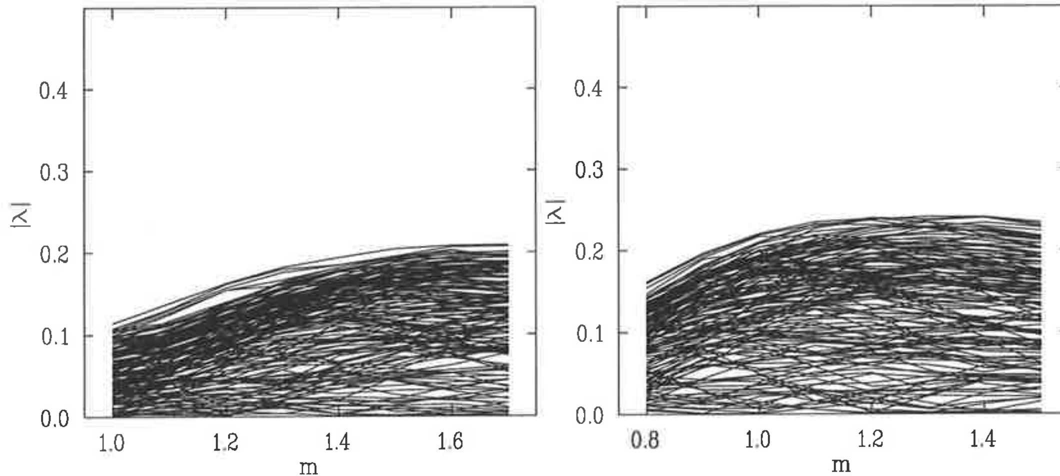

Figure 5.1: Spectral flow of the Wilson action (left) and the clover action (right) at $\beta = 4.38$.

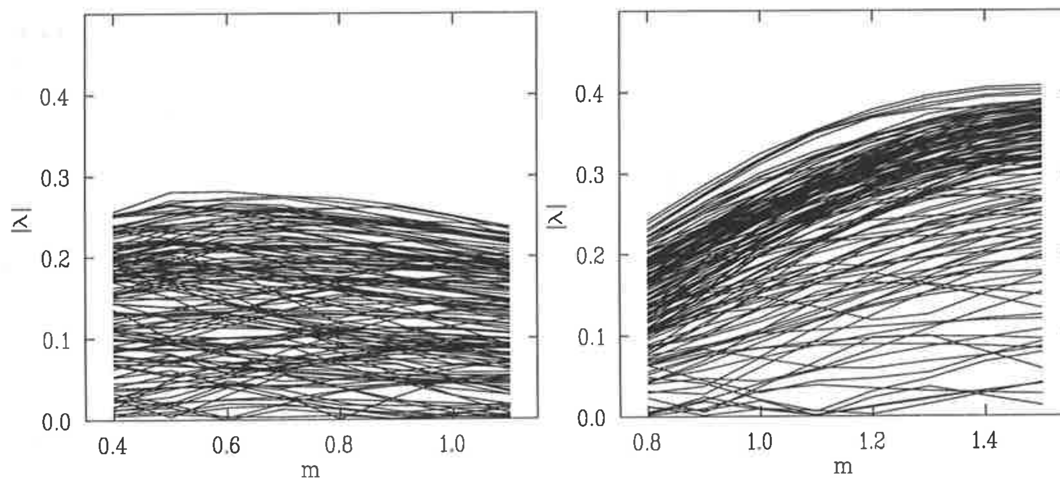

Figure 5.2: Spectral flow of the MFI clover action (left) and the fat link Wilson action (right) at $\beta = 4.38$.

value at which the low-lying density is minimised has moved significantly away from $m = 1.2$ to around $m = 0.6$. As mentioned earlier, essentially all MFI does in this case is to change the value of c_{sw} to $1.0/u_0^3$, pushing it towards its non-perturbative value. Modifying the Wilson action by smearing the irrelevant operators provides a considerable improvement. While there are still some small modes present, their density has been greatly reduced, and the spectral flow now has a clear division between the isolated low-lying modes and the modes where the spectral density becomes high which are well separated from zero. Smearing was performed with $\alpha = 0.7$ and $n_{\rm ape} = 12$ smearing sweeps.

Results for the fat link clover and FLIC12 actions are shown in Figure 5.3. The spectral flow of the fat link clover action clearly demonstrates the superiority

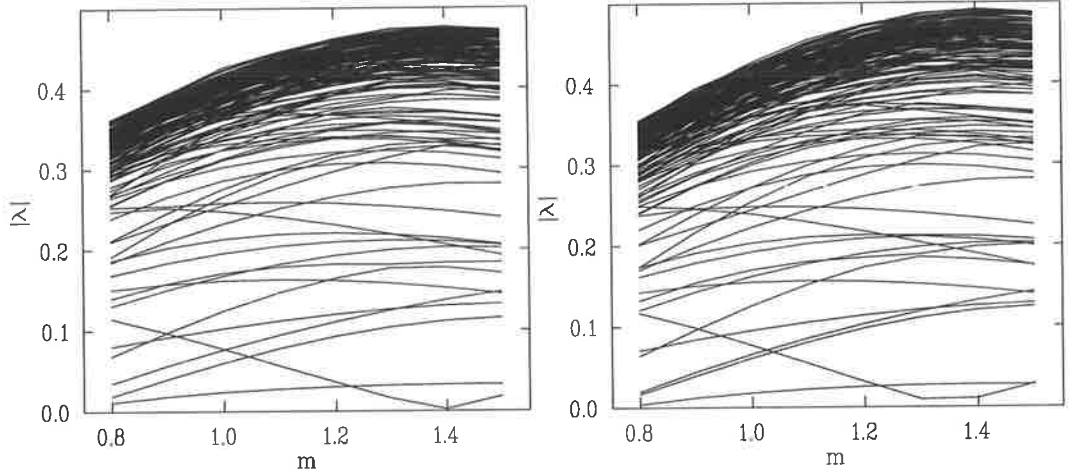

Figure 5.3: Spectral flow of the fat link clover action (left) and FLIC12 action (right) at $\beta = 4.38$.

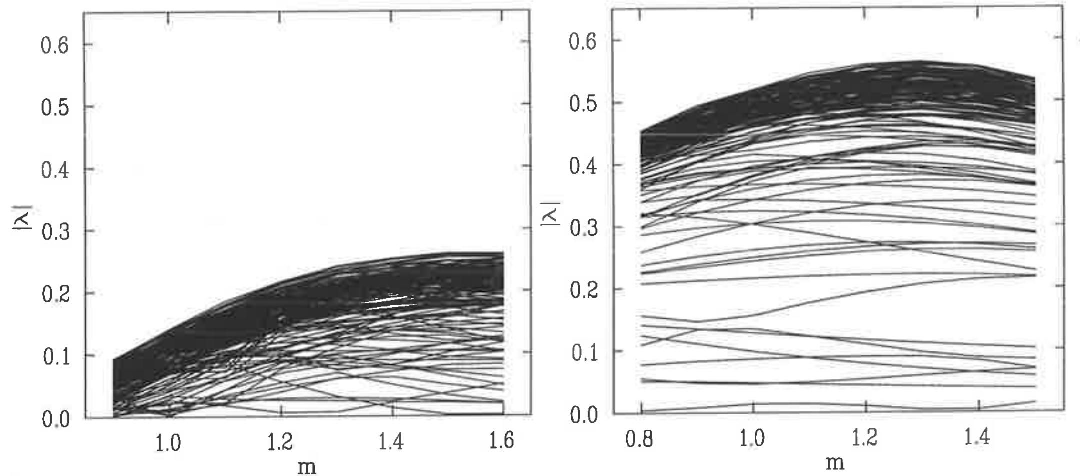

Figure 5.4: Spectral flow of the Wilson action (left) and FLIC4 action (right) at $\beta = 4.60$.

of clover-improved actions. The gap around zero is enhanced again over the fat link Wilson action, and the number of isolated low-lying modes is significantly reduced. As the fat links are already close to unity, the addition of mean field improvement only affects the fat link clover flow slightly, raising the gap around zero a little and spreading the eigenvalues upwards slightly also. The low-lying density is again very good in this case and far superior to that of the Wilson action.

To confirm our results we choose the Wilson action as a "baseline" and compare it against the FLIC action (the best of the alternative actions) on a larger, finer lattice, $12^3 \times 24$ at $\beta = 4.60$. This time we only use 4 smearing sweeps in the FLIC action since FLIC4 has less fattening and is the choice used

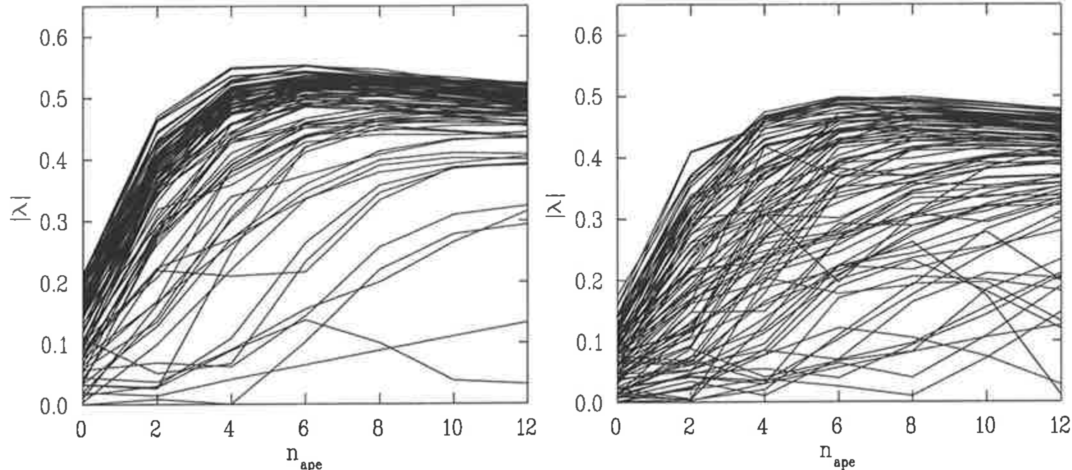

Figure 5.5: Dependence of the FLIC spectrum at $\beta = 4.60, m = 1.35$ (left) and $\beta = 4.38, m = 1.45$ (right) on the number of APE smearing sweeps.

in actual simulations[28]. We see that the Wilson action benefits significantly from the smaller lattice spacing, as there is now a visible separation from zero before the modes become dense. The FLIC action has the same characteristics as on the coarser lattice, but it now has a peak separation of the dense modes from zero of around 0.45!

Additionally, we tested the dependence of the FLIC action upon the amount of smearing done. As stated in [23], we only effectively need to vary the product αn_{ape}, so we fix α at 0.7 and vary n_{ape} between 0 and 12. We observe that the initial 4-6 sweeps have a significant effect, but past 6 sweeps the effect is marginal, with the low lying density remaining roughly constant and the eigenvalues being compressed very slightly downwards.

5.4 Condition Number Analysis

Having obtained some understanding of the low-lying spectra of the various actions via the flow diagrams, we now turn to quantitative comparisons. Firstly we examine the condition number, κ, of the different actions as a function of m. We show below the condition numbers having projected out the lowest 5 eigenmodes and the lowest 15 eigenmodes on the 2 lattices that we used. The points are the mean condition numbers across the ensembles, and the error bars indicate the minimum and maximum condition numbers, giving an idea of the variation in κ. The smeared irrelevant-term actions here used 12 APE sweeps at $\alpha = 0.7$ for the coarse lattice and 4 sweeps for the fine lattice. Some points are offset horizontally for clarity.

Two things are immediately noticeable. Firstly, the smeared irrelevant-term actions are much better conditioned than the unsmeared actions, and secondly,

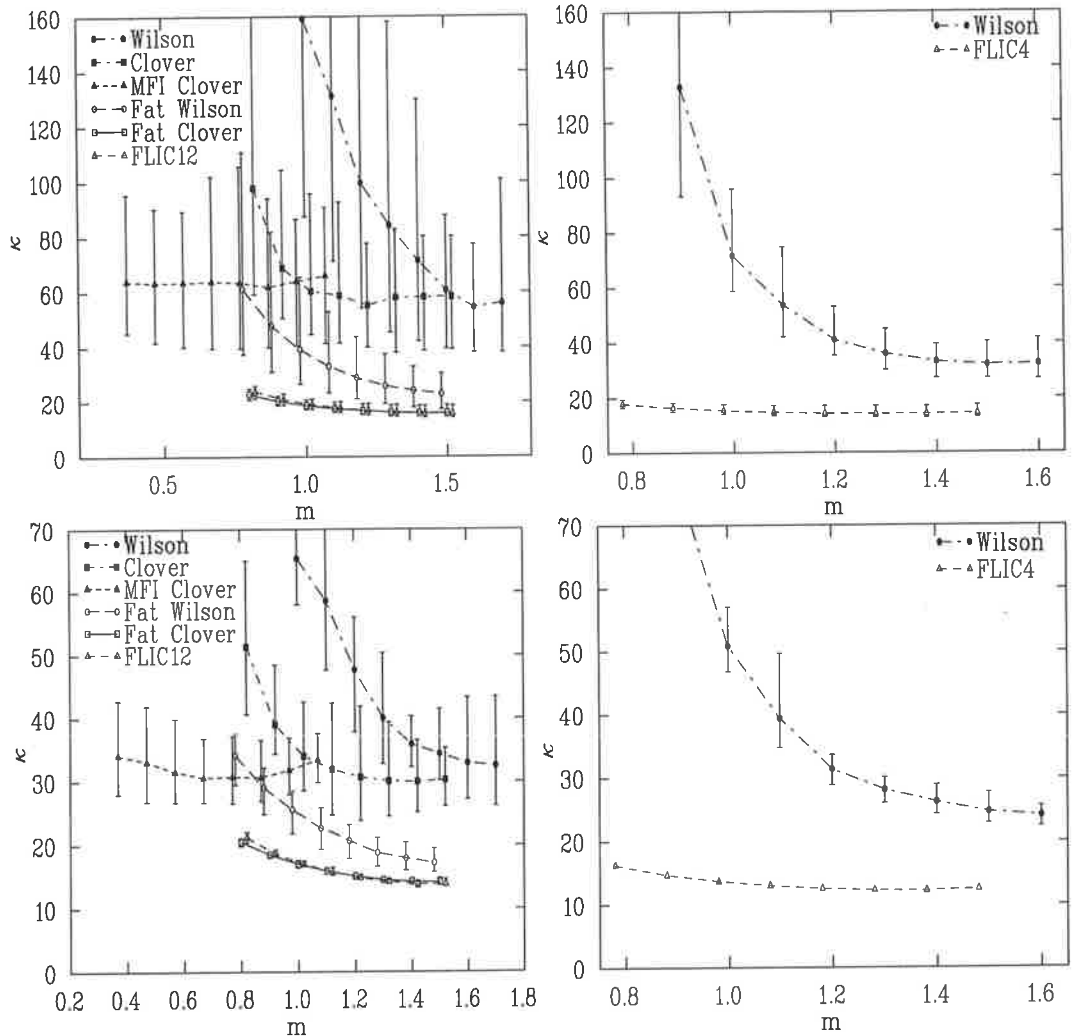

Figure 5.6: Condition numbers of the various actions. (Top-left) Results for $\beta = 4.38$ with 5 projected modes. (Bottom-left) Results for $\beta = 4.38$ with 15 projected modes. (Top-right) Results for $\beta = 4.60$ with 5 projected modes. (Bottom-right) Results for $\beta = 4.60$ with 15 projected modes.

the variation of κ between configurations is less. It should be noted that the variation (error bars) are displayed for all actions, but are smaller than the plot symbol at some points of the fat clover and FLIC lines. Projecting out an additional 10 eigenvalues has a significant effect on the unsmeared actions, but relatively little effect on the smeared actions due to reduction in the number of isolated low-lying values. In terms of condition number, the fat clover and FLIC actions are clearly and significantly superior to the other actions, with the FLIC action possessing a (slight) edge over the fat clover which arises from the mean field improvement.

As the clover term is quite fast to evaluate, we discard the fat Wilson as a

candidate action at this point as it is the least well-conditioned of the smeared actions. Given the similarity between the clover-improved actions with and without mean-field improvement, we focus on the MFI clover and FLIC actions. We now compare in detail the performance for three actions: the Wilson, MFI clover and FLIC. To see how improving the condition number translates into a saving in CG iterations, we calculated the number of Multi-CG iterations required to evaluate D_o once across the ensemble for each of these actions, using some typical simulation parameters. The number of modes we project out is determined by the spectral density at the highest mode. The spectral flow results show that in each of the actions tested, past a certain point the spectral density becomes very high and there is no computational benefit in projecting modes past this point. Therefore, we choose to stop projecting out modes when only a marginal improvement in the condition number is gained by projecting out more eigenmodes.

Action	β	Projections	Mean	Min	Max
Wilson	4.38	15	219	188	253
	4.60	15	202	190	212
MFI clover	4.38	15	200	178	240
FLIC12	4.38	10	92	89	100
FLIC6	4.38	10	90	86	101
FLIC4	4.60	15	109	106	112

Table 5.1: Conjugate Gradient (CG) Iterations needed for a single evaluation of $\epsilon_N(x)$ using actual simulation parameters.

The Wilson and MFI clover are tested using the 14th order optimal rational polynomial (ORP) approximation[54]. FLIC results use the 12th order polar decomposition, chosen to give a maximum deviation from $\epsilon(x)$ of less than 10^{-6}. Low-lying modes are projected out where necessary. The sign function solution is calculated to a precision of 10^{-6} across the fine ensemble and the coarse ensemble used above. The value of m is chosen differently for each of the actions to optimise κ. Given the relative lack of improvement in using the MFI clover action compared to the Wilson, we discard it at this point and concentrate on comparing the Wilson and FLIC actions. As the results in Table 5.1 show, the FLIC action is by far the best in terms of convergence with a reduction in iterations compared to the Wilson action of a factor of between 1.9 and 2.4. What is not clear from this is how the saving in iterations translates into the most important quantity, a saving in compute time.

Shifting from a standard Wilson action to a partially smeared action means that we now have two sets of gauge fields, the standard and smeared links. This doubles the number of vector-multiplications needed, and the standard spin-projection trick[66] is no longer applicable, potentially causing an additional

factor of two in both the multiplications needed and the communications needed. We show that this is not the case.

5.5 Spin Projection

To begin, we review the spin-projection trick for the Wilson action, which utilises projection operators in spinor space to reduce the computation required to evaluate the Wilson action. We then generalise this trick to the broader class of split-link actions. Finally, we examine the FLIC action specifically, and discuss a similar trick for reducing the cost of evaluating the clover term.

5.5.1 Standard Spin-Projection Trick

The Wilson-Dirac operator can be written as

$$\begin{aligned}(D_{\mathrm{w}}\psi)_x &= (4+m)\psi_x - \frac{1}{2}\sum_\mu (1-\gamma_\mu)U_\mu(x)\psi_{x+\mu} \\ &\qquad + (1+\gamma_\mu)U_\mu^\dagger(x)\big)\psi_{x-\mu} \\ &= (4+m)\psi_x - \sum_\mu U_\mu(x)\Gamma_\mu^-\psi_{x+\mu} \\ &\qquad + U_\mu^\dagger(x)\Gamma_\mu^+\psi_{x-\mu}, \end{aligned} \tag{5.26}$$

where we have defined the spin projectors

$$\Gamma_\mu^\pm = \frac{1}{2}(1\pm\gamma_\mu). \tag{5.27}$$

If we now examine, for example, $\Gamma_2^\pm$ we see that

$$\Gamma_2^\pm \begin{pmatrix}\psi^1\\ \psi^2\\ \psi^3\\ \psi^4\end{pmatrix} = \begin{pmatrix}\psi^1 \mp \psi^4\\ \psi^2 \pm \psi^3\\ \pm\psi^2 + \psi^3\\ \mp\psi^1 + \psi^4\end{pmatrix}. \tag{5.28}$$

Similar expressions for $\mu = 1,3,4$ allow us to deduce that we only need to evaluate the action of the links on the upper half (in spinor space) of $\Gamma_\mu^\pm\psi_{x\mp\mu}$, as the lower components are equal to the upper components multiplied by ± 1 or $\pm i$. In doing so we can halve the number of floating point multiplications needed in the evaluation of D_{w}, and also reduce intermediate memory usage. This trick can be applied in any of the standard representations for the γ matrices.

5.5.2 Generalised Spin-Projection Trick

We now consider the case where there are two sets of links, $U_\mu(x)$ for the naive Dirac operator $\nabla\!\!\!/$ and $U'_\mu(x)$ for the irrelevant Wilson term (denoted by Δ' to indicate that it contains only the links U'). In the case of the FLIC action the irrelevant links are APE-smeared, but what follows is perfectly general and does not depend upon any particular relationship between U and U'. Now our "split-link" operator is

$$
\begin{aligned}
(D_{\text{split}}\psi)_x &= \big((\nabla\!\!\!/ + \frac{1}{2}\Delta' + m)\psi\big)_x \\
&= (4+m)\psi_x - \frac{1}{2}\sum_\mu \big(U'_\mu(x) - \gamma_\mu U_\mu(x)\big)\psi_{x+\mu} + \big(U'^{\dagger}_\mu(x) + \gamma_\mu U^{\dagger}_\mu(x)\big)\psi_{x-\mu}.
\end{aligned}
\tag{5.29}
$$

We can observe that our projectors do not present themselves immediately as they did before. At this point, compared to the standard Wilson action, we must perform four times as many floating point multiplications, two for the split links, and two for the loss of the spin projectors. However, we have

$$
\mathbf{1} = \Gamma^+_\mu + \Gamma^-_\mu \text{ and } \gamma_\mu = \Gamma^+_\mu - \Gamma^-_\mu, \tag{5.30}
$$

which implies

$$
\begin{aligned}
(D_{\text{split}}\psi)_x = (4+m)\psi_x - \frac{1}{2}\sum_\mu &\big(U'_\mu(x) - U_\mu(x)\big)\Gamma^+_\mu\psi_{x+\mu} + \big(U'_\mu(x) + U_\mu(x)\big)\Gamma^-_\mu\psi_{x+\mu} + \\
&\big(U'^{\dagger}_\mu(x) + U^{\dagger}_\mu(x)\big)\Gamma^+_\mu\psi_{x-\mu} + \big(U'^{\dagger}_\mu(x) - U^{\dagger}_\mu(x)\big)\Gamma^-_\mu\psi_{x-\mu}.
\end{aligned}
\tag{5.31}
$$

It is now clear that by defining symmetrised and anti-symmetrised links,

$$
U^+_\mu(x) = \frac{1}{2}\big(U'_\mu(x) + U_\mu(x)\big) \text{ and } U^-_\mu(x) = \frac{1}{2}\big(U'_\mu(x) - U_\mu(x)\big) \tag{5.32}
$$

we can write

$$
\begin{aligned}
(D_{\text{split}}\psi)_x = (4+m)\psi_x - \sum_\mu &U^-_\mu(x)\Gamma^+_\mu\psi_{x+\mu} + U^+_\mu(x)\Gamma^-_\mu\psi_{x+\mu} \\
&+ U^{+\dagger}_\mu(x)\Gamma^+_\mu\psi_{x-\mu} + U^{-\dagger}_\mu(x)\Gamma^-_\mu\psi_{x-\mu}.
\end{aligned}
\tag{5.33}
$$

Immediately we see that the Wilson spin projection trick is simply a special case of the split link trick where $U = U'$. The same saving in multiplications that we received in the Wilson case applies here, so we have in principle a factor of two compared to the Wilson action because U^- is not zero. In actuality, efficient cache usage will reduce this to less than a factor of two.

5.5.3 Clover Trick

The FLIC action is a split-link action with clover term, so for completeness we review a (well-known) similar trick for the clover term that exploits the structure of $\sigma_{\mu\nu}$. In the evaluation of the clover term, we note that in the chiral representation of γ matrices,

$$\gamma_4 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \gamma_5 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}, \tag{5.34}$$

the matrix $\sigma_{\mu\nu}$ satisfies the following (in 2×2 block notation),

$$\sigma_{12} \begin{pmatrix} \psi \\ \chi \end{pmatrix} = \sigma_{34} \begin{pmatrix} -\psi \\ \chi \end{pmatrix}, \quad \sigma_{13} \begin{pmatrix} \psi \\ \chi \end{pmatrix} = \sigma_{24} \begin{pmatrix} \psi \\ -\chi \end{pmatrix}, \quad \sigma_{14} \begin{pmatrix} \psi \\ \chi \end{pmatrix} = \sigma_{23} \begin{pmatrix} -\psi \\ \chi \end{pmatrix}. \tag{5.35}$$

So we have, for example,

$$F_{12}\sigma_{12} \begin{pmatrix} \psi \\ \chi \end{pmatrix} + F_{34}\sigma_{34} \begin{pmatrix} \psi \\ \chi \end{pmatrix} = \begin{pmatrix} (F_{12} - F_{34})\sigma_{12}\psi \\ (F_{12} + F_{34})\sigma_{12}\chi \end{pmatrix}. \tag{5.36}$$

This means that if we store the combinations $F_{12} \pm F_{34}, F_{13} \pm F_{24}, F_{14} \pm F_{23}$ we can halve the number of floating point multiplications needed in the evaluation of the clover term, further improving the computational efficiency of the FLIC action. Moreover, the formulation of the split link action in (5.33) allows groups who have efficient code for the Wilson action to simply implement efficient code for the FLIC action.

5.6 Computational Analysis

The results of the previous section reduce the cost of evaluating the FLIC action to approximately twice that of the standard Wilson action. We have verified using our implementation that the cost of FLIC is (slightly less than) twice that of the Wilson, including the cost of the clover term. On the other hand, evaluating the action of $\epsilon^{(n)}(x)$ on a vector costs $O(2N)$ vector multiplications in addition to the two evaluations of the kernel, H. It quickly becomes clear that the only real way to see how much of an improvement we have made is to do an actual calculation and compare the compute time needed.

To test the actual speedup, we choose to calculate the low-lying eigenmodes of $H_o^2 = D_o^\dagger D_o$ for the two different kernels, Wilson and FLIC. This calculation allows us to verify that both kernels give the appropriate spectral properties[46], and also allows us to calculate directly the relative compute time needed to evaluate D_o in each case. For the Wilson action we used the 14th order Rational Polynomial Approximation, with mass parameter $m = 1.65$ and projected out

15 eigenvalues. For the FLIC action, use the polar decomposition at 12th order, without any loss in accuracy for the sign function approximation. This saves us a (small) amount of computation. To optimise the condition number we choose to perform only 6 APE sweeps with the mass parameter set to $m = 1.45$ and projecting out 10 eigenvalues. To minimise the computation needed, we implement individual pole convergence testing in our Multi-CG routine. The first pole is considered converged in the nth iteration according to the usual criterion based on the residue, $||r_n|| < \delta$, where we chose $\delta = 10^{-8}$. The convergence criterion for the other poles is easily deduced by noting the shifted polynomial structure of the residual, $r_n^i = P_n(H^2 + \sigma(i))r_0 = \zeta_n^\sigma P_n(H^2)r_0 = \zeta_n^\sigma r_n$. Then the ith pole is considered converged if

$$||r_n||\zeta_n^{\sigma(i)} < 0.1 \times \delta, \tag{5.37}$$

where $\zeta_n^{\sigma(i)}$ is defined as in Eq. (2.44) of Ref. [53]. We have tested this convergence criterion by calculating individual residues and found it to be numerically very safe, and also to save significant amounts of computation. We consider the ten $8^3 \times 16, \beta = 4.38$ lattices. Computations are performed on 4 nodes of the Orion supercomputer, a Sun E420R cluster comprised of 40 nodes, with each node possessing 4 GB of RAM, 16 MB of L2-cache, and 4 UltraSPARC II 450 MHz processors and with nodes connected by Myrinet networking. The lowest 6 eigenmodes of H_o^2 are calculated on each configuration using the Ritz functional method [67]. We measure the compute time spent in each of the different parts of the "inner-CG" calculation, with the following results.

Code portion	$t_{\rm Wilson}/t_{\rm FLIC}$
1 Kernel-vector evaluation (H)	0.59
1 Multi-shift CG iteration (including H)	0.864
1 Multi-shift CG iteration (excluding H)	1.127
1 overlap-vector evaluation	1.867

Table 5.2: Ratio of actual compute time spent in the various parts of the algorithm.

The results show that using the FLIC action as the kernel in the overlap formalism provides a saving of a factor of 1.9 in actual compute-time spent in evaluating the overlap action. This is easily understood by first observing that the time spent in the fermion matrix multiplication constitutes less than half of the compute time spent in the inner CG inversion. Additionally, as the improved condition number of the FLIC kernel allows us to use the 12th order polar decomposition, we expend less effort per iteration in the CG component of the sign function evaluation. This is because the number of unconverged poles per iteration is reduced, as demonstrated in Table 5.3.

Pole	Wilson	FLIC6	Pole	Wilson	FLIC6
1	188^{+32}_{-21}	85^{+11}_{-6}	8	55^{+4}_{-3}	19^{+1}_{-1}
2	188^{+32}_{-21}	82^{+10}_{-4}	9	39^{+2}_{-2}	14^{+1}_{-1}
3	188^{+32}_{-21}	65^{+6}_{-4}	10	28^{+1}_{-2}	10^{+1}_{-0}
4	188^{+31}_{-21}	50^{+4}_{-2}	11	19^{+2}_{-1}	7^{+0}_{-1}
5	161^{+15}_{-13}	39^{+3}_{-2}	12	14^{+0}_{-1}	4^{+0}_{-0}
6	116^{+7}_{-8}	31^{+2}_{-2}	13	9^{+0}_{-1}	-
7	80^{+5}_{-5}	24^{+2}_{-1}	14	5^{+1}_{-0}	-

Table 5.3: Breakdown of the mean convergence for each of the poles. Shown are the mean number of iterations, and the maximum and minimum number of iterations indicated by their difference with the mean.

These facts mean that the overall compute time per inner CG iteration increases by only 15% when moving to the FLIC kernel, and hence the saving of 55% in the total number of inner CG iterations needed translates into a saving in compute time. Thus we have shown that the FLIC action is numerically superior to the Wilson action as an overlap kernel. What has not been answered is what, if any, are the differences in physical properties of D_o using the different kernels. For example, overlap fermions are free of $O(a)$ errors irrespective of the choice of kernel, but in general may have different $O(a^2)$ errors.

5.7 Further Improvements

The use of non-perturbatively improved renormalisation group gauge actions such as Iwasaki or DBW2 has been shown to improve the low-lying spectrum of the Wilson operator[68, 69], with DBW2 giving the best improvement. Our analysis so far has been based on tadpole improved glue, but we expect that the improvements that we have gained thus far to be above and beyond any improvements from gauge action. To verify this, we conduct a comparison of the low lying spectrum of the Wilson and FLIC actions on DBW2 glue. Calculations are performed on three lattices, two $8^3 \times 16$ lattices at $\beta = 9.836(a = 0.194 \text{ fm}), 10.413(a = 0.165 \text{ fm})$ and one $12^3 \times 24$ lattice, at $\beta = 11.783(a = 0.125 \text{ fm})$. Additionally, we test whether the inclusion of a three-loop improved $F_{\mu\nu}$ in the FLIC action (denoted FLIC-3L) gives any improvement over the standard one-loop clover term. The lowest ten eigenvalues are calculated as a function of m for an ensemble of 10 configurations at each β.

Figure 5.7 demonstrates the flow across the fine $\beta = 11.783$ ensemble. We can see that the Wilson action has enjoyed a considerable reduction in the number of isolated low-lying modes compared to the tadpole improved glue case, and also a slight increase in the gap away from zero of the dense spectral region. The

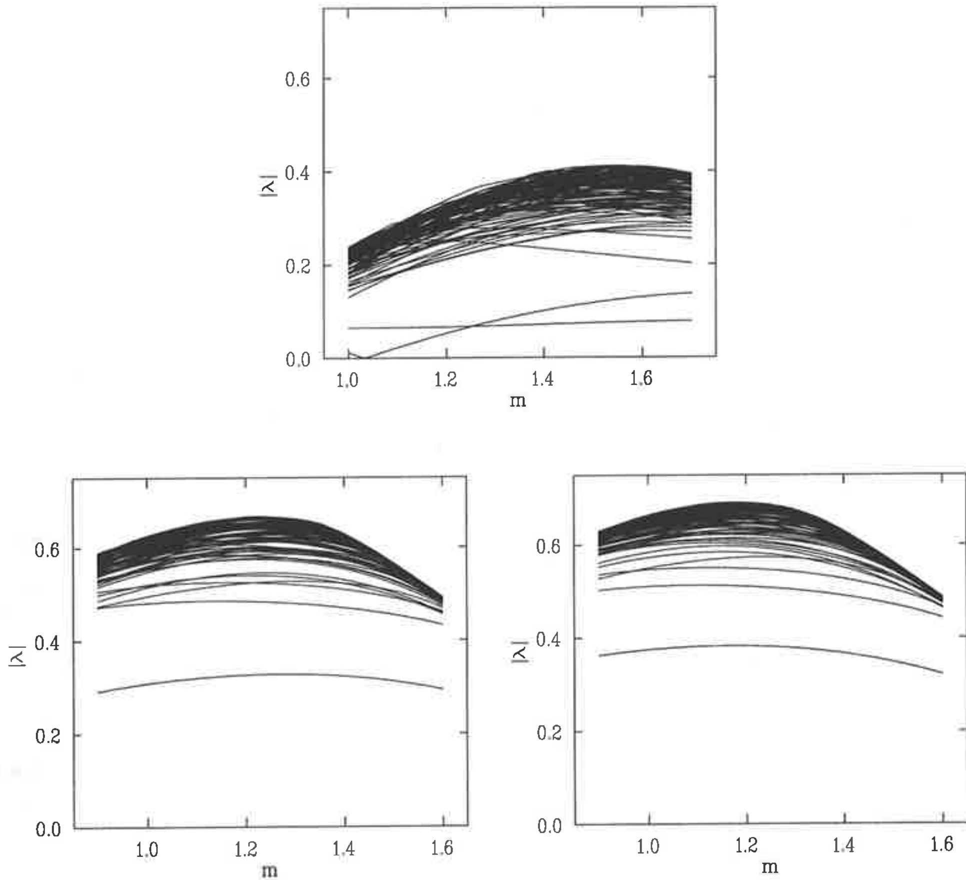

Figure 5.7: Spectral flow of $H_{\rm w}$ (top), $H_{\rm flic}$ (bottom left) and $H_{\rm flic}^{\rm 3L}$ (bottom right) at $\beta = 11.783$ (right).

FLIC action displays similar benefits, but once again the gap from zero is much better than the Wilson action. The FLIC-3L action is equally good, displaying a slightly larger gap than for its one loop counterpart.

The results from the two $8^3 \times 16$ lattices are shown in Fig. 5.8. The improvement in comparison to the Luscher-Weisz glue is seen once again, both in the isolated low modes and the gap of the dense region, for all previously tested actions. Again, the FLIC spectrum is significantly better than that of the Wilson action. The same slight increase in the gap from zero of the FLIC-3L spectrum compared to FLIC is also observed. So we have confirmed that as expected the improved spectral properties of the FLIC action compared to the Wilson are in addition to any improvements from the gluonic action.

Finally, in Fig. 5.9 we study the effects of different numbers of smearing sweeps on the low lying spectrum of FLIC and FLIC-3L. The qualitative behaviour across all three lattices is similar, with the initial sweeps bringing signif-

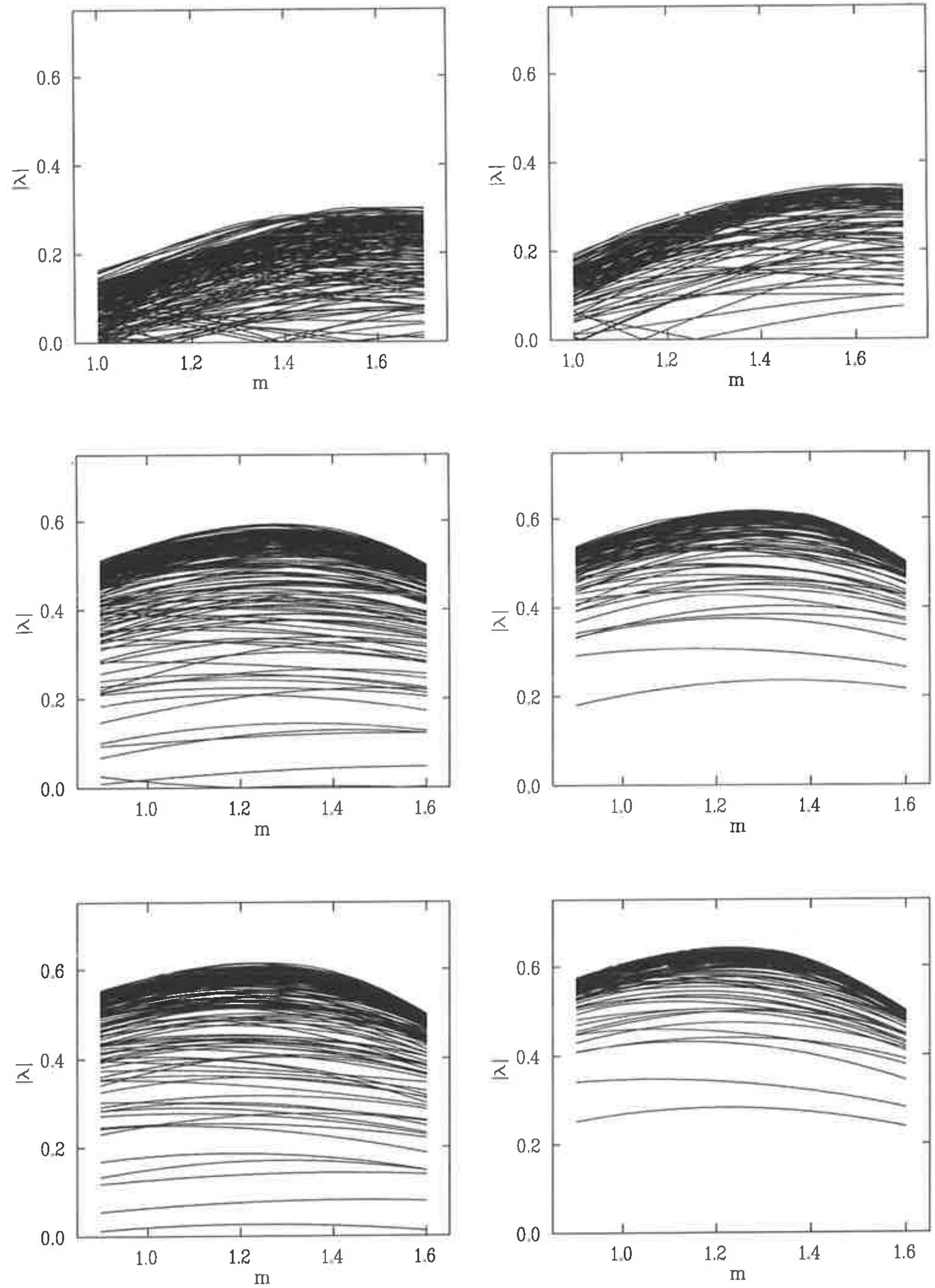

Figure 5.8: Spectral flow of $H_{\rm w}$ (top row), $H_{\rm flic}$ (middle row) and $H^{\rm 3L}_{\rm flic}$ (bottom row) at $\beta = 9.836$ (left column) and $\beta = 10.413$ (right column).

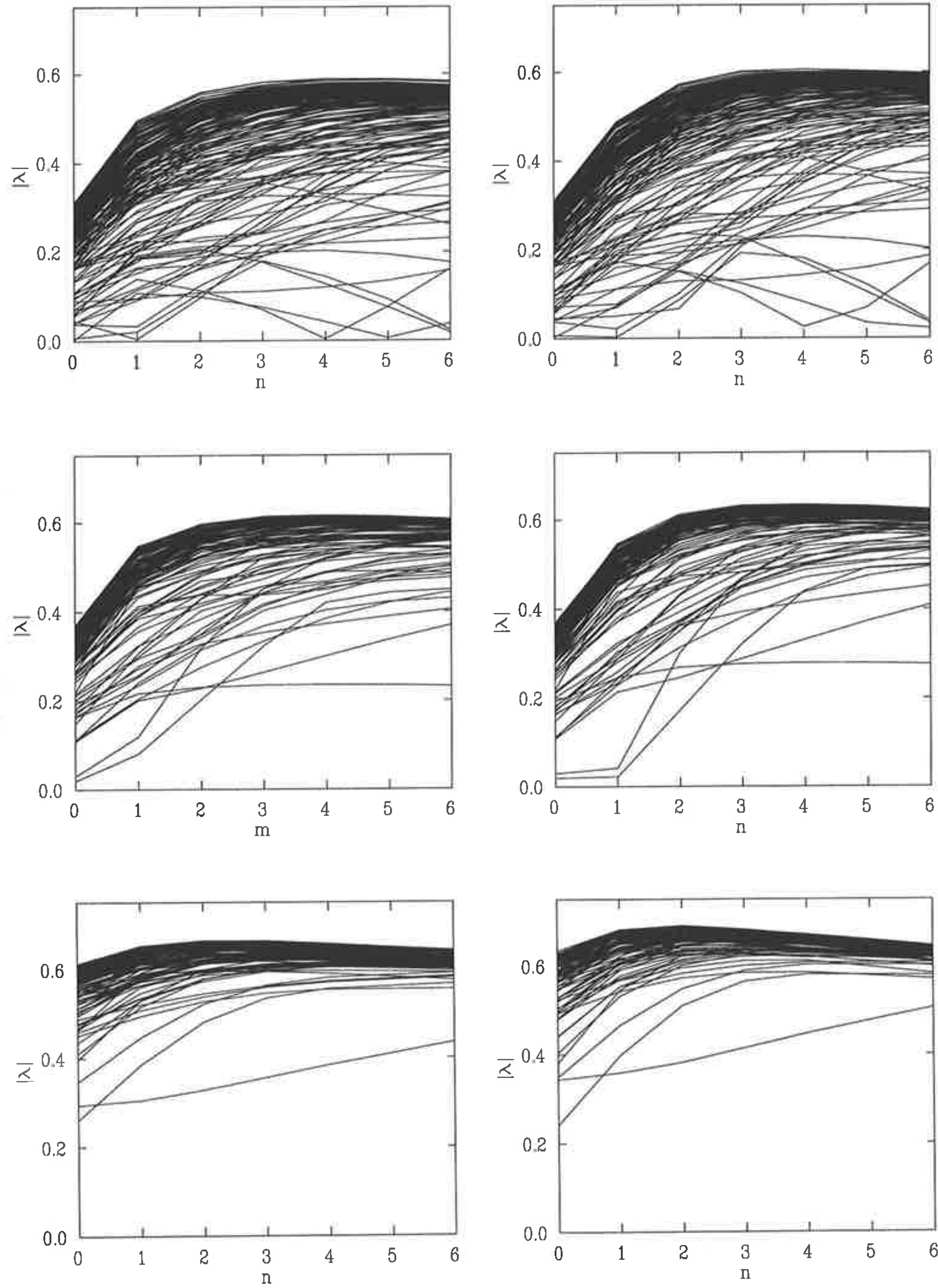

Figure 5.9: Dependence of the spectrum of H_{flic} (left column) H_{flic}^{3L} (right column) on the number of smearing sweeps at $\beta = 9.836$ (top), $\beta = 10.413$ (middle), and $\beta = 11.783$ (bottom).

icant improvement, and then at some point additional smearing has little effect. Both FLIC and FLIC-3L display nearly identical behaviour.

To reinforce the results of the spectral flow comparison, the condition numbers of the three actions are displayed in Fig. 5.10. Error bars again indicate the maximum and minimum κ across the ensemble, with the lines giving the mean condition number, as a function of m, after projecting out the lowest 5 eigenmodes. The FLIC and FLIC-3L lines are well below the Wilson lines, giving a clear indication of the improvement in condition number. Furthermore, the two FLIC actions display behaviour that is nearly independent of β, while we see that the Wilson action performs increasingly poorly on the coarser lattices. This is a significant advantage for FLIC, as it allows one to go to larger lattice spacings (and hence larger physical volumes) at an (approximately) fixed cost for D_o.

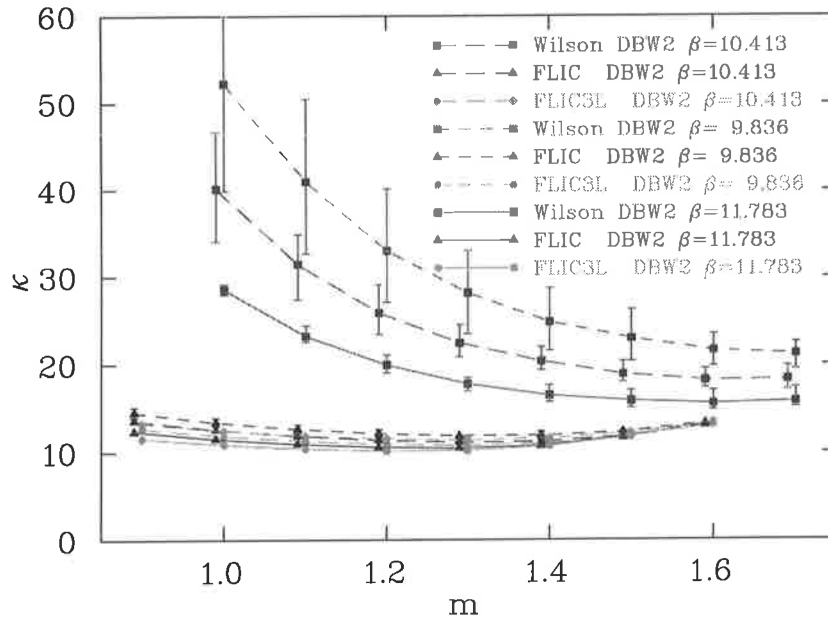

Figure 5.10: Condition number of the Wilson, FLIC and FLIC-3L actions as a function of m, for the three different DBW2 lattices at $\beta = 9.836, 10.413, 11.783$. Note that the FLIC results are approximately independent of β.

5.8 Analytic Bounds

The locality of the overlap-Dirac operator with a FLIC kernel, or simply the FLIC Overlap, would follow from the proof in the case of the standard overlap,

provided the spectrum of H_{flic} has a gap away from zero. We proceed to derive both upper and lower bounds.

Lemma 5.8.1. *Upper bound. H_{flic} is bounded from above, by*

$$||H_{\text{flic}}|| \leq 3 + 8\sqrt{2} + |4 - m|.$$

Proof. Let T'_μ be the parallel transport operator using smeared links. Define

$$T_\mu^\pm = \frac{1}{2}(T'_\mu \pm T_\mu). \tag{5.38}$$

Then by Eq. (5.33) we have

$$H_{\text{flic}} = (4-m) - \sum_\mu \left((\Gamma_\mu^+ T_\mu^- + \Gamma_\mu^- T_\mu^{-\dagger}) + (\Gamma_\mu^- T_\mu^+ + \Gamma_\mu^+ T_\mu^{-\dagger}) \right) - \sum_{\nu>\mu} \frac{1}{2} \sigma_{\mu\nu} F_{\mu\nu}. \tag{5.39}$$

Now, consider

$$(\Gamma_\mu^+ T_\mu^{-\dagger} + \Gamma_\mu^- T_\mu^-)(\Gamma_\mu^+ T_\mu^- + \Gamma_\mu^- T_\mu^{-\dagger}) = \Gamma_\mu^+ T_\mu^{-\dagger} T_\mu^- + \Gamma_\mu^- T_\mu^- T_\mu^{-\dagger}. \tag{5.40}$$

It is clear that

$$||\Gamma_\mu^\pm T_\mu^\pm T_\mu^{\pm\dagger}|| \leq 1, \tag{5.41}$$

(with any combination of different $\pm$) and hence

$$||\Gamma_\mu^\mp T_\mu^{\mp\dagger} + \Gamma_\mu^\pm T_\mu^\mp|| \leq \sqrt{2}. \tag{5.42}$$

Furthermore, as $F_{\mu\nu}$ is constructed from plaquettes, $||U_{\mu\nu}(x)|| \leq 1$, then it is also straightforward that

$$||F_{\mu\nu}|| \leq 1, \tag{5.43}$$

and we also know $||\sigma_{\mu\nu}|| \leq 1$. Then by triangle inequalites, we have that

$$||H_{\text{flic}}|| \leq |4-m| + \sum_\mu 2\sqrt{2} + \sum_{\nu>\mu} \frac{1}{2} = 3 + 8\sqrt{2} + |4-m|. \tag{5.44}$$

We do not find that this bound is saturated in practice. Typically, we find

$$\lambda_{\max}(H_{\text{flic}}) \approx 4.2 + |4-m|, \tag{5.45}$$

that is the upper bound derived for H_{w} in [3] is only exceeded marginally. □

Lower bounds on H_{w} were derived in the presence of a plaquette smoothness condition on the gauge field. As the FLIC operator contains two sets of links, we will need to make some additional assumptions. It seems reasonable to assume that the smeared links satisfy their own plaquette smoothness condition. Furthermore, we also need to consider "mixed" plaquettes, where some of the

links are thin (unsmeared) and some are smeared. It does not seem unreasonable to assume a third smoothness condition for these mixed plaquettes. We note that the mixed plaquettes include degenerate or "collapsed" plaquettes, where the two forward and two backward links that make up the plaquette trace the same path but in opposite directions (the closed curve forms a line rather than a loop). If constructed using a single set of links, the degenerate plaquette would normally be unity, but because in the mixed case the forward and backward paths can use different links, this no longer holds.

Lemma 5.8.2. *Lower bound. Let $U_\mu(x)$ denote the standard gauge field, and $U'_\mu(x)$ the smeared gauge field. Let $U_{\mu\nu}(x)$ denote the standard plaquette field, and $U''_{\mu\nu}(x)$ denote the smeared plaquette field. Let $U'_{\mu\nu}(x)$ denote the mixed plaquette field, which includes degenerate loops. Then given that*

$$||1 - U_{\mu\nu}|| \leq \epsilon, \qquad ||1 - U'_{\mu\nu}|| \leq \epsilon', \qquad ||1 - U''_{\mu\nu}|| \leq \epsilon'',$$

for $\epsilon, \epsilon', \epsilon'' > 0$, $H_{\text{flic}}(-m)$ is bounded from below by

$$\lambda_{\min}(H_{\text{flic}}(-m)) \geq \sqrt{1 - 48\epsilon' - 45\epsilon''} - |1 - m|.$$

Proof. First recall that

$$\not{\nabla}^2 = \nabla \cdot \nabla + \frac{1}{4}[\gamma, \gamma] \cdot [\nabla, \nabla], \tag{5.46}$$

$$\Delta^2 = 4\Delta + 4\nabla \cdot \nabla + \sum_{\mu \neq \nu} \Delta_\mu \Delta_\nu. \tag{5.47}$$

Let ∇'_μ, Δ'_μ denote the usual operators, but constructed using smeared links $U'_\mu(x)$. Define $F'_{\mu\nu}$ such that

$$\frac{1}{4}[\gamma_\mu, \gamma_\nu] F'_{\mu\nu} = \frac{1}{2}\sigma_{\mu\nu} F^{\text{cl}}_{\mu\nu}, \tag{5.48}$$

then

$$H^2_{\text{flic}}(-m) = (-\not{\nabla} + \frac{1}{2}\Delta' - \frac{1}{8}[\gamma, \gamma] \cdot F' - m)(\not{\nabla} + \frac{1}{2}\Delta' - \frac{1}{8}[\gamma, \gamma] \cdot F' - m), \tag{5.49}$$

and thus

$$\begin{aligned} H^2_{\text{flic}}(-1) = 1 - \not{\nabla}^2 + \frac{1}{4}\Delta'^2 + (\frac{1}{8}[\gamma, \gamma] \cdot F')^2 - \Delta' + \frac{1}{4}[\gamma, \gamma] \cdot F' \\ - \frac{1}{2}[\not{\nabla}, \Delta'] + \frac{1}{8}[\not{\nabla}, [\gamma, \gamma] \cdot F'] - \frac{1}{16}\{\Delta', [\gamma, \gamma] \cdot F'.\} \end{aligned} \tag{5.50}$$

Then using the identities (5.46) and (5.47) it is straightforward that

$$\begin{aligned} H^2_{\text{flic}}(-1) = 1 - \not{\nabla}^2 + \not{\nabla}'^2 - \frac{1}{4}[\gamma, \gamma] \cdot [\nabla', \nabla'] + \sum_{\mu \neq \nu} \Delta'_\mu \Delta'_\nu + (\frac{1}{8}[\gamma, \gamma] \cdot F')^2 \\ + \frac{1}{4}[\gamma, \gamma] \cdot F' - \frac{1}{2}[\not{\nabla}, \Delta'] + \frac{1}{8}[\not{\nabla}, [\gamma, \gamma] \cdot F'] - \frac{1}{16}\{\Delta', [\gamma, \gamma] \cdot F'\} \end{aligned} \tag{5.51}$$

From our assumptions, we have that

$$||[T_\mu, T_\nu]|| \le \epsilon, \qquad ||[T_\mu, T'_\nu]|| \le \epsilon', \qquad ||[T'_\mu, T'_\nu]|| \le \epsilon'', \tag{5.52}$$

and similarly so if we replace any T_μ or T'_μ with its conjugate. Then recalling our steps in the derivation at the end of the previous chapter, we have

$$||\frac{1}{4}[\gamma, \gamma] \cdot [\nabla', \nabla']|| \le \sum_{\nu>\mu} \epsilon'', \tag{5.53}$$

and $\Delta'_\mu \Delta'_\nu = W_{\mu\nu} + X_{\mu\nu}$, where $W_{\mu\nu} \ge 0$ and

$$||X|| = \frac{1}{8}||\sum_{\mu\neq\nu} T'_\mu[T'^\dagger_\mu, T'_\nu + T'^\dagger_\nu] + T'^\dagger_\mu[T'_\mu, T'_\nu + T'^\dagger_\nu]|| \le \sum_{\nu>\mu} \epsilon''. \tag{5.54}$$

The familiar looking term $Y = -\frac{1}{2}[\nabla\!\!\!/, \Delta']$ is now a mixed commutator, and so some symmetries that we used previously are spoiled. Now we have

$$||Y|| = \frac{1}{4}||\sum_{\mu,\nu} \gamma_\mu [T_\mu - T^\dagger_\mu, T'_\nu + T'^\dagger_\nu]|| \le \sum_{\mu,\nu} \epsilon'. \tag{5.55}$$

Let

$$Z = \frac{1}{4}[\gamma, \gamma] \cdot F' - \frac{1}{16}\{\Delta', [\gamma, \gamma] \cdot F'\}. \tag{5.56}$$

A straightforward consequence of our assumptions is that $||F'_{\mu\nu}|| \le \epsilon''$, and so

$$\begin{aligned} ||Z|| &= \frac{1}{4}||\sum_{\mu\neq\nu} \Big([\gamma_\mu, \gamma_\nu]F'_{\mu\nu} - \frac{1}{4}\sum_\lambda \{2 - T'_\lambda - T'^\dagger_\lambda, [\gamma_\mu, \gamma_\nu]F'_{\mu\nu}\}\Big)|| \\ &= ||\sum_{\mu\neq\nu} \Big(\frac{3}{4}[\gamma_\mu, \gamma_\nu]F'_{\mu\nu} - \frac{1}{8}\sum_\lambda \gamma_\mu\gamma_\nu \{T'_\lambda + T'^\dagger_\lambda, F'_{\mu\nu}\}\Big)|| \le \frac{11}{2}\sum_{\nu>\mu} \epsilon'' \end{aligned} \tag{5.57}$$

Now consider $C = \frac{1}{8}[\nabla\!\!\!/, [\gamma, \gamma] \cdot F']$. Then $C = A + B$ where

$$||A|| = \frac{1}{2}||\sum_{\mu\neq\nu} \gamma_\nu [\nabla_\mu, F'_{\mu\nu}]|| \le 2\sum_{\nu>\mu} \epsilon', \tag{5.58}$$

$$||B|| = \frac{1}{4}||\sum_{\mu\neq\nu\neq\lambda} \gamma_\lambda\gamma_\mu\gamma_\nu [\nabla_\lambda, F'_{\mu\nu}]|| \le \frac{1}{2}\sum_{\mu\neq\nu\neq\lambda} \epsilon'. \tag{5.59}$$

Finally we examine $D = \nabla\!\!\!/'^2 - \nabla\!\!\!/^2$. So,

$$D = \frac{1}{4}\sum_{\mu,\nu} \gamma_\mu\gamma_\nu \big((T'_\mu - T'^\dagger_\mu)(T'_\nu - T'^\dagger_\nu) - (T_\mu - T^\dagger_\mu)(T_\nu - T^\dagger_\nu)\big). \tag{5.60}$$

Now we consider

$$(T'_\mu T'_\nu - T_\mu T_\nu)\psi(x) = (U'_\mu(x)U'_\nu(x+e_\mu) - U_\mu(x)U_\nu(x+e_\mu))\psi(x+e_\mu+e_\nu)$$
$$= (1 - U^{\mathrm{d}}_{\mu\nu}(x))U'_\mu(x)U'_\nu(x+e_\mu)\psi(x+e_\mu+e_\nu). \quad (5.61)$$

where $U^{\mathrm{d}}_{\mu\nu} = U_\mu(x)U_\nu(x+e_\mu)U'^{\dagger}_\nu(x+e_\mu)U'^{\dagger}_\mu(x)$ is the degenerate plaquette at x along the $x, x+e_\mu, x+e_\mu+e_\nu$ line. But by our assumptions we then have

$$||T'_\mu T'_\nu - T_\mu T_\nu|| \leq \epsilon' \;\forall \mu, \nu, \quad (5.62)$$

and it is clear that the same bound holds true if we replace T_μ, T'_μ and/or T_ν, T'_ν with their conjugates. Then

$$D = \frac{1}{4}\sum_\mu (T'^2_\mu + T'^{\dagger 2}_\mu) - (T^2_\mu + T^{\dagger 2}_\mu) - \frac{1}{4}\sum_{\mu\neq\nu} \gamma_\mu\gamma_\nu (T'_\mu T'^{\dagger}_\nu + T'^{\dagger}_\mu T'_\nu) - (T_\mu T_\nu{}^\dagger + T_\mu{}^\dagger T_\nu), \quad (5.63)$$

and hence

$$||D|| \leq \frac{1}{2}\sum_\mu \epsilon' + \sum_{\nu>\mu} \epsilon'. \quad (5.64)$$

Therefore, ignoring non-negative operators and combining our bounds we obtain

$$H^2_{\mathrm{flic}}(-1) \geq 1 - 48\epsilon' - 45\epsilon'', \quad (5.65)$$

and the flow inequality can then be used to obtain the required result.

□

Remarkably, the bound does not depend upon ϵ, and hence the smoothness of the background field only enters in weakly via ϵ'. This gives us an immediate explanation of why the condition number of H_{flic} is nearly independent of β. The bound is only meaningful in the area defined by $48\epsilon' + 45\epsilon'' < 1$. This means that we need (at least) that $\epsilon' < \frac{1}{48}$ and $\epsilon'' < \frac{1}{45}$, which is more stringent than the requirements placed upon ϵ in the bounds on H_{w}. However, as ϵ', ϵ'' depend upon the smeared links, we expect that they will satisfy these requirements more readily than ϵ.

While we have seen the numerical superiority of the FLIC Overlap, we have not yet addressed the question of physical properties. We are interested in doing this in both quenched and unquenched QCD, which leads us to the issue of conducting dynamical simulations with actions that contain smeared links, the subject of our next investigation.

Chapter 6

Dynamical FLIC Fermions

Recent progress in the field has shown that the behaviour of quenched QCD can differ from the true theory both qualitatively and quantitatively in the chiral regime. As it is in the chiral regime where the difference from the valence approximation will be highlighted, we would like to go to light quark masses in dynamical QCD. This is an extremely computationally expensive endeavour. Ideally, this would be done using overlap fermions, although large scale dynamical overlap simulations would push the limits of current computing power, to say the least. Alternatively, one could go to the so-called partially quenched approximation to explore some of the effects of fermionic vacuum fluctuations.

6.1 Partial Quenching

The term partially quenched has come to have two meanings. The first is an approximation to QCD in which the dynamical sea quark mass differs from the valence quark mass. The second meaning refers to the possibility of simulating QCD using different lattice fermion actions in the sea and valence sectors. As in the continuum limit both discretisations must give the continuum fermion action, on the surface the approximation does not seem too unreasonable (although the approximation does have some more subtle theoretical implications which we choose to ignore for pragmatic reasons).

The partially quenched approximation allows us to take advantage of the good properties of the overlap in the valence sector, while we can use a less computationally intensive, non-chiral action in the sea quark sector. In example 4.2 we derived the continuum limit of the (standard) overlap. From the derivation, we can see that in a sense the overlap can be viewed as a mechanism to turn a non-chiral action (the kernel) into a chiral one. Therefore, the overlap kernel seems like a natural choice for the sea quark action (we refer to this choice as the dynamical kernel approximation). So, for the FLIC overlap, FLIC fermions would be chosen for the dynamical fermion sector.

6.2 Hybrid Monte Carlo

The standard technique for simulating dynamical fermions has for some time now been Hybrid Monte Carlo (HMC) [70]. It is exact, ergodic and (using the standard leapfrog integration) is $O(V^{\frac{5}{4}})$, that is, it scales almost linearly with the lattice volume V. We briefly review the HMC algorithm for generating dynamical gauge field configurations.

As we saw in §2.4, the fermionic degrees of freedom can be integrated out of the functional integral to yield a functional integral soley over gauge field space, weighted with an effective action

$$S_{\text{eff}}[U] = S_{\text{g}}[U] - \ln \det D_{\text{f}}[U], \tag{6.1}$$

where the fermionic vacuum fluctuations are encoded in the fermionic determinant. The functional integral on the lattice is evaluated through the technique of importance sampling, where an independent ensemble $\{U_i\}$ of representative gauge fields distributed according to the probability distribution

$$\rho(U_i) = e^{-S_{\text{eff}}[U_i]}, \tag{6.2}$$

is sampled and the desired observable is approximated by the ensemble average

$$\langle \mathcal{O} \rangle \approx \frac{1}{n} \sum_{i=1}^{n} \mathcal{O}[U_i]. \tag{6.3}$$

What is needed to do this is an efficient update algorithm to generate configurations with the desired probability distribution. In the Metropolis algorithm, the update $U \to U'$ is proposed entirely randomly, one link at a time, and the change is accepted with probability $\rho(U \to U') = e^{-\Delta S}$, where $\Delta S = S[U'] - S[U]$. In more efficient update algorithms the change is based upon $\frac{\delta S}{\delta U}$, the variation of the action with respect to U.

Ergodicity, the property that starting from a given gauge field our update algorithm has a finite probability of generating any other, is usually incorporated into update algorithms through the introduction of a random number field into the updating process. Exactness, the property that the gauge fields generated have the desired probability distribution, free from systematic errors, is usually implemented through the use of a Metropolis style accept/reject step, where the function $\rho(U \to U') = e^{-\Delta S}$ is evaluated to determine the probability that the update $U \to U'$ is accepted.

There are a variety of techniques for generating statistically independent ensembles of complex fields with a given distribution, but they do not apply to Grassmannian (fermionic) fields. Therefore, in order to deal with the fermionic determinant, we make use of the fact that

$$\det M = \frac{1}{\det M^{-1}} = \int \mathcal{D}\phi^\dagger \mathcal{D}\phi \; e^{-\int d^4x \; \phi^\dagger(x) M^{-1} \phi(x)}, \tag{6.4}$$

and introduce an auxiliary pseudo-fermion field ϕ, so called because it is a complex (bosonic) field rather than Grassmannian. For the integral to be convergent, we require that M is positive definite, so noting that $\det D_\mathrm{f}$ is real and positive we set $M = D_\mathrm{f}^\dagger D_\mathrm{f}, \Rightarrow \det M = \det D_\mathrm{f}^2$, representing two degenerate flavours of sea quarks (it is then possible to use $\det\sqrt{M}$ if a single flavour is desired[71]).

While the use of pseudo-fermions solves one problem, they introduce another. Quenched gauge fields are easy to generate because the gauge action is local, hence $\Delta S_\mathrm{g}[U]$ is local, and therefore one can make use of efficient local updating algorithms, such as the pseudo-heatbath, which effectively updates one link at a time. However the (expensive) evaluation of D_f^{-1} depends on the whole lattice, and so in the pseudo-fermionic case $\Delta S_\mathrm{eff}[U]$ is non-local, precluding the efficient use of local updating algorithms[1].

The Hybrid Molecular Dynamics (HMD) algorithm allows one to update the entire gauge field simultaneously, but suffers from systematic (finite step size) errors. Hybrid Monte Carlo (HMC) is a global updating algorithm which is a combination of HMD and the Metropolis algorithm, removing the systematic errors through the introduction of a global accept/reject step. An outline of the HMC algorithm as applied to pseudo-fermions is as follows (see also for example [5]). A detailed description of the HMC implementation is given in §A.7.

The four-dimensional lattice theory is made five-dimensional through the introduction of a fictitious (simulation) time, or evolution parameter τ. The gauge field U is associated with its (fictitious) conjugate momenta P, and the five-dimensional system is described by the Hamiltonian,

$$\mathcal{H}[U,P] = \sum_{x,\mu} \frac{1}{2} \operatorname{Tr} P_\mu(x)^2 + S_\mathrm{eff}[U]. \tag{6.5}$$

For Gaussian distributed P the expectation value of an observable is unaffected by the 5D kinetic energy,

$$\langle \mathcal{O} \rangle = \frac{1}{\mathcal{Z}} \int \mathcal{D}P\mathcal{D}U \mathcal{O}[U] e^{-\mathcal{H}[U,P]}, \quad \mathcal{Z} = \int \mathcal{D}P\mathcal{D}U e^{-\mathcal{H}[U,P]}. \tag{6.6}$$

Given U, the update $U \to U'$ consists of

(i) *Refreshment:* Sample P from a Gaussian ensemble, $\rho(P) \propto e^{-\frac{1}{2}\operatorname{Tr} P^2}$. Generate a pseudo-fermionic background field ϕ according to $\rho(\phi) \propto e^{-\phi^\dagger M^{-1} \phi}$.

(ii) *Molecular Dynamics Trajectory:* Integrate Hamilton's equations of motion to deterministically evolve (U, P) along a phase space trajectory to (U', P').

(iii) *Metropolis step:* Accept or reject the new configuration (P', U') with probability $\rho(U \to U') = \min(1, e^{-\Delta\mathcal{H}}), \Delta\mathcal{H} = \mathcal{H}[U'] - \mathcal{H}[U]$.

[1] A local alternative to HMC, the Local Bosonic Algorithm (LBA)[72] does exist, although was developed much later than HMC. While LBA is competitive with HMC, HMC remains the preferred algorithm.

The discretised equations of motion are derived by requiring that the Hamiltonian be conserved along the trajectory, $\frac{\mathrm{d}\mathcal{H}}{\mathrm{d}\tau} = 0$. The following discretised equations of motion then approximately conserve $\mathcal{H}$ for small step sizes, $\Delta\tau$,

$$U_\mu(x, \tau + \Delta\tau) = U_\mu(x, \tau) \exp\left(i\Delta\tau P_\mu(x, \tau)\right), \tag{6.7}$$

$$P_\mu(x, \tau + \Delta\tau) = P_\mu(x, \tau) - U_\mu(x, \tau) \frac{\delta S_{\mathrm{eff}}}{\delta U_\mu(x, \tau)}. \tag{6.8}$$

Previously, it has not been clear how to perform HMC with fermion actions that make use of the APE blocking technique in combination with a projection of the blocked links back into the special unitary group. This reunitarisation is often performed using an iterative maximisation of a gauge invariant measure, and this choice of reuniterisation is the source of the difficulty. The problem is that the iterative technique is not differentiable with respect to the gauge field and thus it is not possible to calculate $\frac{\delta S}{\delta U}$, which is necessary for the equations of motion above. In the next section we consider an alternative technique and show that is does not suffer from this problem, allowing the simulation of dynamical fat link fermions with standard HMC (and its variants).

6.3 SU(3) Projection

Recall from §3.5.1 that the APE smeared links $U_\mu^{(n)}(x)$ present in the FLIC fermion action were constructed from $U_\mu(x)$ by performing n smearing sweeps, where in each sweep we first perform an APE blocking step,

$$V_\mu^{(j)}(x)[U^{(j-1)}] = (1-\alpha)\, \longrightarrow + \frac{\alpha}{6} \sum_{\nu\neq\mu} \sqcap + \sqcup\,, \tag{6.9}$$

followed by a projection back into $SU(3)$, $U_\mu^{(j)}(x) = \mathcal{P}(V_\mu^{(j)}(x))$. Frequently, the projection is performed using an algorithm which updates $U^{(j)}$ iteratively in order to maximise the following gauge invariant measure,

$$U_\mu^{(j)}(x)[U^{(j-1)}] = \max_{U' \in SU(3)} \mathrm{Re}\,\mathrm{Tr}(U'_\mu(x) V_\mu^{(j)\dagger}(x)). \tag{6.10}$$

We refer to this projection technique as MaxReTr projection. As we mentioned earlier, it is not differentiable with respect to $U_\mu(x)$ and hence not suitable for use in HMC.

Now, given any matrix X, then $X^\dagger X$ is hermitian and may be diagonalised. Then it is possible (for $\det X \neq 0$) to define a matrix

$$W = X \frac{1}{\sqrt{X^\dagger X}} \tag{6.11}$$

whose spectrum lies on the complex unit circle and is hence unitary ($w(z) = z/z^* z$ is the complex version of the sign function). Furthermore, W possesses the same gauge transformation properties as X.

Example 6.3.1. *Show this.*

Proof. Let $X_\mu(x)$ transform as

$$X_\mu(x) \to G(x)X_\mu(x)G^\dagger(x+e_\mu), \tag{6.12}$$

then

$$X_\mu^\dagger(x) \to G(x+e_\mu)X_\mu^\dagger(x)G^\dagger(x), \tag{6.13}$$

and hence

$$X_\mu^\dagger(x)X_\mu(x) \to G(x+e_\mu)X_\mu^\dagger(x)X_\mu(x)G^\dagger(x+e_\mu). \tag{6.14}$$

Now, $[X_\mu^\dagger(x)X_\mu(x)]^{-\frac{1}{2}}$ has the same transformation properties as $X_\mu^\dagger(x)X_\mu(x)$ and it is then straight forward to see that

$$W_\mu(x) \to G(x)W_\mu(x)G^\dagger(x+e_\mu), \tag{6.15}$$

as required. □

Given the unitary matrix W, we can construct another matrix,

$$W' = \frac{1}{\sqrt[3]{\det W}}W \tag{6.16}$$

which is special unitary[73][2]. As there are three different complex roots, we have a $\mathbb{Z}_3$ ambiguity which we break by choosing the principal value of the cube root[3]. The principal value is chosen as the projected matrices lie closest to those given by the MaxReTr method, and are hence smoother (in the sense of the mean plaquette being close to unity). We refer to this technique for projecting $X_\mu(x)$ into the special unitary group as unit circle projection.

The two methods produce smeared links that are different but numerically close (according to the usual matrix norm, $||A|| = \sqrt{\lambda_{\max}(A^\dagger A)}$). Using the mean link as a measure of the smoothness of the smeared gauge field, Table 6.1 indicates that the two methods presented here produce equally smooth gauge fields. We examine the mathematical (in-)equivalence of the two methods in §6.6.

While numerically the two methods may be nearly equivalent, unit circle projection possesses a significant advantage over MaxReTr projection. Recall from the previous chapter that the matrix inverse square root function can be approximated by a rational polynomial (whose poles lie on the imaginary axis),

$$W[X] \approx W_k[X] \equiv d_0 X(X^\dagger X + c_{2n})\sum_{l=1}^{k}\frac{b_l}{X^\dagger X + c_l}. \tag{6.17}$$

[2]This reference incorrectly omits the cube root.

[3]For complex z, the principal value of the cube root satisfies $-\frac{\pi}{3} < \arg\sqrt[3]{z} < \frac{\pi}{3}$. For purely real z, we choose $\sqrt[3]{z}$ to be real.

Sweep	Unit Circle	MaxReTr
0	**0.866138301214314**	**0.866138301214314**
1	**0.9603**13394813806	**0.9603**48747275940
2	**0.9807**35000838119	**0.9807**51346847750
3	**0.9883**84926461589	**0.9883**93707639555
4	**0.99210**3013943516	**0.99210**7844842705
5	**0.99418**2852413813	**0.99418**5532052157
6	**0.99545**7365275018	**0.99545**8835653863
7	**0.99629**3668622924	**0.99629**4454083006
8	**0.996878**305318083	**0.996878**710433084

Table 6.1: The mean link u_0 for a single configuration as a function of number of APE smearing sweeps, for the two different projection methods. The boldface indicates significant digits which match. The configuration is a dynamical gauge field with DBW2 glue and FLIC sea fermions, at $\beta = 8.0, \kappa = 0.128$.

This approximation is differentiable in a matrix sense for all X for which the inverse square root can be defined. This means that we can construct $\frac{\delta S}{\delta U}$ for fermion actions which involve unit circle projection, and hence it is a reuniterisation method which is compatible with HMC.

6.4 Equations of Motion

Having now written down the APE smearing prescription (with projection) in a differentiable closed form, we proceed to derive the equations of motion necessary for the use of the HMC algorithm with FLIC fermions.

6.4.1 Mathematical Preliminaries

The equations of motion are derived using multi-variate calculus. To make the derivation simple and provide an understanding of how best to implement the equations efficiently, we develop some appropriate mathematical tools. Using index notation, we define a (minimal) set of tensor operations (including differentiation) such that we can perform the derivation in an index free language.

The derivative of a real-valued function $f[A]$ with respect to the matrix A is a rank 2 type (1,1) tensor (distinguishing contravariant and covariant indices),

$$[\frac{\partial f}{\partial A}]^i{}_j = \frac{\partial}{\partial A^j{}_i} f[A]. \tag{6.18}$$

The derivative of a matrix-valued function $M[A]$ with respect to the matrix A

is a rank 4 type (2,2) tensor,

$$[\frac{\partial M}{\partial A}]^i{}_j{}^k{}_l = \frac{\partial}{\partial A^j{}_k} M[A]^i{}_l. \tag{6.19}$$

The set of type (m,n) tensors $\mathcal{T}^m_n$ forms a vector space. We define the outer product $\otimes : \mathcal{T}^1_1 \times \mathcal{T}^1_1 \to \mathcal{T}^2_2$ as

$$(A \otimes B)^i{}_j{}^k{}_l = A^i{}_j B^k{}_l. \tag{6.20}$$

Noting carefully the index ordering, define the "direct" product $\oplus : \mathcal{T}^1_1 \times \mathcal{T}^1_1 \to \mathcal{T}^2_2$ as

$$(A \oplus B)^i{}_j{}^k{}_l = A^k{}_j B^i{}_l. \tag{6.21}$$

Given a scalar function $f[B]$ and a matrix function $B[A]$ the (scalar-matrix) chain rule states

$$\frac{\partial f}{\partial A} = \frac{\partial f}{\partial B} \star \frac{\partial B}{\partial A}, \tag{6.22}$$

where we define the contraction induced by the chain rule as the (rank 2) star product, $\star : \mathcal{T}^1_1 \times \mathcal{T}^2_2 \to \mathcal{T}^1_1$, with

$$(A \star T)^i{}_l = A^j{}_k T^i{}_j{}^k{}_l. \tag{6.23}$$

Given two matrix functions $M[B]$ and $B[A]$, the (matrix-matrix) chain rule states

$$\frac{\partial M}{\partial A} = \frac{\partial M}{\partial B} \star \frac{\partial B}{\partial A}, \tag{6.24}$$

where we define the contraction induced by this chain rule as the (rank 4) star product, $\star : \mathcal{T}^2_2 \times \mathcal{T}^2_2 \to \mathcal{T}^2_2$, with

$$(S \star T)^i{}_j{}^k{}_l = S^i{}_m{}^n{}_l T^m{}_j{}^k{}_n. \tag{6.25}$$

It is interesting to note that the star product induces an algebra structure on the vector space of type (2,2) tensors, that is, $(\mathcal{T}^2_2, +, \star)$ is an algebra with multiplicative identity $I \otimes I$.

Now, define juxtaposition for $A \in \mathcal{T}^1_1, T \in \mathcal{T}^2_2$ by the contractions

$$(AT)^i{}_j{}^k{}_l = A^i{}_m T^m{}_j{}^k{}_l, \tag{6.26}$$
$$(TA)^i{}_j{}^k{}_l = T^i{}_j{}^k{}_m A^m{}_l. \tag{6.27}$$

We note that juxtaposition commutes through star products. If $A, B, C \in \mathcal{T}^1_1$, and $S, T \in \mathcal{T}^2_2$ then

$$T \star AS = TA \star S, \qquad T \star SA = AT \star S, \qquad A \star BSC = CAB \star S. \tag{6.28}$$

Furthermore, the star product respects the outer and direct products,

$$(A \otimes B) \star (C \otimes D) = AC \otimes DB, \tag{6.29}$$
$$(A \oplus B) \star (C \oplus D) = \mathrm{Tr}(AD)C \oplus B, \tag{6.30}$$

but it prefers the direct product,

$$(A \otimes B) \star (C \oplus D) = ACB \oplus D, \tag{6.31}$$
$$(A \oplus B) \star (C \otimes D) = A \oplus CBD. \tag{6.32}$$

All our derivatives will be derived from the basic matrix differentiation rule. Given matrices M, A, B, C then for $M = ABC$ we have

$$\frac{\partial M}{\partial B} = A \otimes C. \tag{6.33}$$

An immediate consequence of this is that

$$\frac{\partial M}{\partial M} = I \otimes I. \tag{6.34}$$

The (scalar-matrix) product rule is

$$\frac{\partial}{\partial A}(fM) = \frac{\partial f}{\partial A} \oplus M + f\frac{\partial M}{\partial A}. \tag{6.35}$$

The (matrix-matrix) product rule is

$$\frac{\partial}{\partial A}(XY) = X\frac{\partial Y}{\partial A} + \frac{\partial X}{\partial A}Y, \tag{6.36}$$

which is easily shown to imply the identity

$$\frac{\partial X^{-1}}{\partial A} = -X^{-1}\frac{\partial X}{\partial A}X^{-1}. \tag{6.37}$$

Of particular numerical importance is the identity

$$A \star (B \otimes C) = CAB \star I \otimes I = CAB, \tag{6.38}$$

which has two major benefits. It allows us to evaluate two matrix multiplications instead of an outer product (computational saving), hence enabling us to implement the equations of motion without having to store any tensor fields (memory saving). For the implementation details, see §A.7.

6.4.2 Standard Derivatives

The equations of motion for FLIC fermions are derived starting from the equations for the standard clover fermion action. We divide the effective action into its gauge part and pseudo-fermionic part,

$$S_{\text{eff}} = S_{\text{g}} + S_{\text{pf}}. \tag{6.39}$$

We reformulate some standard results in terms of the mathematics of the previous section. We will adopt a more convenient notation for quantities with a lattice site index x, using a subscript $U_{\mu,x}$ rather than $U_\mu(x)$. The matrix products of link variables are often denoted diagrammatically. For a plaquette plus rectangle improved gauge action,

$$S_{\text{g}} = \sum_{x,\mu<\nu} \operatorname{Re}\operatorname{Tr}\left[\beta_{1\times1}(1 - U_{\mu\nu}(x)) + \beta_{2\times1}(1 - R^{2\times1}_{\mu\nu}(x)) + \beta_{1\times2}(1 - R^{1\times2}_{\mu\nu}(x))\right], \tag{6.40}$$

we have

$$\frac{\partial S_{\text{g}}}{\partial U_{\mu,x}} = -\sum_{\nu\neq\mu} \beta_{1\times1}\left(\cdots + \cdots\right) + \beta_{1\times2}\left(\cdots + \cdots\right) + \beta_{2\times1}\left(\cdots + \cdots + \cdots + \cdots\right), \tag{6.41}$$

where the filled circles indicate the point x.

The pseudo-fermionic action is $S_{\text{pf}} = -\sum_x \phi^\dagger_x \eta_x$, where $\eta = (D^\dagger D)^{-1}\phi$, hence by equations (6.36) and (6.37) we have

$$\frac{\partial S_{\text{pf}}}{\partial U_{\mu,x}} = \phi^\dagger (D^\dagger D)^{-1}\left(D^\dagger \frac{\partial D}{\partial U_{\mu,x}} + \frac{\partial D^\dagger}{\partial U_{\mu,x}} D\right)(D^\dagger D)^{-1}\phi. \tag{6.42}$$

Setting $\chi = D\eta$, we obtain

$$\frac{\partial S_{\text{pf}}}{\partial U_{\mu,x}} = \chi^\dagger \frac{\partial D}{\partial U_{\mu,x}}\eta + \eta^\dagger \frac{\partial D^\dagger}{\partial U_{\mu,x}}\chi. \tag{6.43}$$

Now, recall

$$(D_{\text{flic}}\psi)_x = -\frac{1}{2}\sum_\mu \left(\frac{U^{\text{fl}}_{\mu,x}}{u^{\text{fl}}_0} - \gamma_\mu \frac{U_{\mu,x}}{u_0}\right)\psi_{x+\mu} + \left(\frac{U^{\text{fl}\dagger}_{\mu,x}}{u^{\text{fl}}_0} + \gamma_\mu \frac{U^\dagger_{\mu,x}}{u_0}\right)\psi_{x-\mu}$$
$$+ \left(4 + m - \frac{1}{4u_0^{\text{fl}4}}\sigma_{\mu\nu}F^{\text{cl}}_{\mu\nu,x}\right)\psi_x \tag{6.44}$$

contains three terms, the Dirac term (constructed with standard links), the Wilson term and the clover term (using fat links). Hence we may decompose

the pseudofermionic derivative into three terms also. The first comes from the Dirac term,

$$\frac{\partial S_{\rm pf}^{\rm d}}{\partial U_{\mu,x}} = \frac{1}{2u_0} \mathrm{Tr}_{\rm spin}(\eta_x^\dagger \otimes \gamma_\mu \chi_{x+\mu} + \chi_x^\dagger \otimes \gamma_\mu \eta_{x+\mu}), \tag{6.45}$$

while the Wilson and clover terms only explicit dependence is on the smeared links,

$$\frac{\partial S_{\rm pf}^{\rm w}}{\partial U_{\mu,x}^{\rm fl}} = -\frac{1}{2u_0^{\rm fl}} \mathrm{Tr}_{\rm spin}(\eta_x^\dagger \otimes \chi_{x+\mu} + \chi_x^\dagger \otimes \eta_{x+\mu}) \tag{6.46}$$

$$\frac{\partial S_{\rm pf}^{\rm cl}}{\partial U_{\mu,x}^{\rm fl}} = -\frac{1}{4u_0^{\rm fl4}} \mathrm{Tr}_{\rm spin}(\eta_y^\dagger \sigma_{\nu\lambda} \frac{\partial F_{\nu\lambda,y}}{\partial U_{\mu,x}^{\rm fl}} \chi_y + \chi_y^\dagger \sigma_{\nu\lambda} \frac{\partial F_{\nu\lambda,y}}{\partial U_{\mu,x}^{\rm fl}} \eta_y), \tag{6.47}$$

where the vector outer product defines a matrix $(\eta \otimes \chi^\dagger)^i{}_j = \eta^i \chi_j^*$. For convenience, we choose to absorb a factor of $-i$ from $F_{\mu\nu}^{\rm cl}$ into the definition of $\sigma_{\mu\nu}$, such that $\sigma_{\mu\nu} = \frac{1}{2}[\gamma_\mu, \gamma_\nu]$. The contribution to the derivative due to the clover term is then

$$\frac{\partial F_{\nu\lambda,y}}{\partial U_{\mu,x}^{\rm fl}} = \frac{1}{8}\Big(I \otimes \Box\, \delta^{\nu\mu}_{y,x} + \Box \otimes \Box\, \delta^{\nu\mu}_{y,x+\lambda} - I \otimes \Box\, \delta^{\nu\mu}_{y,x} - \Box \otimes \Box\, \delta^{\nu\mu}_{y,x-\lambda}$$
$$+ \Box \otimes I\, \delta^{\nu\mu}_{y,x+\mu} + \Box \otimes \Box\, \delta^{\nu\mu}_{y,x+\mu+\lambda} - \Box \otimes \Box\, \delta^{\nu\mu}_{y,x+\mu-\lambda} - \Box \otimes I\, \delta^{\nu\mu}_{y,x+\mu} \Big), \tag{6.48}$$

where we note that the derivative is zero unless either $\nu = \mu$ or $\lambda = \mu$ and as $\nu \neq \lambda$ we have without loss of generality chosen $\mu = \nu$ to be the horizontal direction and λ to be the transverse (vertical) direction.

6.4.3 Smeared Link Derivatives

Now, having constructed the explicit derivatives of $S_{\rm pf}$ with respect to the thin and fat links, the total derivative with respect to the thin links is

$$\frac{{\rm d}S_{\rm pf}}{{\rm d}U_{\mu,x}} = \frac{\partial S_{\rm pf}}{\partial U_{\mu,x}} + \frac{\partial S_{\rm pf}}{\partial U_{\nu,y}^{\rm fl}} \star \frac{{\rm d}U_{\nu,y}^{\rm fl}}{{\rm d}U_{\mu,x}}. \tag{6.49}$$

If we have performed n sweeps of APE smearing to form the fat links, then the right hand term is constructed through n applications of the chain rule

$$\frac{{\rm d}S}{{\rm d}U_{\mu,x}^{(j-1)}} = \frac{\partial S}{\partial U_{\nu,y}^{(j)}} \star \frac{\partial U_{\nu,y}^{(j)}}{\partial U_{\mu,x}^{(j-1)}} + \frac{\partial S}{\partial U_{\nu,y}^{(j)\dagger}} \star \frac{\partial U_{\nu,y}^{(j)\dagger}}{\partial U_{\mu,x}^{(j-1)}}, \tag{6.50}$$

until we arrive at $\frac{\mathrm{d}S}{\mathrm{d}U^{(0)}_{\mu,x}}$. For the sake of both computational efficiency and simplicity, this chain rule is itself composed of several chain rules, and hence evaluated in several steps. Each step corresponds to a step in the APE smearing process, but we go through them in reverse order.

The final step in the APE smearing process (with unit circle projection) is

$$U^{(n)}_{\mu,x} = \frac{1}{\det W^{(n)-\frac{1}{3}}_{\mu,x}} W^{(n)}_{\mu,x}. \tag{6.51}$$

Therefore the first chain rule corresponds to this step,

$$\begin{aligned}\frac{\partial S}{\partial W_{\mu,x}} &= \frac{\partial S}{\partial U^{(n)}_{\mu,x}} \star \frac{\partial U^{(n)}_{\mu,x}}{\partial W_{\mu,x}} \\ &= \frac{\partial S}{\partial U^{(n)}_{\mu,x}} \star \left(-\frac{1}{3} \det W^{-\frac{4}{3}}_{\mu,x} \frac{\partial \det W_{\mu,x}}{\partial W_{\mu,x}} \oplus W_{\mu,x} + \det W^{-\frac{1}{3}}_{\mu,x} I \otimes I \right),\end{aligned} \tag{6.52}$$

where

$$\det A = \epsilon^{ijk} A^1{}_i A^2{}_j A^3_k \tag{6.53}$$

and hence denoting the permutations $\pi_i = (i \bmod 3) + 1, \pi_i^2 = \pi_{\pi_i}$,

$$\frac{\partial \det A}{\partial A^i{}_j} = \epsilon_{jlm} A^{\pi_i}{}_l A^{\pi_i^2}{}_m. \tag{6.54}$$

There are several chain rules that correspond to

$$W^{(n)}_{\mu,x} = V^{(n)}_{\mu,x} (V^{(n)\dagger}_{\mu,x} V^{(n)}_{\mu,x})^{-\frac{1}{2}}. \tag{6.55}$$

For the first, we define $H_{\mu,x} = V^\dagger_{\mu,x} V_{\mu,x}$. Then

$$\frac{\partial S}{\partial H_{\mu,x}} = \frac{\partial S}{\partial W_{\mu,x}} \star \frac{\partial W_{\mu,x}}{\partial H_{\mu,x}}. \tag{6.56}$$

Using

$$W_{\mu,x} \approx d_0 V_{\mu,x} (H_{\mu,x} + c_0) \sum_{l=1}^{k} \frac{b_l}{H_{\mu,x} + c_l}, \tag{6.57}$$

we have

$$\frac{\partial W_{\mu,x}}{\partial H_{\mu,x}} = d_0 V_{\mu,x} \left(I \otimes \sum_{l=1}^{k} \frac{b_l}{H_{\mu,x} + c_l} - (H_{\mu,x} + c_0) \sum_{l=1}^{k} b_l \frac{1}{H_{\mu,x} + c_l} \otimes \frac{1}{H_{\mu,x} + c_l} \right). \tag{6.58}$$

We can then construct

$$\begin{aligned}\frac{\partial S}{\partial V_{\mu,x}} &= \frac{\partial S}{\partial W_{\mu,x}} \star \frac{\partial W_{\mu,x}}{\partial V_{\mu,x}} + \frac{\partial S}{\partial H_{\mu,x}} \star \frac{\partial H_{\mu,x}}{\partial V_{\mu,x}} \\ &= \frac{\partial S}{\partial W_{\mu,x}} \star (I \otimes H^{-\frac{1}{2}}_{\mu,x}) + \frac{\partial S}{\partial H_{\mu,x}} \star (V^\dagger_{\mu,x} \otimes I),\end{aligned} \tag{6.59}$$

and also

$$\frac{\partial S}{\partial V^{\dagger}_{\mu,x}} = \frac{\partial S}{\partial W^{\dagger}_{\mu,x}} \star \frac{\partial W^{\dagger}_{\mu,x}}{\partial V^{\dagger}_{\mu,x}} + \frac{\partial S}{\partial H_{\mu,x}} \star \frac{\partial H_{\mu,x}}{\partial V^{\dagger}_{\mu,x}}$$

$$= \frac{\partial S}{\partial W^{\dagger}_{\mu,x}} \star (H^{-\frac{1}{2}}_{\mu,x} \otimes I) + \frac{\partial S}{\partial H_{\mu,x}} \star (I \otimes V_{\mu,x}). \tag{6.60}$$

Last, we make use of the chain rule

$$\frac{\partial S}{\partial U^{(n-1)}_{\mu,x}} = \frac{\partial S}{\partial V_{\nu,y}} \star \frac{\partial V_{\nu,y}}{\partial U^{(n-1)}_{\mu,x}} + \frac{\partial S}{\partial V^{\dagger}_{\nu,y}} \star \frac{\partial V^{\dagger}_{\nu,y}}{\partial U^{(n-1)}_{\mu,x}}, \tag{6.61}$$

where

$$V_{\nu,y} = (1-\alpha) \longrightarrow + \frac{\alpha}{6} \sum_{\lambda \neq \nu} \uparrow\rightarrow\downarrow + \downarrow\rightarrow\uparrow, \tag{6.62}$$

$$V^{\dagger}_{\nu,y} = (1-\alpha) \longleftarrow + \frac{\alpha}{6} \sum_{\lambda \neq \nu} \uparrow\leftarrow\downarrow + \downarrow\leftarrow\uparrow. \tag{6.63}$$

It is then straightforward to show that

$$\frac{\partial V_{\nu,y}}{\partial U^{(n-1)}_{\mu,x}} = (1-\alpha) I \otimes I + \frac{\alpha}{6} \sum_{\lambda \neq \nu} \uparrow \otimes \downarrow \; \delta^{\mu\nu}_{y,x-\lambda} + \downarrow \otimes \uparrow \; \delta^{\mu\nu}_{y,x+\lambda}$$

$$+ I \otimes \rightarrow\downarrow \; \delta^{\mu\lambda}_{y,x} + \downarrow\rightarrow \otimes I \; \delta^{\mu\lambda}_{y,x+\lambda-\nu}.$$

and

$$\frac{\partial V^{\dagger}_{\nu,y}}{\partial U^{(n-1)}_{\mu,x}} = \frac{\alpha}{6} \sum_{\lambda \neq \nu} I \otimes \downarrow\leftarrow \; \delta^{\mu\lambda}_{y,x-\nu} + \leftarrow\uparrow \otimes I \; \delta^{\mu\lambda}_{y,x+\lambda}. \tag{6.64}$$

Hence,

$$\frac{\partial S}{\partial U^{(n-1)}_{\mu,x}} = (1-\alpha) \frac{\partial S}{\partial V_{\mu,x}} + \frac{\alpha}{6} \sum_{\nu \neq \mu} \frac{\partial S}{\partial V_{\mu,x-\nu}} \star \uparrow \otimes \downarrow + \frac{\partial S}{\partial V_{\mu,x+\nu}} \star \downarrow \otimes \uparrow$$

$$+ \frac{\partial S}{\partial V_{\nu,x}} \star I \otimes \rightarrow\downarrow + \frac{\partial S}{\partial V_{\nu,x+\mu-\nu}} \star \downarrow\rightarrow \otimes I + \frac{\partial S}{\partial V^{\dagger}_{\mu,x-\nu}} \star I \otimes \downarrow\leftarrow + \frac{\partial S}{\partial V^{\dagger}_{\mu,x+\mu}} \star \leftarrow\uparrow \otimes I. \tag{6.65}$$

Having constructed the total derivative of the action with respect to $U_{\mu,x}$, we can calculate the variation of S with respect to the gauge field,

$$\frac{\delta S}{\delta U} = \frac{\mathrm{d}S}{\mathrm{d}U} - U^{\dagger} \frac{\mathrm{d}S}{\mathrm{d}U^{\dagger}} U^{\dagger} = \frac{\mathrm{d}S}{\mathrm{d}U} - U^{\dagger} \left(\frac{\mathrm{d}S}{\mathrm{d}U} \right)^{\dagger} U^{\dagger}, \tag{6.66}$$

and hence the necessary equations of motion, (6.7) and (6.8).

6.5 Simulation Results

The implementation of HMC as applied to FLIC fermions is given in Appendix A. We have implemented a modified version of the Ritz algorithm to diagonalise arrays of 3×3 matrices in parallel. This routine is used in the SU(3) projection step, and is also used to calculate the matrix exponentials that are needed in other parts of the algorithm, avoiding the need to use polynomial approximations to the exponential. In particular, this means that the accuracy of the exponential in Eq. (6.7) no longer depends upon the step size $\Delta\tau$. We have implemented a standard two-flavour HMC, with multiple time scales. Expensive pseudo-fermion momenta updates are performed at a larger step size $\Delta\tau = \Delta\tau_{\rm pf}$ and the cheaper gauge momenta updates are performed more often, $\Delta\tau_{\rm g} = \frac{1}{n}\Delta\tau_{\rm pf}$, for some integer n. Molecular dynamics trajectories are of unit length, $n_{\rm md}\Delta\tau = 1$.

An eighth order Zolotarev approximation to the inverse square root is used to approximate $W_{\mu,x}$ in unit circle projection. We find that the spectral range at this order is ample. In smooth gauge backgrounds it is easily shown that unit circle projection is well defined, that is $\det V^\dagger_{\mu,x}V_{\mu,x} > 0$.

Lemma 6.5.1. *Given* $||1 - U_{\mu\nu,x}|| \leq \epsilon \; \forall x, \mu, \nu$, *we have a lower bound*

$$V^\dagger_{\mu,x}V_{\mu,x} \geq 1 - 2\alpha\epsilon - \alpha^2\epsilon^2.$$

Proof. First we note that APE blocking may be written in terms of the plaquette field,

$$V_{\mu,x} = U_{\mu,x}\big(1 - \frac{\alpha}{6}\sum_{\pm\nu\neq\mu}(1 - U^\dagger_{\mu\nu,x})\big). \tag{6.67}$$

Define $Z = \sum_{\pm\nu\neq\mu}(1 - U^\dagger_{\mu\nu,x})$, then

$$V^\dagger_{\mu,x}V_{\mu,x} = 1 - \frac{\alpha}{6}(Z + Z^\dagger) + \frac{\alpha^2}{36}Z^\dagger Z. \tag{6.68}$$

As $||Z|| \leq 6\epsilon$, we then have

$$\lambda_{\rm min}(V^\dagger_{\mu,x}V_{\mu,x}) \geq 1 - 2\alpha\epsilon - \alpha^2\epsilon^2, \tag{6.69}$$

as required. □

While the smeared link equations of motions are complex, our implementation evaluates them efficiently due to the optimisations that can be performed through the calculus we constructed earlier. At large sea quark masses the code already spends over 90% of its time in the CG inversion required to calculate $\eta = (D^\dagger D)^{-1}\phi$, and as the quark mass decreases this fraction increases. So as is standard, the generation of dynamical gauge fields is dominated by the CG inversion. Simulation results are presented in Table 6.2.

β	κ	S_{gauge}	$\Delta\tau$	$\frac{\Delta\tau_{\text{pf}}}{\Delta\tau_{\text{g}}}$	ρ_{acc}	u_0	a	m_π
3.6	0.1347	IMP	0.0143	2	0.55	0.8226	0.247(9)	0.702
3.7	0.1340	IMP	0.0147	2	0.64	0.8338	0.218(4)	0.680
3.8	0.1332	IMP	0.0151	2	0.65	0.8443	0.180(2)	0.738
3.9	0.1310	IMP	0.0200	2	0.66	0.8534	0.153(2)	0.834
3.9	0.1325	IMP	0.0156	2	0.55	0.8540	0.146(2)	0.702
4.0	0.1301	IMP	0.0200	2	0.66	0.8614	0.132(2)	0.906
4.0	0.1318	IMP	0.0161	2	0.64	0.8625	0.121(2)	0.799
4.1	0.1283	IMP	0.0200	2	0.75	0.8680	0.114(1)	1.088
4.1	0.1305	IMP	0.0166	2	0.70	0.8685	0.104(1)	0.668
4.2	0.1246	IMP	0.0200	2	0.86	0.8736	0.107(1)	1.496
4.2	0.1266	IMP	0.0200	2	0.80	0.8738	0.097(1)	1.346
4.3	0.1253	IMP	0.0200	2	0.83	0.8788	0.091(1)	1.574
4.4	0.1255	IMP	0.0200	2	0.88	0.8836	0.086(1)	1.411
4.5	0.1253	IMP	0.0200	2	0.83	0.8878	0.075(1)	1.657
4.6	0.1254	IMP	0.0200	2	0.84	0.8916	0.072(1)	1.617
7.0	0.1315	DBW2	0.0152	2	0.74	0.8344	0.252(6)	0.780
7.0	0.1345	DBW2	0.0156	2	0.68	0.8352	0.233(8)	0.673
7.5	0.1310	DBW2	0.0156	2	0.79	0.8516	0.206(3)	0.779
8.0	0.1305	DBW2	0.0161	2	0.73	0.8663	0.168(2)	0.764
8.5	0.1300	DBW2	0.0166	3	0.71	0.8774	0.134(1)	0.782
9.0	0.1224	DBW2	0.0200	2	0.79	0.8858	0.137(3)	1.412
9.0	0.1296	DBW2	0.0200	2	0.78	0.8865	0.115(1)	0.753
9.5	0.1228	DBW2	0.0200	2	0.82	0.8934	0.109(2)	1.576
10.0	0.1234	DBW2	0.0200	2	0.83	0.9000	0.099(2)	1.502
10.5	0.1236	DBW2	0.0200	2	0.79	0.9056	0.093(1)	1.567
11.0	0.1239	DBW2	0.0200	2	0.81	0.9110	0.086(1)	1.473

Table 6.2: Simulation parameters and results for various dynamical simulations. The parameters are the gauge coupling, hopping parameter, gauge action, step size, and psuedofermion to gauge step size ratio. The results given are the mean link, lattice spacing (in fm) and pion mass (in GeV). Two degenerate flavours of FLIC sea quarks are used, with either Lüscher-Weisz (IMP) glue or DBW2 glue. These results are obtained from 20 $12^3 \times 24$ configurations. Simulations are done using multiple time step HMC with trajectories of unit length. The lattice spacing is set by r_0 via the static quark potential. Although an exact comparison is difficult, for a given step size and quark mass the acceptance rates obtained compare well with standard simulations (see for example Ref. [74]).

It is straightforward to apply our results to generate gauge fields with dynamical FLIC Overlap quarks, although this would be extremely computationally intensive. The availability of HMC as a simulation algorithm for dynamical FLIC fermions is significant, as it scales almost linearly with the lattice volume V, whereas previously there were only $O(V^2)$ alternatives [75]. Furthermore, the method we have described is general and can be applied to any fermion action with reuniterisation, including overlap fermions with a fat link kernel [76, 77, 78, 79], or other types of fat link actions [80] that may involve alternative smearing techniques [81]. Additionally, any of the variants of HMC can also be used, in particular Polynomial HMC[82] or Rational HMC[83] which allow for the simulation of odd numbers of sea quark flavours.

6.6 Projection Analysis

In this section we conduct a comparison of three different SU(3) projection techniques. Given a matrix X, the three different SU(3) matrices U_A, U_B, U_C are defined by

$$(\mathrm{A})\ U_A \in \{U' \in SU(3) |\, \mathrm{Re}\,\mathrm{Tr}(U'X^\dagger) = \max_{U \in SU(3)} \mathrm{Re}\,\mathrm{Tr}(UX^\dagger)\}$$

$$(\mathrm{B})\ U_B = (\det W_B)^{-\frac{1}{3}} W_B, W_B = X[X^\dagger X]^{-\frac{1}{2}} \in U(3)$$

$$(\mathrm{C})\ U_C = (\det W_C)^{-\frac{1}{3}} W_C, W_C = [XX^\dagger]^{-\frac{1}{2}} X \in U(3)$$

Once again, the principal value of the cube root is chosen. The gauge covariance of (B) and (C) is easily verified, as we have seen. While the measure in (A) is gauge invariant, the gauge covariance of (A) is only assured if there is a unique matrix in SU(3) that gives the maximum trace. This is easily understood: suppose the set defined in (A) contains more than one element. Given X, suppose we choose an element V of the set using some method (such as iterative SU(2) trace maximisation). Then there is no *a priori* way of guaranteeing that if we start instead from $X' = G(x)XG(x+\mu)$ that we will choose $V' = G(x)VG(x+\mu)$. Thus we regard this as something to be tested and if possible, proven.

A numerical comparison is straightforwardly done by calculating the difference between the matrices produced by the different methods using a simple measure,

$$||U - V|| = \sqrt{\sum_{i,j} |U_{ij} - V_{ij}|^2}. \tag{6.70}$$

Our results indicate that $||U_A - U_B|| \approx ||U_A - U_C|| \approx 10^{-3}$, while $||U_B - U_C|| = 0$ to within rounding errors. The gauge covariance of (A) was tested and was found to be satisfied to a precision that increased exponentially as the number of iterations of SU(2) maximisation was increased. That is, if we perform 8

sweeps of 3 SU(2) maximisations, we have gauge covariance to 8 significant figures, and so on. This is the first possible indication that method (A) selects a unique matrix. Furthermore, two different starting points for method (A) were found to converge to the same matrix. This is the second possible indication that the method selects a unique matrix.

An interesting point is that if we were in $U(3)$, then the three methods are related.

Lemma 6.6.1. *The matrices W_B and W_C satisfy:*

$$W_B \in \{V \in U(N) | \operatorname{Re}\operatorname{Tr}(V^\dagger X) = \max_{U \in U(N)} \operatorname{Re}\operatorname{Tr}(U^\dagger X)\}, \tag{6.71}$$

$$W_C \in \{V \in U(N) | \operatorname{Re}\operatorname{Tr}(V X^\dagger) = \max_{U \in U(N)} \operatorname{Re}\operatorname{Tr}(U X^\dagger)\}. \tag{6.72}$$

Proof. Let X be an arbitrary matrix of dimension N, $\det X \neq 0$. Then $XX^\dagger$ is Hermitian and can be diagonalised into an orthonormal basis. Let the basis of eigenvectors of $XX^\dagger$ be v_i, each having eigenvalue λ_i.

Consider $\operatorname{Re}\operatorname{Tr}(UX^\dagger), U \in U(N)$. The trace of a matrix is basis invariant, so let us choose to work in the basis v_i. In this basis,

$$\operatorname{Re}\operatorname{Tr}(UX^\dagger) = \operatorname{Re}\sum_i [UX^\dagger]_{ii} = \operatorname{Re}\sum_i \langle v_i, UX^\dagger v_i\rangle. \tag{6.73}$$

Define $\varphi_i = \langle v_i, UX^\dagger v_i\rangle$ so that $\operatorname{Re}\operatorname{Tr}(UX^\dagger) = \operatorname{Re}\sum_i \varphi_i$. The Cauchy-Schwarz inequality for two vectors a and b says $|\langle a, b\rangle| \leq ||a||||b||$. Thus

$$\begin{aligned} |\langle v_i, UX^\dagger v_i\rangle| &\leq ||v_i||||UX^\dagger v_i|| = \sqrt{\langle UX^\dagger v_i, UX^\dagger v_i\rangle} \\ &= \sqrt{\langle v_i, XU^\dagger UX^\dagger v_i\rangle} = \sqrt{\langle v_i, XX^\dagger v_i\rangle} = \sqrt{\lambda_i}, \end{aligned} \tag{6.74}$$

where we have used the orthonormality of v_i. Then we have

$$\operatorname{Re}\operatorname{Tr}(UX^\dagger) \leq |\operatorname{Tr}(UX^\dagger)| \leq \sum_i |\varphi_i| \leq \sum_i \sqrt{\lambda_i}. \tag{6.75}$$

Thus we have an upper bound for $\operatorname{Re}\operatorname{Tr}(UX^\dagger)$. In fact this is a least upper bound, which we show by constructing a matrix that saturates the bound. We make use of our choice of basis to obtain

$$\begin{aligned} \operatorname{Re}\operatorname{Tr}(W_C X^\dagger) &= \operatorname{Re}\operatorname{Tr}([XX^\dagger]^{-\frac{1}{2}} XX^\dagger) \\ &= \operatorname{Re}(\sum_i \frac{\lambda_i}{\sqrt{\lambda_i}}) = \sum_i \sqrt{\lambda_i}, \end{aligned} \tag{6.76}$$

thus proving

$$W_C \in \{V \in U(N) | \operatorname{Re}\operatorname{Tr}(V X^\dagger) = \max_{U \in U(N)} \operatorname{Re}\operatorname{Tr}(U X^\dagger)\}. \tag{6.77}$$

The proof of

$$W_B \in \{V \in U(N) | \operatorname{Re}\operatorname{Tr}(V^\dagger X) = \max_{U \in U(N)} \operatorname{Re}\operatorname{Tr}(U^\dagger X)\} \tag{6.78}$$

follows by exact analogy. □

Now in fact these two sets are equal, as $\operatorname{Re}\operatorname{Tr} A^\dagger = \operatorname{Re}\operatorname{Tr} A \Rightarrow \operatorname{Re}\operatorname{Tr} U^\dagger X = \operatorname{Re}\operatorname{Tr} X^\dagger U = \operatorname{Re}\operatorname{Tr} U X^\dagger$. So if the sets defined above contain only one element, we have that $U_B = U_C$. Another sufficient condition for these two matrices to be equivalent is that $[X, X^\dagger] = 0$.

The case of SU(N) is different because while the upper bound still holds, it may not be a least upper bound. W_C is unitary, so let $\det W_C = e^{i\phi}$. Then $U_C = e^{-i\phi/N} W_C$ is special unitary. In this case,

$$\operatorname{Re}\operatorname{Tr}(U_C X^\dagger) = \sum_i \cos(-\phi/N)\sqrt{\lambda_i}, \tag{6.79}$$

which may not be the maximal trace in $SU(N)$. In fact our numerical results indicate that it is not, as method (A) selects an SU(3) matrix with a greater trace. This would explain why (A) is not equivalent to either (B) or (C).

Having seen that the first method is not equivalent to the other two, the question of the gauge covariance of (A) remains. We have numerically verified that we can get this to within roundoff if we perform enough SU(2) maximisations, but the covariance is only proven if we can show that the matrix U_A is unique. We proceed to do this.

Lemma 6.6.2. *The set defined in (A) contains exactly one element.*

Proof. Consider method (C) and the basis of eigenvectors of $XX^\dagger$. Suppose we have ordered the eigenvectors so that $\lambda_i \geq \lambda_j, i \leq j$. Now in SU(3) if we have some element V then we can always get to another element V' by choosing a suitable SU(3) matrix U such that $V' = VU$. Now let $V = U_C$ and let V' be in the set defined in (A). Then we are trying to prove the uniqueness of $\max \operatorname{Re}\operatorname{Tr}(V' X^\dagger) = \max \operatorname{Re}\operatorname{Tr}(U U_C X^\dagger)$. In our chosen basis $U_C X^\dagger$ is diagonal,

$$\begin{aligned} \operatorname{Re}\operatorname{Tr}(U U_C X^\dagger) &= \operatorname{Re}\operatorname{Tr}(U e^{-i\phi/3} \operatorname{diag}[\sqrt{\lambda_1}, \sqrt{\lambda_2}, \sqrt{\lambda_3}]) \\ &= \operatorname{Re}\operatorname{Tr}(e^{-i\phi/3}(U_{11}\sqrt{\lambda_1} + U_{22}\sqrt{\lambda_2} + U_{33}\sqrt{\lambda_3})). \end{aligned} \tag{6.80}$$

It is simple to see that the off-diagonal elements of U will not contribute. As $U \in SU(3)$, its rows and columns form normalised vectors,

$$\sum_i |U_{ij}|^2 = 1 \ \forall j = 1, 2, 3. \tag{6.81}$$

Let $U_{ii} = r_i e^{i\theta_i}$. Then the above equation implies $r_i \leq 1$. The function we are trying to maximise is

$$f(r_i, \theta_i) = \sum_i \cos(\theta_i - \phi/3) r_i \sqrt{\lambda_i}. \tag{6.82}$$

Suppose we have a solution, $r_i = a_i, \theta_i = \alpha_i$, with $0 < a_i < 1$. For each positive cosine, let $a_i' = a_i + \delta$, and for each negative cosine, let $a_i' = a_i - \delta$, for some small δ. As f is independent of the off diagonal elements, we can correspondingly adjust them to remain within $SU(3)$. But then $f(a_i', \theta_i) > f(a_i, \theta_i)$ so we cannot have a maximum. Hence we require for our solution $r_i \in \{0, 1\}$. Suppose we have one non-zero r_i, for $i = j$. Then the maximum is unique, $\theta_j = \phi/3$.

Due to the normalisation condition (6.81) we cannot have two of the r_i non-zero and one zero, as the matrix would then leave $SU(3)$. So all that remains is for all three $r_i = 1$. In this case U is diagonal (in this basis) and we can parameterise it as $U = \mathrm{diag}[e^{i\theta_1}, e^{i\theta_2}, e^{-i(\theta_1+\theta_2)}]$. We need to show that in this case $\max f(\theta_i)$ is unique, that is, the following function has a unique maximum,

$$f(\theta_i) = \cos(\theta_1 - \phi/3)\sqrt{\lambda_1} + \cos(\theta_2 - \phi/3)\sqrt{\lambda_2} + \cos(\theta_1 + \theta_2 + \phi/3)\sqrt{\lambda_3}. \tag{6.83}$$

The global maximum of this function is either at a turning point or at a boundary, but as the function is periodic it must be at a turning point. Defining $\theta_i' = \theta_i - \phi/3$ and differentiating,

$$\frac{\partial f}{\partial \theta_i'} = 0, \tag{6.84}$$

implies that at the turning points the parameters satisfy

$$\sin(\theta_1')\sqrt{\lambda_1} = \sin(\theta_2')\sqrt{\lambda_2}, \tag{6.85}$$

$$\sin(\theta_2') + \sin(\theta_1' + \theta_2' + \phi)\sqrt{\frac{\lambda_3}{\lambda_2}} = 0. \tag{6.86}$$

The first equation has two solutions $\theta_1' = \xi, \pi - \xi$, where $\xi = \arcsin(\sin(\theta_2')\sqrt{\frac{\lambda_2}{\lambda_1}})$, providing $\xi \neq \pm\frac{\pi}{2}$. Only the solution which has a positive cosine will give a global maximum, so we have determined θ_1' uniquely in terms of θ_2'. In the case $\xi = \pm\frac{\pi}{2}$, there is only one solution.

The second equation will determine θ_2'. As $\lambda_3 \leq \lambda_2$, at $\theta_2' = \pi/2$ the $LHS \geq 0$ and at $\theta_2' = -\pi/2$ the $LHS \leq 0$ so by continuity there must exist at least one zero in $[-\pi/2, \pi/2]$, (that is where $\cos\theta_2'$ is positive). Equation (6.86) is equivalent to a 6th order polynomial in $\sin\theta_2'$, easily seen through the use of the trigonometric expansion for $\sin(\theta_1' + \theta_2' + \phi)$ and squaring twice, once for $\cos\theta_1'$ and once for $\cos\theta_2'$. The polynomial is

$$(a_3 \sin^3\theta_2 + a_2 \sin^2\theta_2 + a_0)^2 - (b_2 \sin^2\theta_2 + b_1 \sin\theta_2)^2(1 - b_0 \sin^2\theta_2) = 0, \tag{6.87}$$

where

$$a_0 = -\lambda_3 \sin^2_\phi, \qquad b_0 = \frac{\lambda_2}{\lambda_1}, \tag{6.88}$$

$$a_2 = \lambda_2 + \lambda_3 - \frac{\lambda_3\lambda_2}{\lambda_1}\cos 2\phi, \qquad b_1 = \lambda_3\sqrt{\frac{\lambda_2}{\lambda_1}}\sin 2\phi, \tag{6.89}$$

$$a_3 = -2\lambda_2\sqrt{\frac{\lambda_3}{\lambda_1}}\sin\phi, \qquad b_2 = -2\sqrt{\lambda_3\lambda_2}\cos\phi. \tag{6.90}$$

This means that there are at most 6 solutions for $\sin\theta_2'$. However we have squared the original equation twice before arriving at this polynomial, so not all of our roots will be solutions to eq. (6.86). Denote eq. (6.86) as $g(s_1, s_2, c_1, c_2) = 0$, where $s_i = \sin\theta_i'$, $c_i = \cos\theta_i'$, and s_1, c_1, c_2 are all determined by s_2. By squaring twice we are now solving $g'(s_1, s_2, c_1^2, c_2^2) = 0$. So for every distinct solution (s_1, s_2, c_1, c_2) of $g = 0$ we have 3 additional solutions of $g' = 0$, $(s_1, s_2, c_1, -c_2)$, $(s_1, s_2, -c_1, c_2)$, $(s_1, s_2, -c_1, -c_2)$, unless either $c_1 = 0$ or $c_2 = 0$ in which case we have only one additional solution. We have established there is at least one real solution to g, and as there are at most 6 real solutions to g' this means that the distinct non-trivial (one of $c_i \neq 0$) solutions to g must number at most three.

In fact, to obtain a global maximum we must have both $c_i \neq 0$. This is straightforwardly seen. Suppose $c_1 = 0$, that is $\theta_1' = \pm\frac{\pi}{2}$. Then set

$$\begin{aligned} f_1 = f(\theta_1' \to \pm\frac{\pi}{2}, \theta_2') &= \cos\theta_2'\sqrt{\lambda_2} + \cos(\theta_2' + \phi \pm \frac{\pi}{2})\sqrt{\lambda_3} \\ &= \cos\theta_2'\sqrt{\lambda_2} \mp \sin(\theta_2' + \phi)\sqrt{\lambda_3}. \end{aligned} \tag{6.91}$$

Now, we define

$$\begin{aligned} f_3 = f(\theta_1' \to -\theta_2' - \phi \pm \frac{\pi}{2}, \theta_2') &= \cos(-\theta_2' - \phi \pm \frac{\pi}{2})\sqrt{\lambda_1} + \cos\theta_2'\sqrt{\lambda_2} \\ &= \pm\sin(\theta_2' + \phi)\sqrt{\lambda_1} + \cos\theta_2'\sqrt{\lambda_2}. \end{aligned} \tag{6.92}$$

Then as $\lambda_1 \geq \lambda_3$, we can choose the appropriate sign of $\pm\frac{\pi}{2}$ such that $f_3 \geq f_1$. In the non-degenerate case, $\lambda_1 > \lambda_3$, we must have $f_3 > f_1$, and hence that $c_1 = 0$ cannot give a global maximum. The argument for $c_2 = 0, \lambda_2 > \lambda_3$ follows similarly. Hence, excluding the degenerate cases, the global maximum must satisfy both $c_i \neq 0$ and there is at most one such solution. As we have an upper bound we know a global maximum exists and thus there is exactly one.

Finally, we consider the two possible degenerate cases. As the λ_i are ordered, if $\lambda_1 = \lambda_3$ the $\lambda_2 = \lambda_1$ also, and maximum of f is unique, $\theta_1' = \theta_2' = -\frac{\phi}{3}$. In the second case we consider $\lambda_1 > \lambda_2, \lambda_2 = \lambda_3$. It is clear that $c_1 = 0$ will not give a global maximum. So we need to maximise

$$\begin{aligned} f(\theta_i') &= \cos\theta_1'\sqrt{\lambda_1} + \big(\cos\theta_2' + \cos(\theta_1' + \theta_2' + \phi)\big)\sqrt{\lambda_2} \\ &= \cos\theta_1'\sqrt{\lambda_1} + 2\cos\frac{1}{2}(\theta_1' + 2\theta_2' + \phi)\cos\frac{1}{2}(\theta_1' + \phi)\sqrt{\lambda_2}. \end{aligned} \tag{6.93}$$

Clearly, the maximum will be obtained when $\theta'_2 = -\frac{1}{2}(\theta'_1 + \phi)$. If $c_2 \neq 0$ then this maximum is unique. If $c_2 = 0$ then $\theta'_1 + \phi = \pm\pi$ and hence $\cos\frac{1}{2}(\theta'_1 + \phi) = 0$, giving

$$f(\theta'_i) = -\cos\phi\sqrt{\lambda_1}. \tag{6.94}$$

One can easily obtain a value of f greater than this, simply choose $\theta'_1 = 0, \theta'_2 = 0$ to get $f = \sqrt{\lambda_1} + 2(\cos\frac{\phi}{2})^2\sqrt{\lambda_2}$, thus $c_2 = 0$ cannot give a global maximum. Hence for a global maximum $c_2 \neq 0$, and again there is at most one such solution. Having exhausted all the possibilities, we now conclude that the global maximum is unique, as required. Furthermore, we have shown that dividing by $\sqrt[3]{\det W}$ as in methods (B) and (C) is not sufficiently general to find the maximum, and to relate these to method (A) we need three parameters, θ_1, θ_2, and ϕ. □

Chapter 7

FLIC Overlap Properties

In this chapter we investigate the physical properties of FLIC overlap fermions. The structure of the quark propagator in momentum space is studied, and a comparison against previous studies of the Wilson overlap propagator in quenched QCD is performed. A brief investigation into topology is undertaken, examining the Atiyah-Singer index theorem on the lattice, and once again comparing with previous Wilson overlap results.

7.1 Quark Propagator in Momentum Space

The quark propagator in momentum space is a fundamental quantity in QCD. Its infrared structure gives us insight into the dynamical generation of mass due to the spontaneous breaking of chiral symmetry within QCD. Its ultraviolet behaviour allows us to obtain the running quark mass[84, 85]. Lattice QCD allows a direct probe of the non-perturbative quark propagator. The momentum space quark propagator has been studied previously using different gauge fixing and fermion actions[86, 87, 88, 89], including studies of the (standard) Wilson overlap in Landau gauge[90, 91]. We will be interested in comparing the Wilson overlap and FLIC overlap in quenched QCD, as well as studying the effects of (partially quenched) dynamical fermions on the structure of the propagator.

Recall from §1.3.2 that the tree-level ($A_\mu(x) = 0$) quark propagator is identified with the (Euclidean space) fermionic Greens function,

$$(\partial\!\!\!/ + m^0)\Delta_{\mathrm{f}}(x, y) = \delta^4(x - y), \tag{7.1}$$

where m^0 is the bare quark mass. In momentum space this equation is straighforwardly solved,

$$\tilde{\Delta}_{\mathrm{f}}(p) = \frac{1}{i p\!\!\!/ + m^0}. \tag{7.2}$$

Denote $S^{(0)}(p) \equiv \tilde{\Delta}_{\mathrm{f}}(p)$ to be the tree-level propagator in momentum space. Then in the presence of gauge field interactions, define $S_{\mathrm{bare}}(p)$ to be the (fourier

transform) of the (interacting) fermionic Green's function,

$$(\not{D} + m^0)\Delta_{\mathrm{f}}^{\mathrm{bare}}(x,y) = \delta^4(x-y). \tag{7.3}$$

We define the mass function $M(p)$ and the bare renormalisation function $Z(p)$ such that the bare quark propagator has the form

$$S_{\mathrm{bare}}(p) = \frac{Z(p)}{i\not{p} + M(p)}. \tag{7.4}$$

Then for the renormalisation point ζ, the renormalised quark propagator is given by

$$S_\zeta(p) = \frac{Z_\zeta(p)}{i\not{p} + M(p)} = Z_2(\zeta, a) S_{\mathrm{bare}}(p), \tag{7.5}$$

where $Z_\zeta(p)$ is the (ζ-dependent) renormalisation function, and $Z_2(\zeta, a)$ is the wave function renormalisation constant, which depends on ζ and the regulator parameter a. The a-dependence of S_{bare} is implicit. $Z_2(\zeta, a)$ is chosen such that

$$Z_\zeta(\zeta) = 1. \tag{7.6}$$

As $S_\zeta(p)$ is multiplicatively renormalisable, all of the ζ-dependence is contained within $Z_\zeta(p)$, that is, the mass function is ζ-independent.

7.1.1 The Overlap Propagator

In the continuum, it is easily seen that the massless quark propagator anti-commutes with γ_5,

$$\{\gamma_5, S^{\mathrm{c}}_{\mathrm{bare}}(p)\big|_{m^0=0}\} = 0. \tag{7.7}$$

A straightforward consequence of the Ginsparg-Wilson relation is the inverse of the overlap Dirac operator satisfies

$$\{\gamma_5, D_{\mathrm{o}}^{-1}\} = 2\gamma_5. \tag{7.8}$$

Recalling from eq. 4.19 that $\lim_{a\to 0} D_{\mathrm{o}} = \frac{1}{2m_{\mathrm{w}}}\not{D}$, where m_{w} is the dimensionless overlap regulator parameter, it is then natural to define the (external) massless bare overlap propagator on the lattice as[39, 92]

$$S_{\mathrm{bare}}(p)|_{m^0=0} \equiv \frac{1}{2m_{\mathrm{w}}}(D_{\mathrm{o}}^{-1} - 1), \tag{7.9}$$

as we then have that

$$\{\gamma_5, S_{\mathrm{bare}}(p)\big|_{m^0=0}\} = 0, \tag{7.10}$$

as in the continuum case. The massive (external) bare overlap propagator is defined as[92]

$$S_{\mathrm{bare}}(p) \equiv \frac{1}{2m_{\mathrm{w}}(1-\mu)}(D_{\mathrm{o}}^{-1}(\mu) - 1), \tag{7.11}$$

and with the identification

$$\mu = \frac{m^0}{2m_{\mathrm{w}}} \tag{7.12}$$

satisfies

$$S^{-1}_{\mathrm{bare}}(p) = S^{-1}_{\mathrm{bare}}(p)\big|_{m^0=0} + m^0. \tag{7.13}$$

Example 7.1.1. *Show this.*

Proof. Recall that the massive overlap Dirac operator is given by

$$D_{\mathrm{o}}(\mu) = (1-\mu)D_{\mathrm{o}} + \mu. \tag{7.14}$$

Define $D_{\mathrm{c}}(\mu)$ such that

$$D_{\mathrm{c}}^{-1}(\mu) = \frac{1}{1-\mu}(D_{\mathrm{o}}^{-1}(\mu) - 1). \tag{7.15}$$

Then it is simply shown that

$$D_{\mathrm{o}} = \frac{1-\mu D_{\mathrm{c}}^{-1}(\mu)}{(1-\mu)D_{\mathrm{c}}^{-1}(\mu)+1}, \tag{7.16}$$

and hence

$$D_{\mathrm{o}}^{-1} - 1 = \frac{(1-\mu)D_{\mathrm{c}}^{-1}(\mu)+1}{1-\mu D_{\mathrm{c}}^{-1}(\mu)} - 1. \tag{7.17}$$

Thus,

$$D_{\mathrm{c}}^{-1}(0) = \frac{D_{\mathrm{c}}^{-1}(\mu)}{1-\mu D_{\mathrm{c}}^{-1}(\mu)}, \tag{7.18}$$

which implies

$$D_{\mathrm{c}}(\mu) = D_{\mathrm{c}}(0) + \mu. \tag{7.19}$$

Simply noting that $D_{\mathrm{c}} = 2m_{\mathrm{w}}S^{-1}_{\mathrm{bare}}(p)$ is then sufficent to give the required result. □

In order to construct $M(p)$ and $Z(p)$ on the lattice, we first define $\mathcal{B}(p), \mathcal{C}_\mu(p)$ such that

$$S_{\mathrm{bare}}(p) = -i\not{\mathcal{C}}(p) + \mathcal{B}(p). \tag{7.20}$$

Then

$$\mathcal{C}_\mu(p) = \frac{i}{n_{\mathrm{s}}n_{\mathrm{c}}}\,\mathrm{Tr}[\gamma_\mu S_{\mathrm{bare}}(p)], \tag{7.21}$$

$$\mathcal{B}(p) = \frac{1}{n_{\mathrm{s}}n_{\mathrm{c}}}\,\mathrm{Tr}[S^{\mathrm{bare}}(p)], \tag{7.22}$$

where the trace is over colour and spinor indices only, and $n_{\mathrm{s}}, n_{\mathrm{c}}$ specify the dimension of the spinor and colour vector spaces. Now, define the functions

$B(p), C_\mu(p)$ such that the inverse of the bare (lattice) quark propagator has the form

$$S_{\text{bare}}^{-1}(p) = i\not{\mathcal{C}}(p) + \mathcal{B}(p). \tag{7.23}$$

Then it is easily seen that

$$C_\mu = \frac{\mathcal{C}_\mu}{\mathcal{C}^2 + \mathcal{B}^2}, \qquad B = \frac{\mathcal{B}}{\mathcal{C}^2 + \mathcal{B}^2}, \tag{7.24}$$

where as per usual $\mathcal{C}^2 = \mathcal{C} \cdot \mathcal{C}$.

The kinematical lattice momentum q_μ is defined such that at tree-level

$$(S^{(0)})^{-1}(p) = i\not{q} + m^0, \tag{7.25}$$

that is $q_\mu(p) = C_\mu^{(0)}(p), m^0 = B^{(0)}(p)$. We note that the simple form of these relations is one of the advantages of the overlap, as the absence of additive mass renormalisation prevents the need for having to perform any tree-level correction[90] (necessary for other fermions actions[86, 87, 88]), outside of identifying the correct momentum variable q. Now, we define $A(p)$ such that

$$S_{\text{bare}}^{-1}(p) = i\not{q}A(p) + B(p). \tag{7.26}$$

The mass function $M(p)$ and renormalisation function $Z(p)$ may then be straightforwardly constructed,

$$Z(p) = \frac{1}{A(p)}, \qquad M(p) = \frac{B(p)}{A(p)}. \tag{7.27}$$

7.1.2 Simulation Details

While FLIC overlap and Wilson overlap are both free of $O(a)$ errors, the actions may in general differ at $O(a^2)$, and as such there may be some difference in their properties at finite a. We conduct a comparison between the two actions by calculating the FLIC overlap propagator on the same set of quenched lattices as in the Wilson overlap studies[90, 91]. Furthermore, the FLIC overlap propagator with a dynamical kernel is calculated on two lattices to perform an exploratory study of the dynamical sector. All the lattices have approximately the same volume, and the partially quenched lattice spacings approximately match the quenched spacings. The details of all five lattices are given in Table 7.1.

Landau gauge is chosen for the gauge fixing. An improved gauge fixing scheme[93] is used, and a Conjugate Gradient Fourier Acceleration [94] algorithm is chosen to perform the gauge fixing. We use periodic boundary conditions in the spatial directions and anti-periodic in the time direction. With this choice of boundary conditions, for a lattice with dimensions of extent $n_\mu, \mu = x, y, z, t$, the available momenta p_μ has components

$$p_\mu = \pm\frac{2k\pi}{an_\mu} - \frac{k_t\pi}{an_\mu};\ k \in \mathbb{Z}, -\frac{n_\mu}{2} < k \leq \frac{n_\mu}{2}, k_t = 1 \text{ if } \mu = t, 0 \text{ otherwise}. \tag{7.28}$$

Volume	β	κ_{sea}	a (fm)	m_{sea}	u_0	Phys. Vol. (fm^4)
$8^3 \times 16$	4.286	0.0000	0.194	∞	0.8721	$1.552^3 \times 3.104$
$12^3 \times 24$	4.60	0.0000	0.125	∞	0.8888	$1.5^3 \times 3.00$
$16^3 \times 32$	4.80	0.0000	0.100	∞	0.8966	$1.6^3 \times 3.20$
$8^3 \times 16$	3.70	0.1340	0.207	120	0.8625	$1.656^3 \times 3.312$
$12^3 \times 24$	4.00	0.1318	0.124	162	0.8338	$1.488^3 \times 2.976$

Table 7.1: Parameters for the five different lattices. All use a tadpole improved Luscher Weisz gluon action. Lattice spacings are set by the string tension. Shown is the lattice volume, gauge coupling β, sea quark hopping parameter κ_{sea}, lattice spacing, sea quark mass (in MeV), the mean link and the physical volume.

At tree level the FLIC overlap and Wilson overlap have identical behaviour (although the choice of $m_{\text{w}} = 1.4$ for the FLIC overlap was slightly different from that chosen for the Wilson overlap). The tree level propagator is calculated directly by setting the links and the mean link to unity. The kinematical lattice momentum q is obtained numerically from the tree level propagator, although it could equally well have been obtained from the analytic form for q derived in [90].

Each lattice ensemble consists of 50 configurations. The lattice Green's function is obtained by inverting the FLIC overlap Dirac operator on each configuration using a multi-mass CG inverter[95]. The Fourier transform is taken to move to momentum space, and the bare quark propagator is obtained from the ensemble average. The quark propagator is calculated for 15 different masses. The details of the FLIC overlap parameters used are presented in Table 7.2.

7.1.3 Results

We now turn to the results of our investigation. The mass function $M(p)$ and the renormalisation function $Z_\zeta(p)$ are calculated on each of the lattices. A cylinder cut[90] was applied to all the data, to reduce the effects of rotational symmetry violation. The cylinder cut selects momenta with $\Delta p \leq \frac{2\pi}{n_i}$, where n_i is the spatial extent of the lattice, and $\Delta p = |p| \sin\theta_p$ is a measure of the distance from the hypercubic diagonal $\hat{n} = \frac{1}{2}(1,1,1,1)$. The angle θ_p is determined by $|p|\cos\theta = p \cdot \hat{n}$. The figures displaying the full results for each of the lattices are displayed at the end of this section. The renormalisation point ζ was chosen to be approximately $p = 3$ GeV for $Z_\zeta(p)$ and $q = 5.5$ GeV when shown against q.

While the quark masses obtained from the FLIC overlap and Wilson overlap calculations are similar, they are not matched. In particular, the lightest five FLIC overlap masses are significantly lighter than the corresponding Wilson overlap masses. In order to compare the two actions, we have selected the heaviest nine masses, and divided them into sets of three. A quadratic interpolation

	4.286	4.60	4.80	3.70	4.00	m^0
m_w	1.4	1.4	1.4	1.4	1.4	
n_{ape}	7	4	4	14	6	
n_{low}	20	15	5	30	30	
μ_1	0.00622	0.00400	0.00305	0.00622	0.00400	18
μ_2	0.01244	0.00800	0.00610	0.01244	0.00800	36
μ_3	0.01866	0.01200	0.00915	0.01866	0.01200	54
μ_4	0.02488	0.01600	0.01220	0.02488	0.01600	72
μ_5	0.03110	0.02000	0.01525	0.03110	0.02000	90
μ_6	0.03732	0.02400	0.01830	0.03732	0.02400	108
μ_7	0.04354	0.02800	0.02134	0.04354	0.02800	126
μ_8	0.04976	0.03200	0.02439	0.04976	0.03200	144
μ_9	0.06220	0.04000	0.03049	0.06220	0.04000	181
μ_{10}	0.07464	0.04800	0.03659	0.07464	0.04800	217
μ_{11}	0.09330	0.06000	0.04574	0.09330	0.06000	271
μ_{12}	0.12439	0.08000	0.06098	0.12439	0.08000	362
μ_{13}	0.15549	0.10000	0.07623	0.15549	0.10000	452
μ_{14}	0.18659	0.12000	0.09148	0.18659	0.12000	543
μ_{15}	0.21769	0.14000	0.10672	0.21769	0.14000	633

Table 7.2: FLIC overlap parameters. For each β, displayed is the regulator parameter m_{w}, the number of APE smearing sweeps used in the FLIC kernel, at a smearing fraction of 0.7, the number of low eigenmodes projected out, and the 15 lattice masses μ. The final column shows the bare quark mass in MeV, which is approximately the same for each lattice (see Eq. (7.12)).

of each set of three masses to a common running quark mass is then performed. The interpolations are performed to give common running quark mass values of 600, 300 and 200 MeV. The same interpolations are performed for both the mass function and the renormalisation function. Results are shown in Figure 7.1.

First we examine the mass function. Both actions show the same asymptotic behaviour (as expected) and agree in the high momentum regime. The dynamical generation of mass appears to start at a higher momenta for FLIC overlap than for Wilson overlap, and as such there is a slight difference in the intermediate momentum regime. While the qualitative behaviour is similar in the low momentum regime, the FLIC overlap visibly displays enhanced mass generation for low momenta compared to the Wilson overlap. The difference appears to be more pronounced for heavier masses. The source of this difference is likely to be different discretisation errors between the two actions.

The comparison of the renormalisation function results between the two actions leads to similar conclusions. The actions once again agree asymptotically, and as one moves to lower momentum they begin to differ. Interestingly, the level

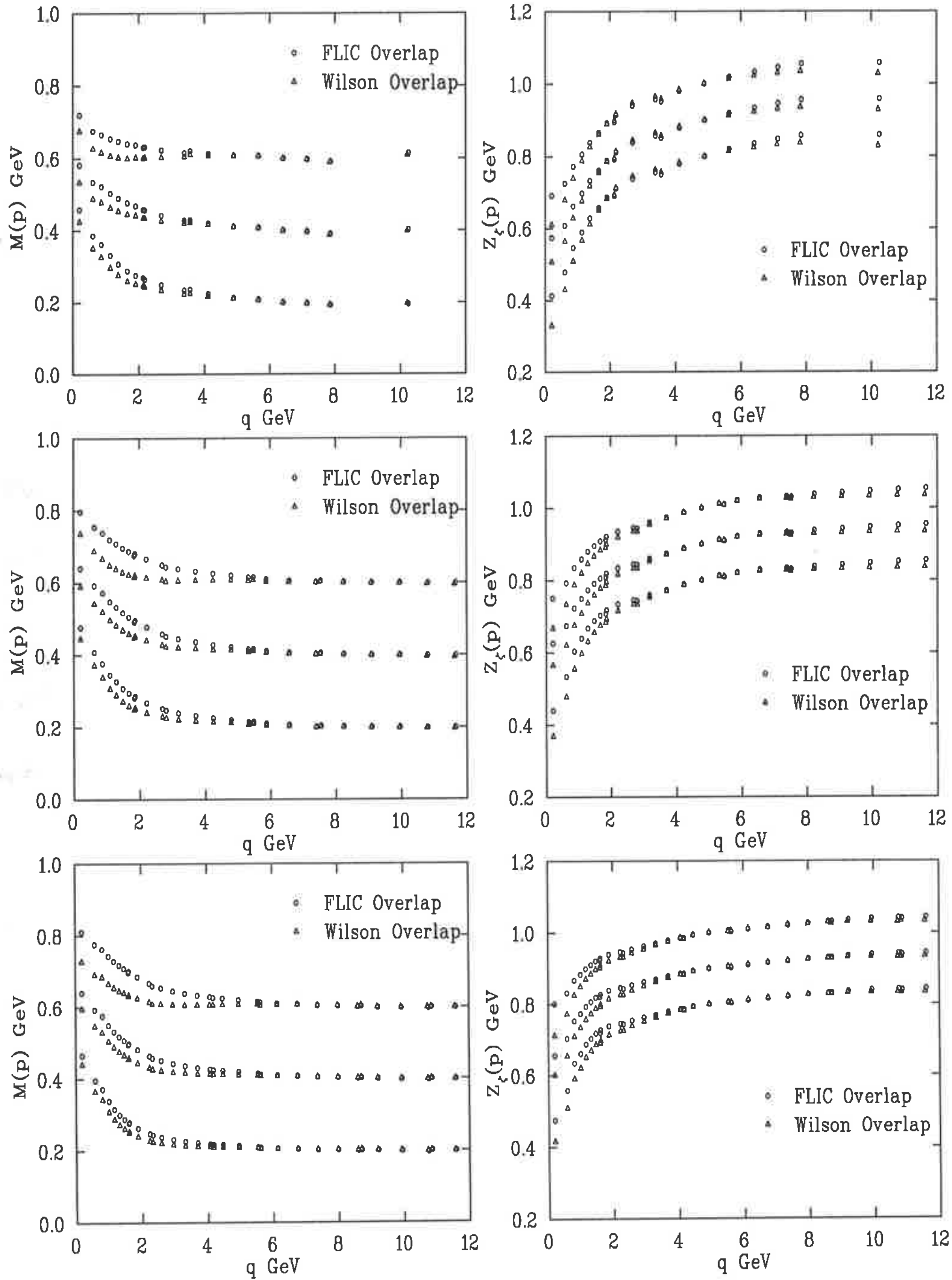

Figure 7.1: Cylinder cut data comparing the interpolated mass function $M(p)$ (left) and renormalisation function $Z_\zeta(p)$ (right), for the three quenched lattices $\beta = 4.286$ (top), $\beta = 4.60$ (middle), $\beta = 4.80$ (bottom). The different $Z_\zeta(p)$ have been offset vertically for clarity, the actual asymptotic values are the same.

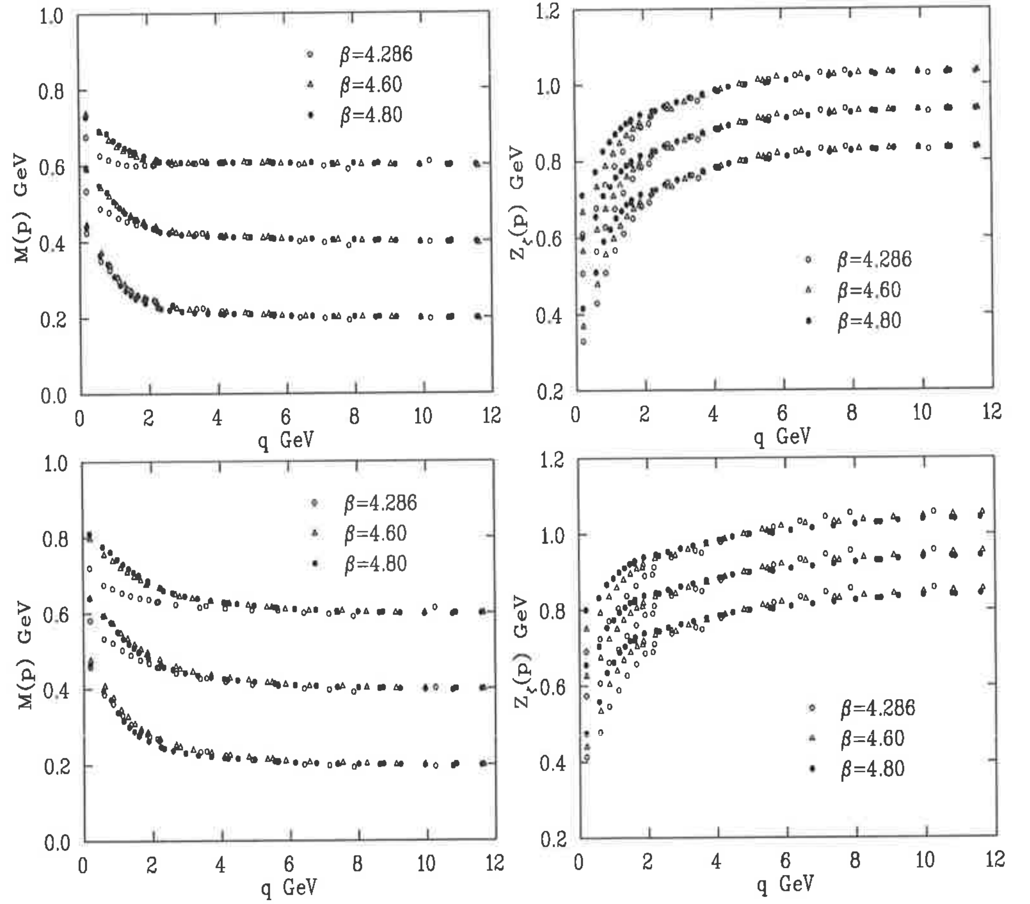

Figure 7.2: Cylinder cut data showing the scaling of the Wilson overlap (top row) and FLIC overlap (bottom row) interpolated mass function $M(p)$ and renormalisation function $Z_\zeta(p)$ for the three quenched lattices. The different $Z_\zeta(p)$ have been offset vertically for clarity, the actual asymptotic values are the same.

of disagreement in the low momentum regime for $Z_\zeta(p)$ seems to be consistent for each of the interpolated masses. What we can conclude from this study is that as expected, there is some difference in the physical properties of the FLIC overlap and Wilson overlap, primarily in the infrared, and more pronounced at larger masses. It is conceivable that as the Wilson overlap contains no smearing, its low momentum behaviour might retain some sensitivity to ultraviolet lattice artifacts.

Using the same interpolating technique, we can test the scaling of the mass function and renormalisation function. The results for both the Wilson overlap and FLIC overlap for the three quenched lattices, interpolated to common masses, are displayed in Fig. 7.2. We see good agreement for $M(p)$ at the two finer lattice spacings, suggesting that we are close to the continuum limit there, but the coarser lattice shows some scaling violations, more so at the heavier

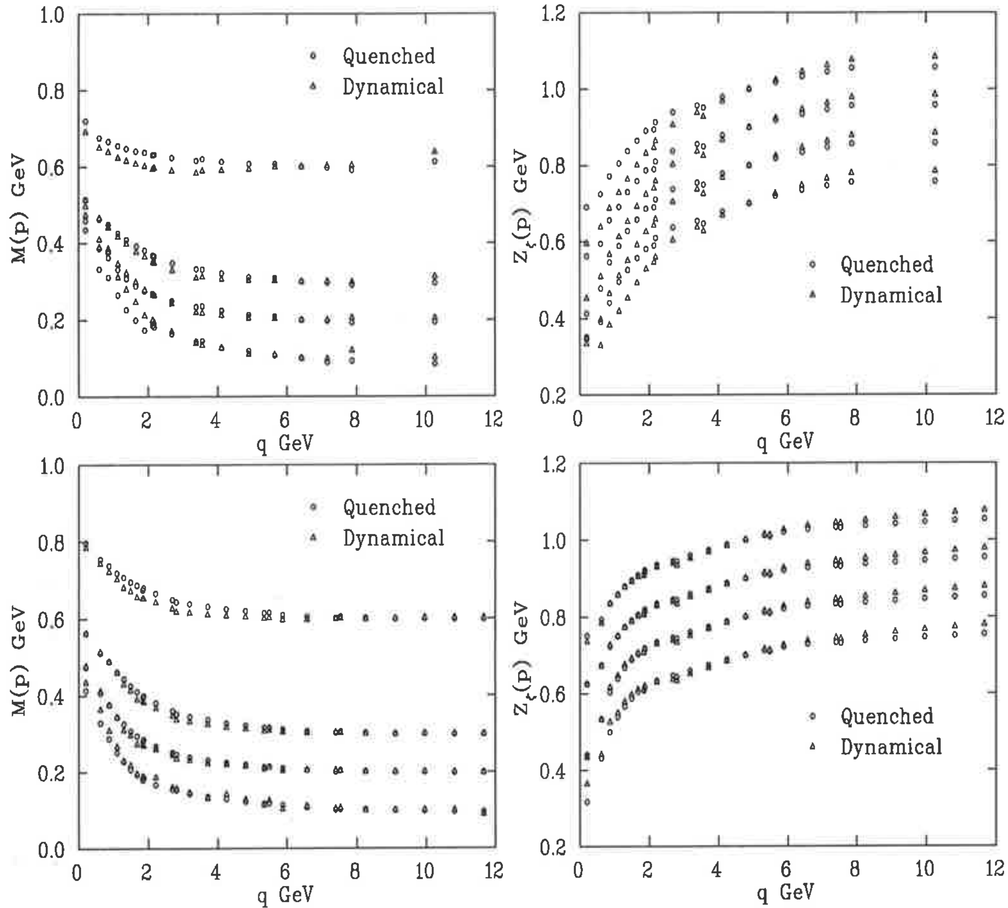

Figure 7.3: Cylinder cut data comparing the FLIC Overlap interpolated mass function $M(p)$ (left) and renormalisation function $Z_\zeta(p)$ (right), for the (approximately) matched quenched and dynamical lattices, $8^3 \times 16$ (top) and $12^3 \times 24$ (bottom). The different $Z_\zeta(p)$ have been offset vertically for clarity, the actual asymptotic values are the same.

quark masses. Once again this is likely to be due to the discretisation errors, as the shorter Compton wavelength of the heavy quarks can "fall through" the lattice spacing. The difference between the three lattices is more visible for $Z_\zeta(p)$, and the level of disagreement is consistent for the three different masses. So $Z_\zeta(p)$ seems to be further from the continuum limit than $M(p)$. The scaling properties of both actions are quite similar. Observing the behaviour on the coarse lattice, we can see that at the largest lattice spacing the dynamical mass generation is suppressed in the infrared, and that the dynamical mass generation is sensitive to the physics at the cutoff scale at the heavier masses. This then suggests that, as the mass generation of the Wilson overlap at moderate to heavy masses is suppressed compared to FLIC overlap, perhaps the coupling between the infrared and ultraviolet physics for the FLIC overlap is reduced.

The results comparing the quenched and partially quenched results on the two (approximately) matched lattices are displayed in Fig. 7.3. Four interpolated masses at 100, 200, 300 and 600 MeV were compared on the $8^3 \times 16$ and the $12^3 \times 24$ lattices. The partially quenched lattices have a sea quark mass of roughly the strange quark mass. We see that there is little difference between the quenched and dynamical results for both the mass function and the renormalisation function. The lightest mass on the $8^3 \times 16$ lattice shows some difference in the low momentum regime, but this is a lattice spacing artifact (the quenched results show some "bumpiness", see Fig. 7.10) as the finer lattice shows little difference. From other studies comparing the quenched and unquenched hadron spectrum at heavy quark masses, which show little difference, our result is somewhat unsurprising. We conclude that in order to see a significant difference between quenched and dynamical results, we need to go to lighter sea quark masses, and hence larger volumes.

Finally, we consider the possibility of extrapolating our results to the chiral limit. In Figure 7.4 we have plotted the dependence of $M(p)$ on the bare mass, at fixed momenta. The lowest ten momenta are shown for the $16^3 \times 32$ quenched lattice, and the $12^3 \times 24$ dynamical lattice. We see that at heavy quark masses the behaviour is linear, while at moderate to light quark masses there is an increasing amount of curvature present. This is consistent with the behaviour of the hadron spectrum predicted by chiral perturbation theory. However, chiral perturbation theory does not give us a prediction of QCD type quantities such as $M(p)$. Using a reliable formula for the extrapolation of $M(p)$ and $Z_\zeta(p)$ requires further study, as we must disentangle the effects of finite volume, quenched chiral logs and genuine curvature.

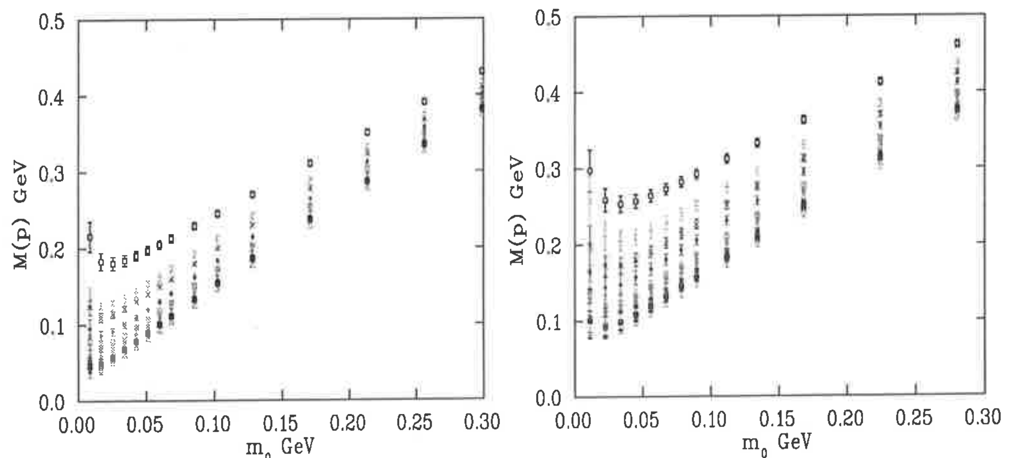

Figure 7.4: Cylinder cut data showing the dependence of the mass function $M(p)$ on the bare mass m^0 at fixed momenta, for the lowest 10 momenta values. Shown is the quenched $\beta = 4.80$ lattice (left) and the dynamical $\beta = 4.00$ lattice (right).

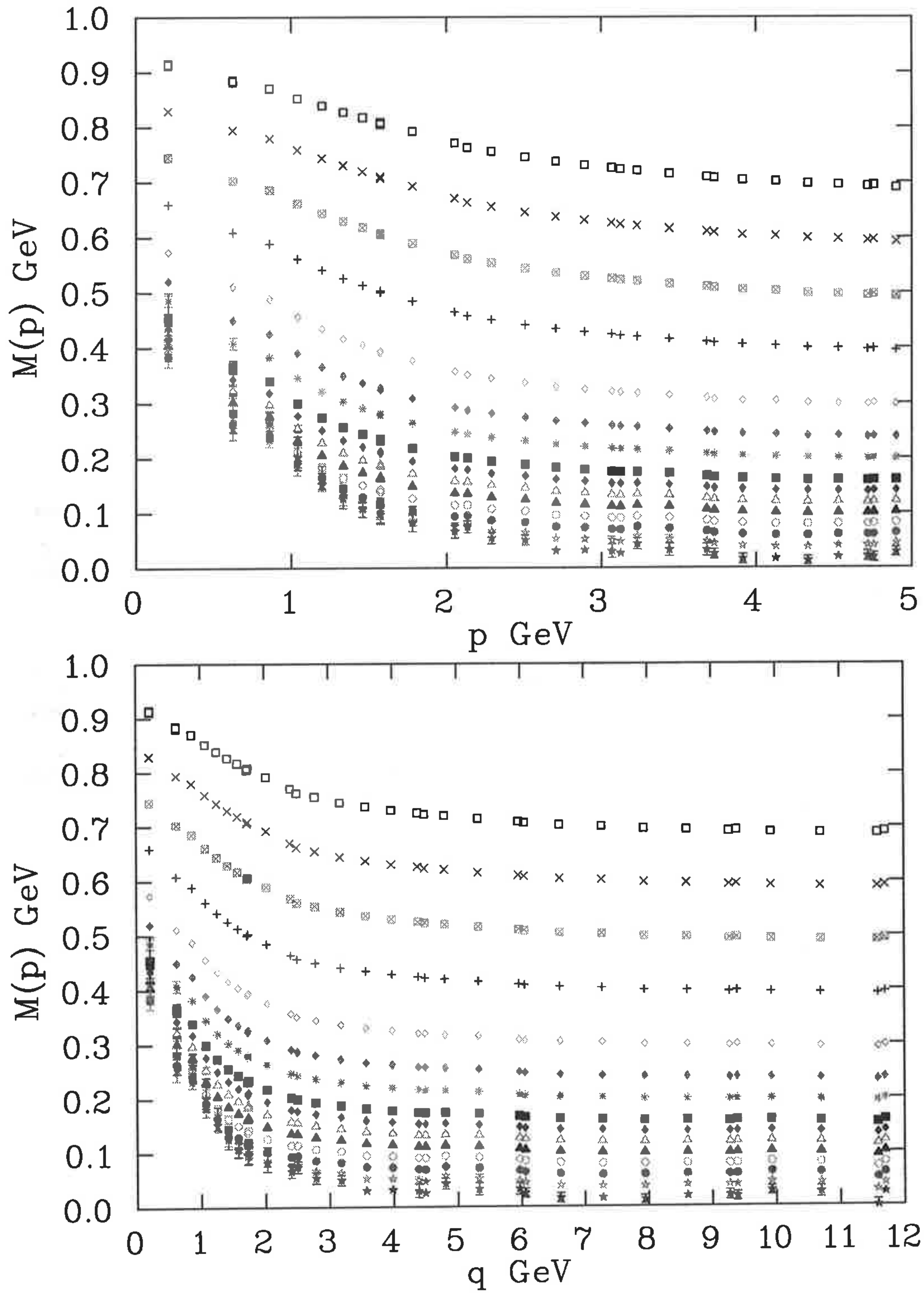

Figure 7.5: Cylinder cut data for the mass function $M(p)$, at finite quark mass for the quenched $16^3 \times 32$ lattice at $\beta = 4.8$. The top plot is against the discrete lattice momentum p, and the bottom plot is against the kinematical lattice momentum q.

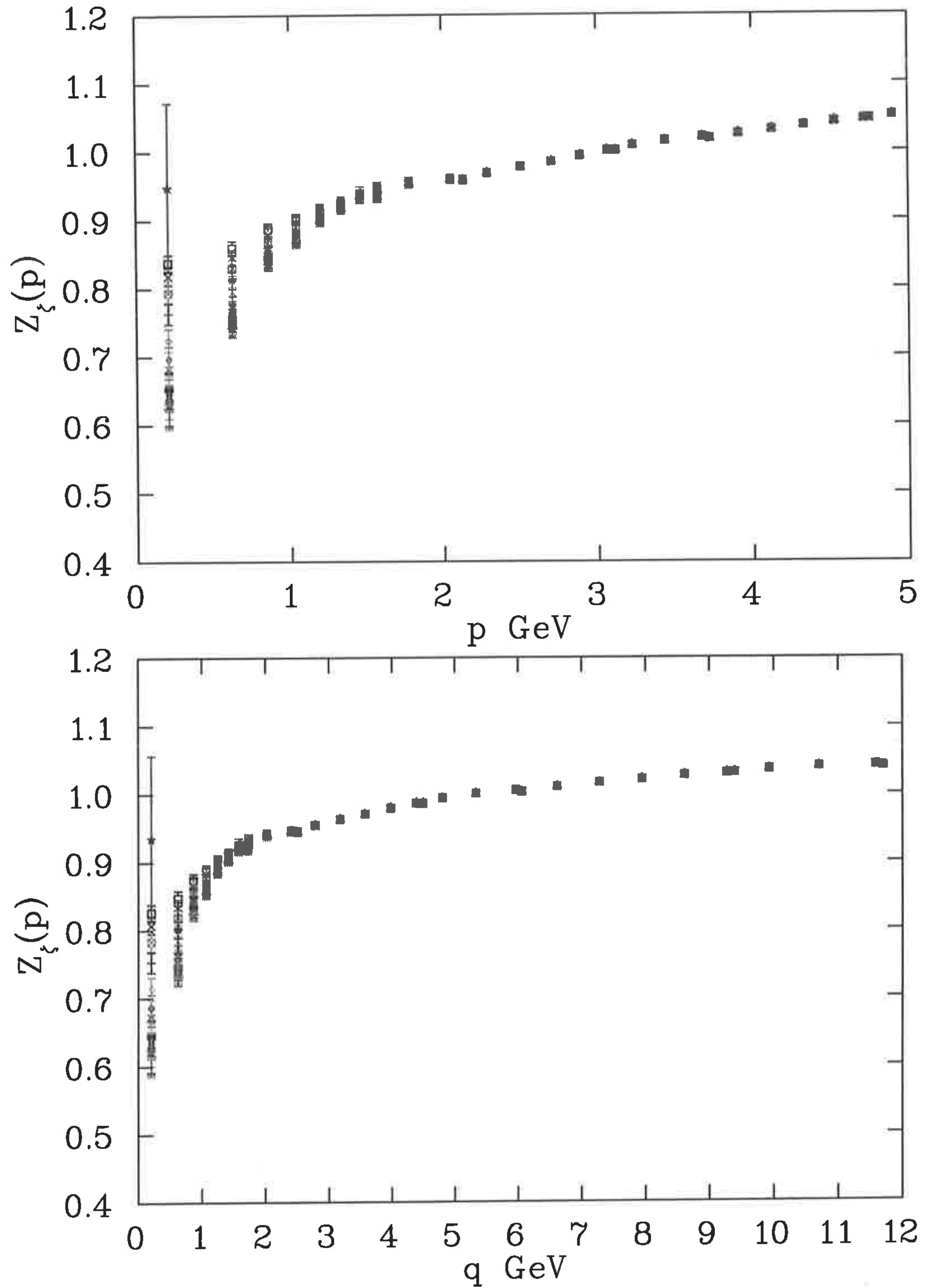

Figure 7.6: Cylinder cut data for the renormalization function $Z_\zeta(p)$ for the quenched $16^3 \times 32$ lattice at $\beta = 4.8$, with renormalization point $\zeta = 5.5$ GeV. The top plot is against the discrete lattice momentum p, and the bottom plot is against the kinematical lattice momentum q. The errant point is due to finite volume effects, and comes from the lightest mass.

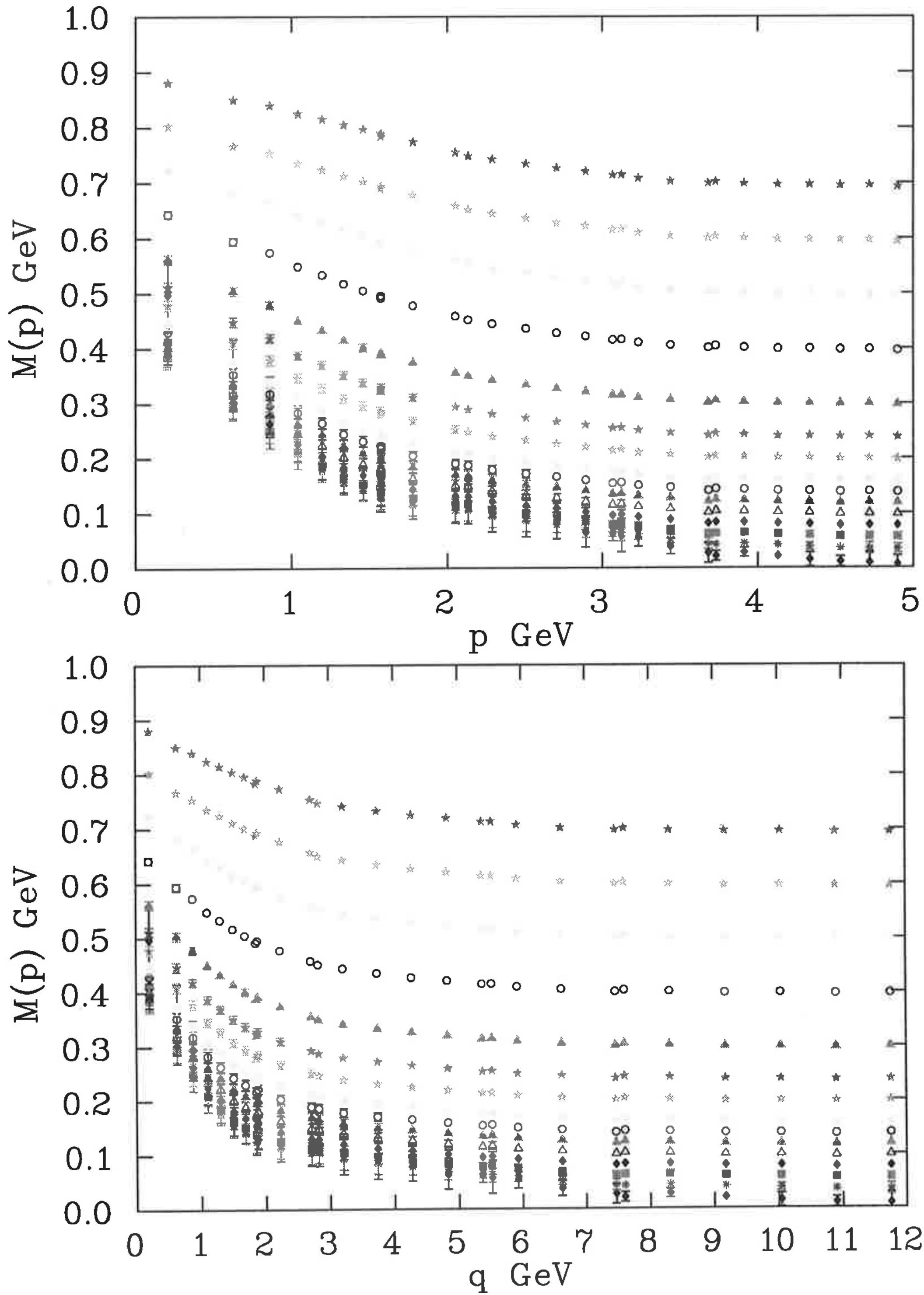

Figure 7.7: Cylinder cut data for the mass function $M(p)$, at finite quark mass for the quenched $12^3 \times 24$ lattice at $\beta = 4.6$. The top plot is against the discrete lattice momentum p, and the bottom plot is against the kinematical lattice momentum q.

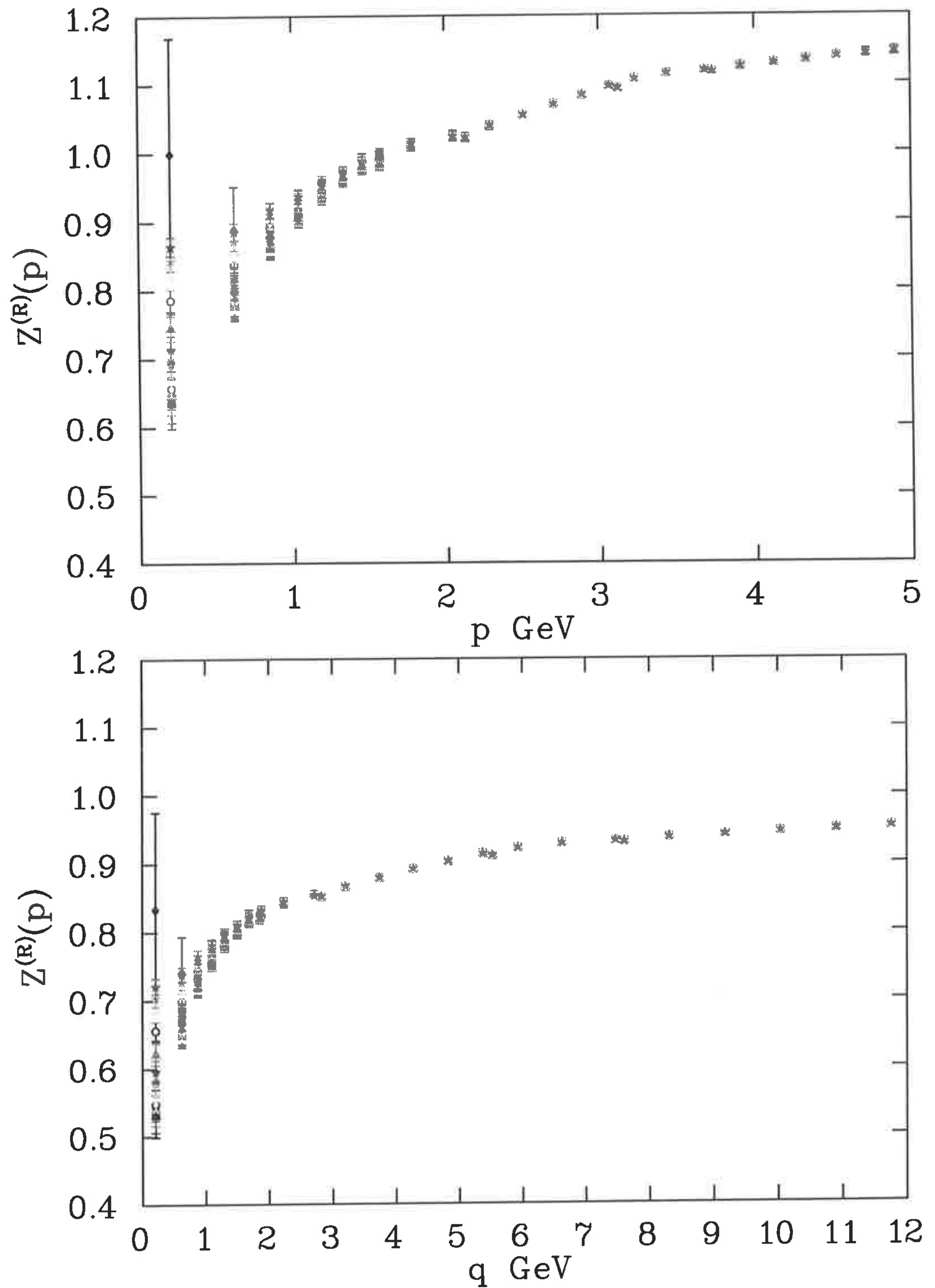

Figure 7.8: Cylinder cut data for the renormalization function $Z_\zeta(p)$ for the quenched $12^3 \times 24$ lattice at $\beta = 4.6$, with renormalization point $\zeta = 5.5$ GeV. The top plot is against the discrete lattice momentum p, and the bottom plot is against the kinematical lattice momentum q.The errant point is due to finite volume effects, and comes from the lightest mass.

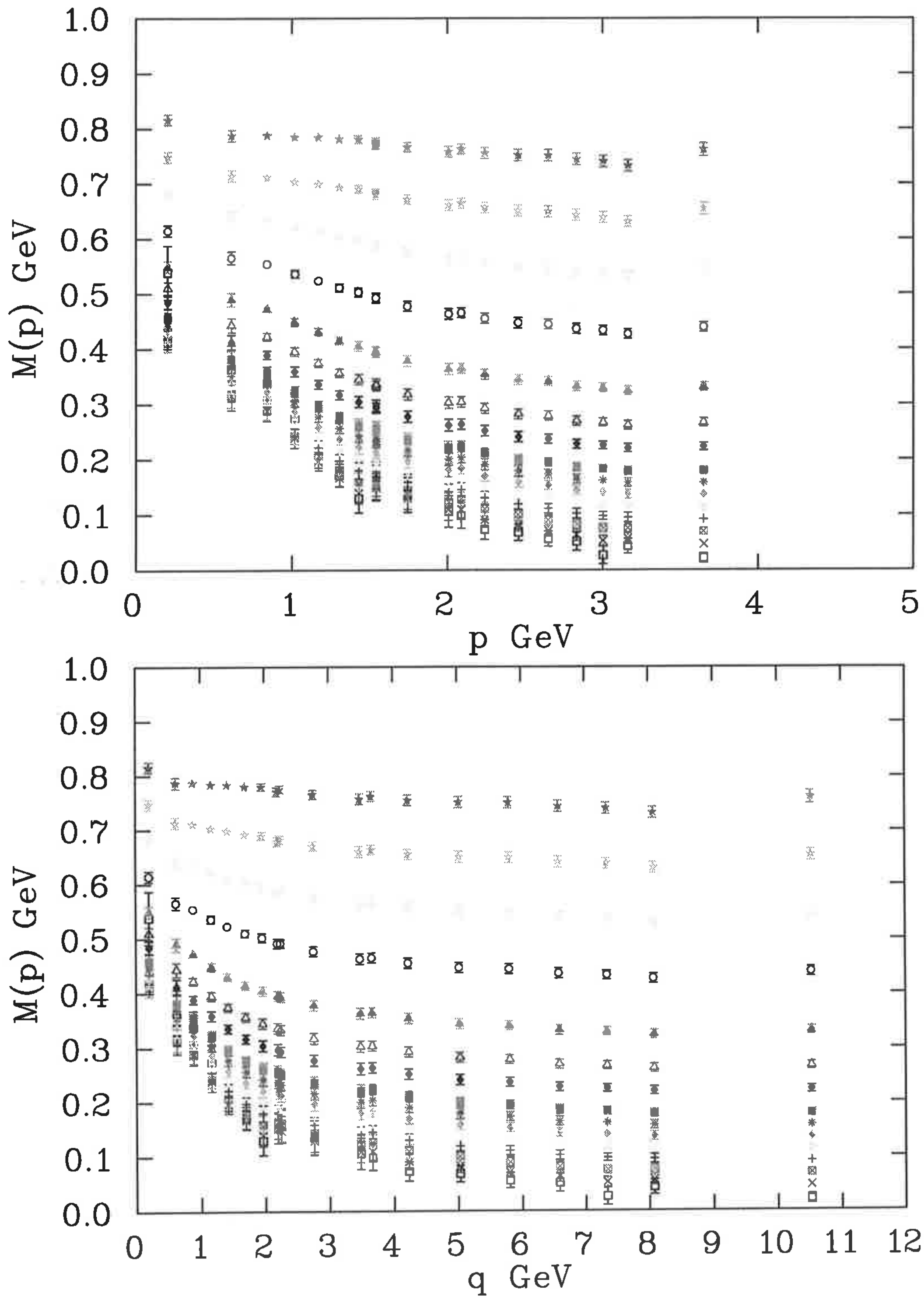

Figure 7.9: Cylinder cut data for the mass function $M(p)$, at finite quark mass for the quenched $8^3 \times 16$ lattice at $\beta = 4.286$. The top plot is against the discrete lattice momentum p, and the bottom plot is against the kinematical lattice momentum q.

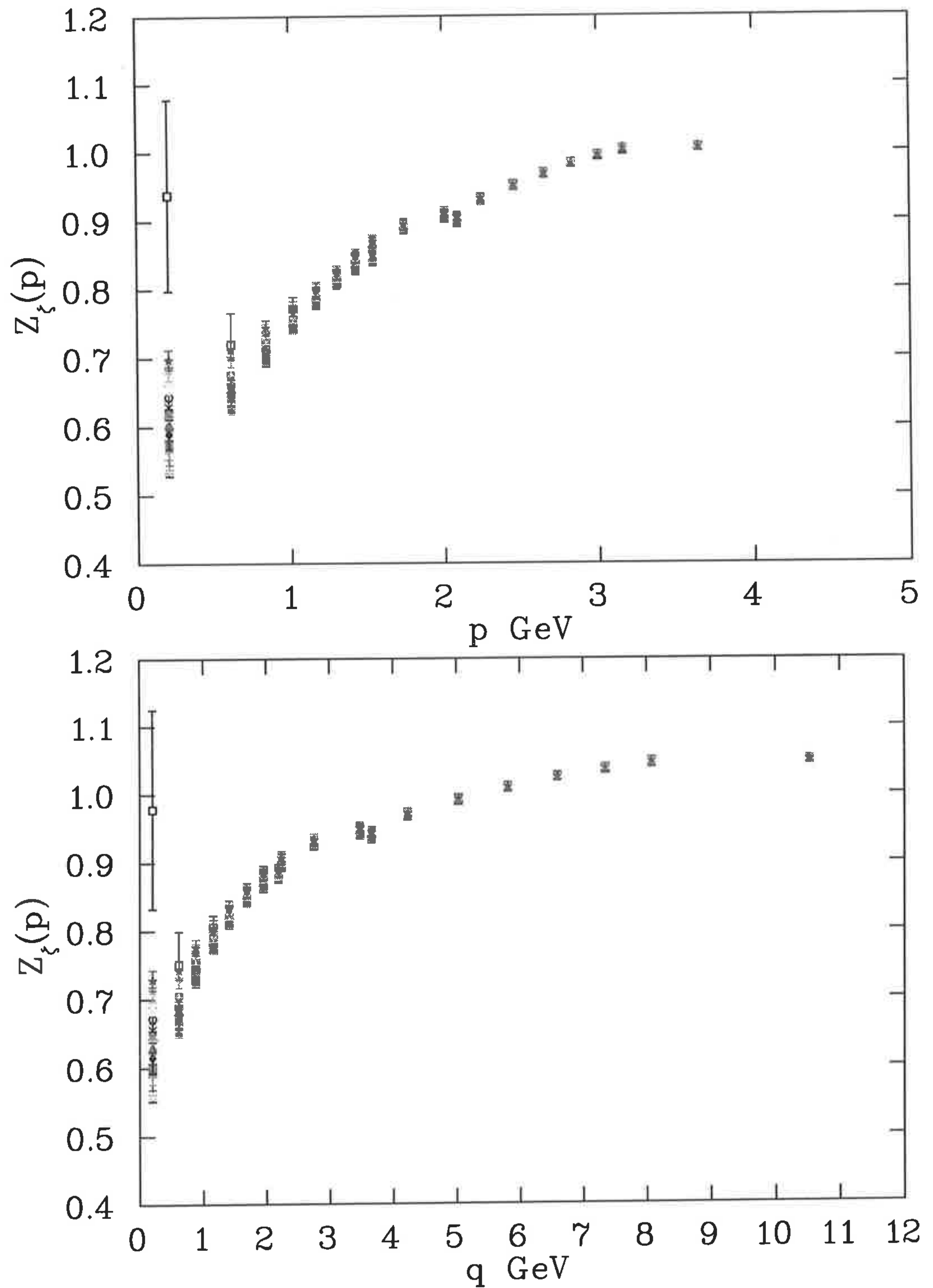

Figure 7.10: Cylinder cut data for the renormalization function $Z_\zeta(p)$ for the quenched $8^3 \times 16$ lattice at $\beta = 4.286$, with renormalization point $\zeta = 5.5$ GeV. The top plot is against the discrete lattice momentum p, and the bottom plot is against the kinematical lattice momentum q. The errant point is due to finite volume effects, and comes from the lightest mass.

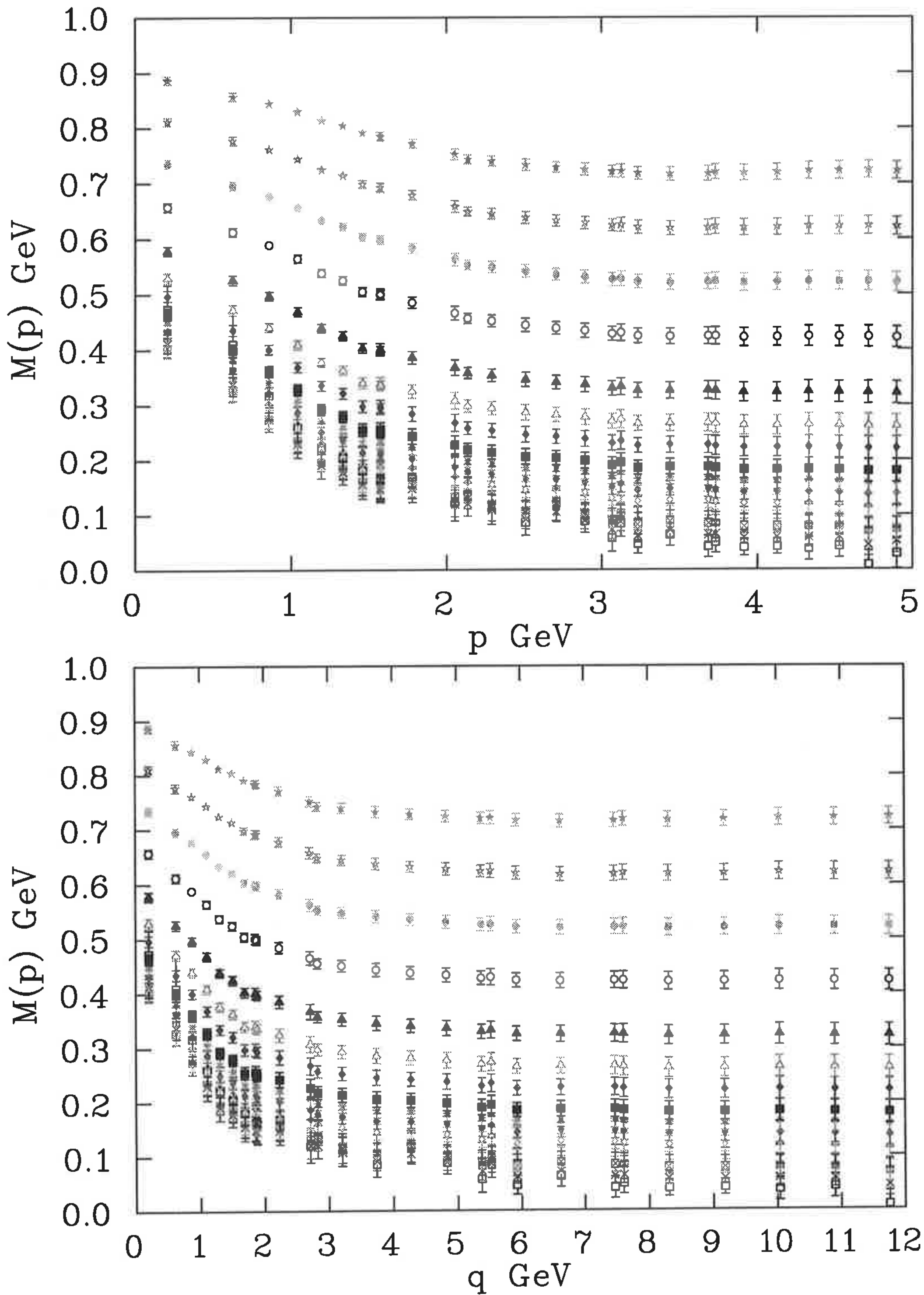

Figure 7.11: Cylinder cut data for the mass function $M(p)$, at finite quark mass for the partially quenched $12^3 \times 24$ lattice at $\beta = 4.0$. The top plot is against the discrete lattice momentum p, and the bottom plot is against the kinematical lattice momentum q.

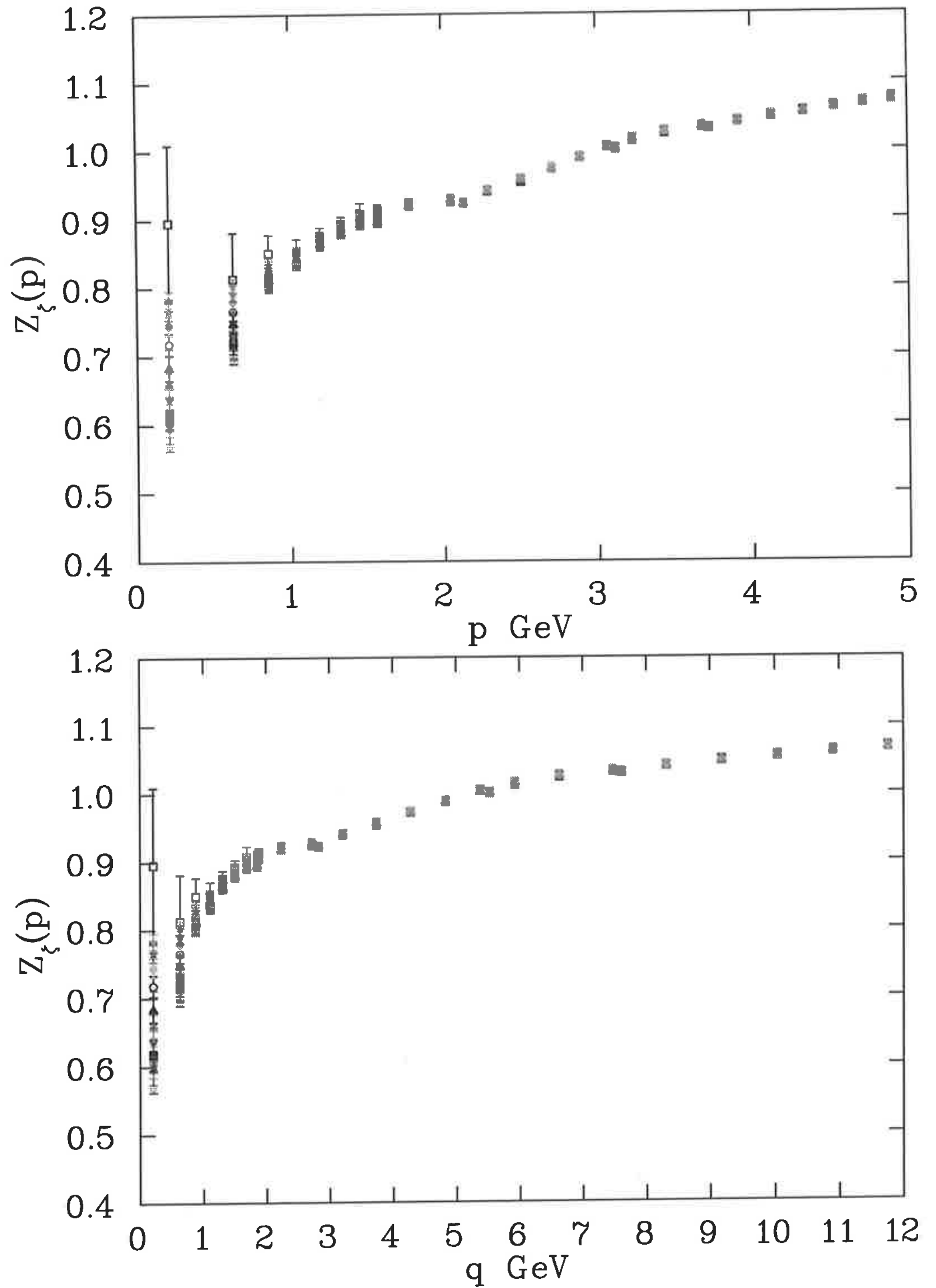

Figure 7.12: Cylinder cut data for the renormalization function $Z_\zeta(p)$ for the partially quenched $12^3 \times 24$ lattice at $\beta = 4.0$, with renormalization point $\zeta = 5.5$ GeV. The top plot is against the discrete lattice momentum p, and the bottom plot is against the kinematical lattice momentum q. The errant point is due to finite volume effects, and comes from the lightest mass.

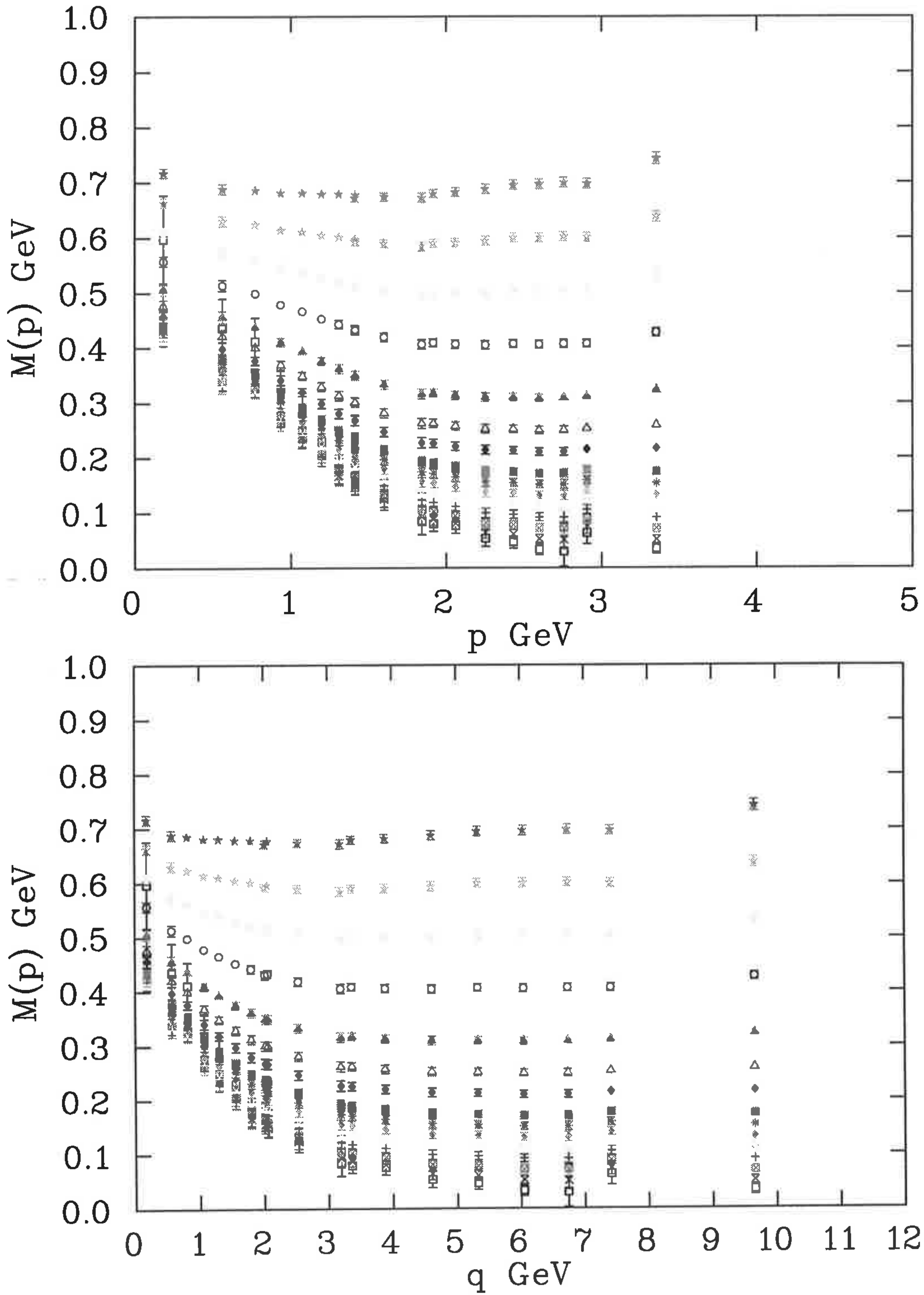

Figure 7.13: Cylinder cut data for the mass function $M(p)$, at finite quark mass for the partially quenched $8^3 \times 16$ lattice at $\beta = 3.7$. The top plot is against the discrete lattice momentum p, and the bottom plot is against the kinematical lattice momentum q.

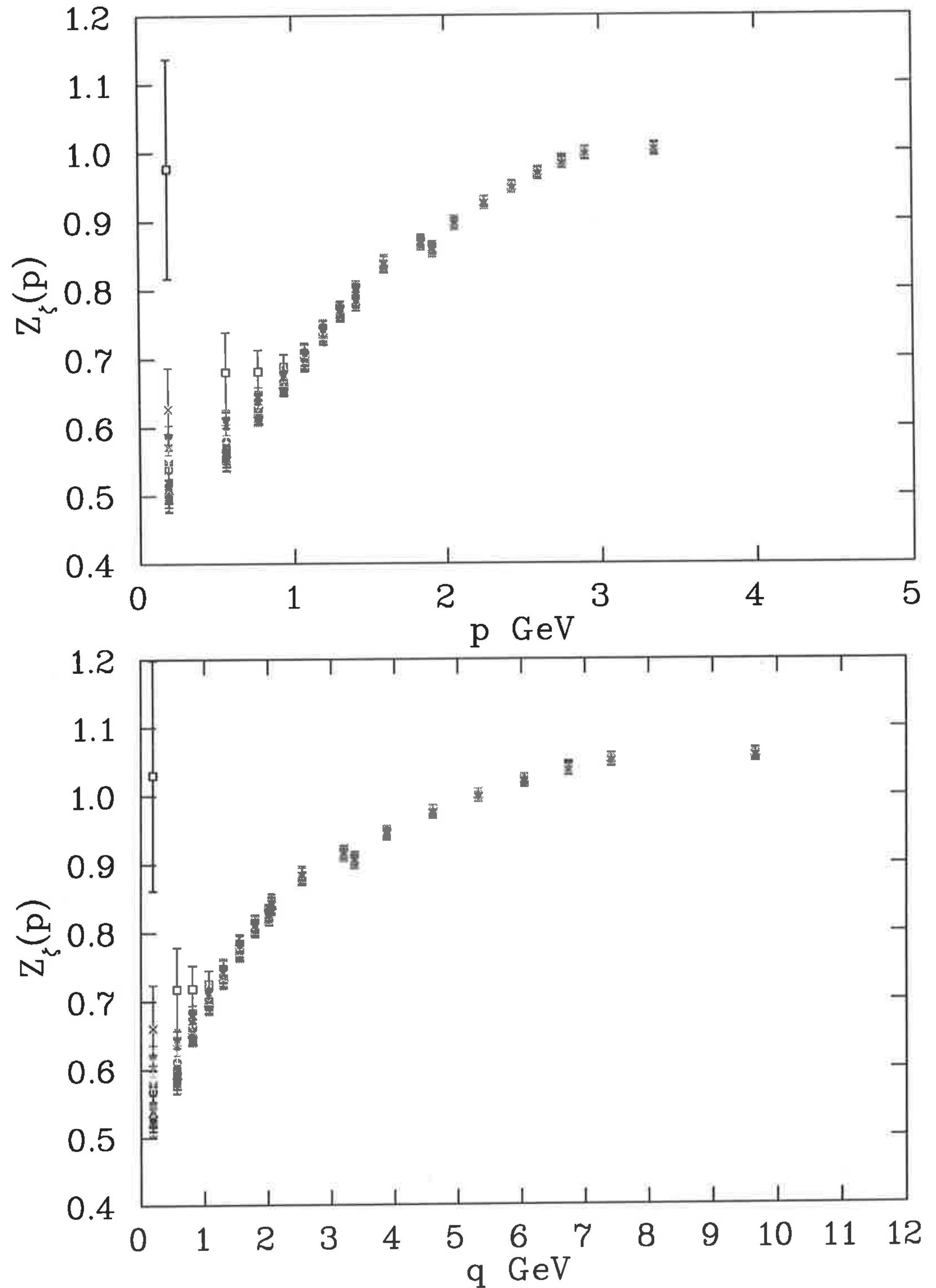

Figure 7.14: Cylinder cut data for the renormalization function $Z_\zeta(p)$ for the partially quenched $8^3 \times 16$ lattice at $\beta = 3.7$, with renormalization point $\zeta = 5.5$ GeV. The top plot is against the discrete lattice momentum p, and the bottom plot is against the kinematical lattice momentum q. The errant point is due to finite volume effects, and comes from the lightest mass.

7.2 Topology

Our last investigation is into topology, and the Atiyah-Singer index theorem. In the continuum, we have that the gluonic topological charge Q is equal to the difference between n_-, the number of left handed zero modes of $\not{D}$, and n_+, the number of right handed modes,

$$Q = n_- - n_+. \tag{7.29}$$

As we have seen, the overlap has robust zero modes in the topologically non trivial gauge backgrounds. A study of the index theorem via the Wilson overlap has been performed elsewhere[96]. We wish to conduct a similar study with the FLIC overlap and compare. We defined the fermionic topological charges

$$Q_{\text{flic}} = n_- - n_+, \qquad Q_{\text{w}} = n_- - n_+ \tag{7.30}$$

via the zero modes of the FLIC overlap and Wilson overlap respectively, as in the continuum index theorem. In this study the FLIC action uses a 3-loop improved clover term. The gluonic topological charge Q_{cool} is defined via the 5-loop $F_{\mu\nu}$,

$$Q_{\text{cool}} = \frac{1}{32\pi^2} \sum_x \epsilon_{\mu\nu\rho\lambda} F_{\mu\nu} F_{\rho\lambda}, \tag{7.31}$$

after performing 9 sweeps of 3-loop over improved cooling[15]. The number of sweeps was chosen as the minimum necessary so that the topological charge for each configuration examined was within 0.25 of an integer (although typically it was within 0.05 of an integer by this stage). For the sake of interest, we also defined an additional gluonic charge Q_{ape} which was calculated via a 3-loop improved field strength tensor on configurations that had been subjected to the same number of APE smearing sweeps that was used in the FLIC kernel for that lattice. The zero modes of the overlap were calculated using the accelerated conjugate gradient Ritz algorithm[67].

β	Volume	S_g	a fm	n_{ape}	m_w(FLIC)	m_w(Wilson)
4.60	$12^3 \times 24$	IMP	0.125	4	1.37	1.37
4.38	$8^3 \times 16$	IMP	0.165	6	1.37	1.37
11.783	$12^3 \times 24$	DBW2	0.125	3	1.24	–
10.413	$8^3 \times 16$	DBW2	0.165	3	1.29	–
9.836	$8^3 \times 16$	DBW2	0.195	4	1.29	–

Table 7.3: Lattice specifications and simulation parameters for the five quenched lattice studies. Shown is the coupling β, lattice volume, gluonic action (either Luscher Weisz(IMP) or DBW2), lattice spacing, number of smearing sweeps used for FLIC and for Q_{ape}, and the FLIC and Wilson overlap regulator mass.

Cfg.	$Q_{\rm cool}$	$Q_{\rm ape}$	$Q_{\rm flic}$	$Q_{\rm w}$	$Q_{\rm cool}$	$Q_{\rm ape}$	$Q_{\rm flic}$	$Q_{\rm w}$
1	−1	−2	−1	−1	+1	+1	+1	0
2	+1	0	+1	+1	+3	+4	+3	+3
3	−4	−4	−4	−4	0	−1	−1	+1
4	0	0	0	0	0	0	0	0
5	+1	+2	+1	+1	−1	−1	0	0
6	−2	−2	−2	−2	0	−1	−1	−1
7	−2	−2	−2	−2	+6	+4	+5	+5
8	−1	−1	−1	−1	−2	−2	−2	−1
9	0	0	0	0	+1	+1	+1	+1
10	−2	−1	−2	−2	0	−1	0	0
11	+1	+1	+1	+1	−3	−3	−4	−2
12	0	0	0	0	−1	0	−1	−1
13	−3	−2	−3	−3	+2	+3	+3	+2
14	−3	−3	−4	−4	−4	−3	−4	+3
15	−3	−2	−3	−2	−3	−3	−3	−3
16	−2	−1	−1	−1	0	−1	−1	0
17	−5	−5	−5	−5	0	−1	−2	0
18	−3	−2	−3	−2	−1	−1	−1	−1
19	−5	−5	−5	−5	+1	+1	+1	+1
20	−4	−4	−5	−4	−1	0	0	+1
ΔQ	–	6	3	4	–	12	9	14

Table 7.4: Topological charge for the Luscher Weisz lattices at $\beta = 4.60, a = 0.125$ fm (left), and $\beta = 4.38, a = 0.165$ fm (right). Shown is the various topological charges for each configuration, and the bottom row shows ΔQ.

We consider five different quenched lattices, with parameters as given in Table 7.3. First, we examine the two lattices used in Ref. [96]. The results are shown in Table 7.4. To compare the different charges, we choose $Q_{\rm cool}$ as a baseline and sum the difference between it and the other three charges $Q_{\rm ape}, Q_{\rm flic}, Q_{\rm w}$ across the twenty configurations studied for each lattice,

$$\Delta Q = \sum_{i=1}^{20} |Q^{(i)} - Q^{(i)}_{\rm cool}|. \tag{7.32}$$

This gave us a measure ΔQ of the amount by which the different topological charges disagreed, and hence a measure of the amount by which the index theorem is violated. As expected, on the finer lattice we observe that the level of disagreement between the different topological charges is smaller than on the coarser lattice. $Q_{\rm ape}$ does not appear to be a particularly robust definition of topology, disagreeing significantly with $Q_{\rm cool}$ and $Q_{\rm flic}$. As seen by the lower ΔQ values, the index theorem is satisfied more often for the FLIC Overlap than the Wilson overlap.

Cfg.	Q_{cool}	Q_{ape}	Q_{flic}	Q_{cool}	Q_{ape}	Q_{flic}	Q_{cool}	Q_{ape}	Q_{flic}
1	+1	+1	+1	+2	+2	+2	−1	−1	−1
2	−1	−1	−1	−1	−1	−1	+8	+7	+8
3	+1	+1	+1	−1	−1	−1	−2	−2	−1
4	0	0	0	+4	+4	+4	0	0	−1
5	+1	+1	+1	0	0	0	−2	−1	−1
6	0	0	0	0	0	0	0	0	0
7	−2	−2	−2	+2	+1	+2	−2	−2	−2
8	+2	+2	+2	+4	+3	+4	−7	−7	−8
9	+2	+2	+2	0	0	0	+1	+1	+1
10	−2	−2	−2	−4	+4	−4	+3	+3	+4
11	−2	−2	−2	+2	+2	+2	+1	+1	+1
12	+3	+3	+3	−2	−2	−2	−2	−2	−2
13	+3	+3	+3	−1	0	0	−5	−5	−5
14	+3	+3	+3	−1	−1	−1	−2	−1	−2
15	−3	−3	−3	−2	−2	−3	+1	+1	+1
16	−2	−2	−2	−4	−3	−4	+2	+2	+2
17	+1	+1	+1	−1	−1	−1	+2	+2	+2
18	+1	+1	+1	0	0	0	−3	−3	−4
19	−1	−1	−1	0	0	0	+1	+1	+1
20	+1	+1	+1	−1	−1	−1	+2	+2	+2
ΔQ	–	0	0	–	4	1	–	2	6

Table 7.5: Topological charge for the DBW2 lattices at $\beta = 11.783, a = 0.125$ fm (left), $\beta = 10.413, a = 0.165$ fm (middle), and $\beta = 9.836, a = 0.195$ fm (right). Shown is the various topological charges for each configuration, and the bottom row shows ΔQ.

The results for the DBW2 lattices are shown in Table 7.5. Remarkably, the index theorem is perfectly satisfied on the fine lattice, and all three different topological charges studied are in agreement. Even at $a = 0.165$ fm we have $\Delta Q_{\text{flic}} = 1$, so we can see that at a fixed lattice spacing the DBW2 gluonic action seems to give better topological results than for Luscher-Weisz glue. The results for the coarsest DBW2 lattice at $a = 0.195$ fm are still quite impressive when compared to the perturbatively improved lattice at $\beta = 4.38$ with $a = 0.165$ fm.

We should remark that in order to get an integer gluonic topological charge, some smoothing must be done so any definition of Q we make will be somewhat artificial. The reason for this is that lattice gauge fields at typical couplings are much rougher than continuum ones. Studies of the localisation of the low lying eigenmodes of the Hermitian Wilson[65] and the FLIC[44] operators indicates that the fermionic actions can "see" through the ultraviolet noise that hides the topology on equilibrium gauge field configurations. The low modes which cross zero are connected to the exact zero modes of the corresponding overlap Dirac

operator[46].

The presence of topological lattice artifacts can cause crossings to occur across the valid range of the overlap regulator mass m_{w}. This introduces an ambiguity in to the definition of topology via the overlap, as Q_{flic} and Q_{w} count the number of crossing between the critical value m_{c} and m_{w}, and hence become m_{w} dependent. In particular, lattice deformed topological objects can cause "double crossings" where the corresponding low-lying mode crosses zero at two different places. These lattice artifacts would be one potential source of violations of the index theorem. We have examined the configurations for which the index theorem is not satisfied, and confirmed that on many of them there are double crossings (see Fig. 7.15). For sufficiently smooth gauge fields, all topological objects would be well formed and their crossings would occur near m_{c}, and the fermion topological charge would be robust. On such smooth configurations, it is highly likely that the index theorem would be perfectly satisfied.

We conclude with some remarks on the dynamical sector. For fixed a, dynamical configurations are much rougher than quenched configurations, so it is likely that the index theorem will be (statistically) less valid. In fact, when attempting to use the FLIC Overlap on dynamical $12^3 \times 24$ lattices at $a > 0.135$, we seem to encounter the Aoki phase[97, 98, 99] where the physical properties of the overlap change, and it is no longer local. The Aoki phase is characterised by the near zero modes of the overlap kernel no longer being isolated, but rather the dense spectrum has no significant separation from zero. We have observed the typical eigenvalues for the dense spectrum of H_{flic} being an order of magnitude smaller in the Aoki phase than in the usual phase. At larger volumes, it is possible that the Aoki phase may set in at smaller a. This means that dynamical overlap simulations on larger volumes are likely to only lie in the valid phase for $a \approx 0.1 - 0.125$ fm.

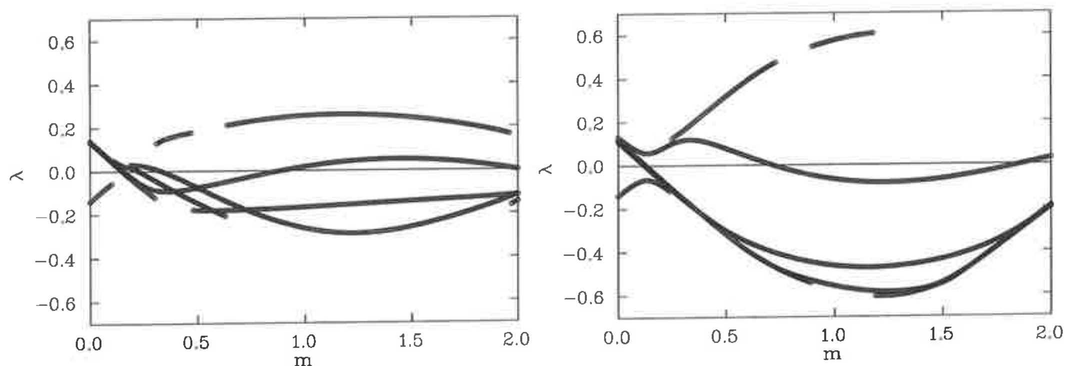

Figure 7.15: The spectral flow of $H^{3L}_{\mathrm{flic}}(-m)$, on two configurations for which the index theorem is not satisfied. The left graph is of configuration 14 for the $\beta = 4.60$ lattice, and the right graph is of configuration 15 for the $\beta = 10.413$ lattice. The smallest 4 eigenvalues are shown. Levels crossing zero more than once within the doubler free region can be clearly identified.

Chapter 8

Conclusion

Lattice Quantum Chromodynamics is the only known way to study non-perturbative QCD directly from first principles. A suitable formulation of chiral symmetry within lattice QCD has been a long sought after goal, which was realised with the advent of overlap fermions. Overlap fermions are a lattice discretisation of the Dirac operator which essentially has all the desired behaviour of the continuum Dirac operator. Overlap fermions remove the light quark computational bottleneck, replacing it with a new one. The new bottleneck is in the evaluation of the matrix sign function acting on the overlap kernel.

Practical implementations of the overlap-Dirac operator use a sum over poles to approximate the matrix sign function. These approximations are evaluated using an iterative conjugate gradient routine. As each iteration requires about twice as much computational effort to evaluate as a single evaluation of the Hermitian Wilson-Dirac operator H_{w}, reducing the number of iterations needed is the most direct way of reducing the expense of the overlap formalism. To succeed in this, we select an overlap kernel with an improved condition number motivated by analytic arguments. From the six candidate actions tested, the FLIC action has the best convergence properties, requiring less low-lying projections than the Wilson action and providing a saving in iterations by about a factor of 2. This saving in iterations translates almost directly into a saving in computation time. The FLIC action is a clover-improved fermion action where the irrelevant operators incorporate APE smeared links.

As the FLIC action has only nearest neighbour couplings, it is well suited to calculations on highly parallel machines. Simple identities in spinor space that are well-known for the Wilson fermion action can be applied easily to the FLIC action. This gives a saving of a factor of two in the computational effort required to evaluate the action of the FLIC operator on a vector. While there may be some implementation dependence in our compute-time results, we believe that this dependence will be sufficiently small that all groups who wish to perform overlap calculations will benefit in moving from the Wilson to the FLIC kernel.

In order to perform dynamical HMC simulations with FLIC fermions, or any

fat-link fermion action, it is necessary to use a smearing function which can be differentiated with respect to the gauge field. The use of reuniterisation in the APE smearing process is the only place where this may not hold. By a suitable choice of projection method, we can maintain the desired differentiability. Unit circle projection can be written in closed form and represented by a rational polynomial approximation. The matrix rational polynomial is differentiable with respect to the gauge field and thus enables one to derive the equations of motions necessary for the use of HMC with FLIC fermions. This provides a significant advantage over the MaxReTr method, where the absence of a differentiable form precludes the use of HMC and hence the availability of a simulation algorithm that scales linearly with the lattice volume V, although there are $O(V^2)$ alternatives [75]. This means that we now have an $O(V)$ algorithm available for the dynamical simulation of FLIC fermions. Furthermore, the method is general and can be applied to any fermion action with reuniterisation, including overlap fermions with a fat link kernel [76, 77, 78, 79], or other types of fat link actions [80] that may involve alternative smearing techniques [81].

We have calculated the quark propagator in momentum space using the FLIC overlap fermion actions in both quenched and partially quenched QCD. A comparison with a previous study in quenched QCD using the Wilson overlap[90] indicates that both the mass function $M(p)$ and the renormalisation function $Z_\zeta(p)$ agree well in the high momentum regime. The comparison was performed using an interpolation technique to match the running quark masses. The dynamical generation of mass appears at larger momenta for the FLIC overlap, with some disagreement beginning for $M(p)$ in the intermediate momentum regime, and a more pronounced difference in the infrared. The fact that this difference is larger at heavier quark masses suggests that it is due to the different $O(a^2)$ discretisation errors of the two actions. The difference was seen for the three different quenched lattice studies. The renormalisation function $Z_\zeta(p)$ displays similar differences, although the difference is less dependent on the running quark mass. From the study we find some indication that the coupling of ultraviolet and infrared physics for the FLIC overlap is reduced compared to the Wilson overlap.

The scaling of the Wilson overlap and FLIC overlap results were also studied. The results for $M(p)$ showed good scaling across the two finer lattices at $a = 0.1$ and $a = 0.125$. The results at $a = 0.195$ showed some difference between the two finer lattices. The scaling was poorer at heavier quarks, which is expected due to the increased sensitivity of heavy quarks to the lattice spacing, due to their shorter Compton wavelength. The renormalisation function $Z_\zeta(p)$ showed slightly poorer scaling than the mass function. A comparison between the two coarser quenched lattices and two approximately matched partially quenched lattices show that at the sea quark mass of around 150 MeV and the moderate to heavy valence quark masses studied the effects of fermionic vacuum fluctuations

are extremely small. This confirms the view that in order to see the effects of dynamical fermions we must go to lighter quark masses.

Topology and the Atiyah-Singer index theorem on the lattice was studied on quenched lattices. The results using perturbatively improved Luscher-Weisz glue showed that the fermionic topological charge defined by the FLIC Overlap was in better agreement with the gluonic topological charge than for the Wilson overlap. The results for the non-perturbatively improved DBW2 glue show that there is a reduced level of topological lattice artifacts and hence the index theorem is better satisfied. Exploratory studies indicate that the Aoki phase in dynamical QCD (for the FLIC overlap at least) occurs at much smaller lattice spacings than in quenched QCD. At moderate to large lattice volumes dynamical FLIC overlap simulations will likely only lie in the usual phase for $a \lesssim 0.125$ fm.

We have shown that FLIC overlap fermions are computationally faster than standard Wilson overlap fermions. The equations of motion for creating dynamical FLIC gauge fields have been presented, and may be easily extended to generate dynamical FLIC overlap configurations, although at current compute power only small to moderate lattice sizes would be feasible. With the ability to construct chiral fermions on the lattice, the chiral regime of quenched QCD is well within reach. Furthermore, we have gained the ability to explore physics that is sensitive to chiral symmetry. However, our studies indicate that exploring the chiral regime of dynamical QCD is still some way into the future, and at the least requires a sigtnificant increase in compute power. Any algorithmic developments to reduce the computational cost would bring this date forward, and further increase the region of QCD that can be explored from first principles on the lattice.

Appendix A

Implementation

The computational power required to perform dynamical lattice QCD simulations at the physical quark mass is still some way away. Any algorithmic improvements to reduce the cost of lattice simulations allow us to push the current boundary of the field. However, any algorithm is only as good as its implementation. A well thought out implementation can achieve sustained rates many times that of a naive one. When designing an implementation, one must keep in mind both the algorithm and the architecture of the target hardware.

The typical architecture of the computers which perform lattice simulations have changed markedly since the inception of the field. Not so long ago, genuine massively parallel supercomputers were leading the field of high-performance computing. The natural geometric parallelism of the lattice was highly suited to these architectures, characterised by a low cost of communication between processing elements compared to the cost of computation. Achieving good sustained floating point performance on such an architecture is relatively easy, as the sub-lattice on each processing element can be made small.

By contrast, computing clusters are now the mainstream solution for high-performance computing. The reason for this is rather pragmatic. The cost per Mflop (million floating point operations per second) of a cluster constructed from mass produced components is comparable or better than that of a proprietary supercomputer. However, while the cost of high speed internode networking may constitute a significant portion of the cost of a cluster, the time scale of even the high end interconnect hardware is orders of magnitude larger than typical compute scales. Compute clusters are characterised by a large cost of communication between processing elements, compared to computation costs. This requires a shift in paradigm, where distributing the problem over a large number of processors may invoke unacceptable high communication overhead.

The game then becomes to use a large sub-lattice per processor (preferably without exceeding cache), and minimise the communication cost where ever possible. For fields which are updated infrequently, the use of shadowing can significantly reduce communication. For each shadowed dimension of the sub-

lattice, the processor stores (one or more) of the neighbouring slices. Then communication only needs to be performed when the field is updated. This is useful for the gauge field. However, the main cost of lattice simulations is in evaluating the action of the Dirac operator on the fermion field, which requires communication for each evaluation, and therefore shadowing the fermion field offers no advantage.

Once the optimal patterns of communication have been established, the task then is to write scalar code which achieves a high sustained performance. The development of good lattice QCD code involves significant time and effort. Therefore, it is highly desirable to have portable code that performs well on more than one computer architecture. While the use of assembly code can give high sustained computational performance, it is by nature not portable. Modern (scalar code) compilers can also give good sustained performance, if fed well designed code. Typically, C and Fortran compilers are available on most clusters. However, C is not well suited towards medium to large scale software development, while Fortran 95 is a modern language that has a good lexical structure and is naturally oriented toward parallelism. Therefore, Fortran 95 is a natural choice for (scalar) lattice code development.

High Performance Fortran (HPF) can give parallelism to scalar code with little effort, but the paradigm is oriented towards the overall program functionality, rather than a per processor view. This means that the programmer loses a large portion of control over the communication patterns, and other optimisation techniques. MPI (Message Passing Interface) is a portable binding to communication hardware which allows for tight control of communication patterns. Therefore, we have chosen to implement our lattice code in Fortran 95 + MPI. The paradigm is modern, with information hiding to an extent, and a focus on taking advantage of data locality. Fundamental "low level" subroutines are optimised, with the program flow emphasised in the "high level" routines. Readability, reusability and rapidity are the three "R's" of modern programming.

The code is presented here. For the sake of space, loops which are manually unrolled in the actual code are given in "rolled up" index form here (and marked with a comment). Also, some higher level communication routines which are repeated for different rank arrays have only been shown in skeleton format, with comments to indicate the purpose of the routine. Apart from these two concessions to brevity, the code is complete. On the three different architectures available to the author (SUN, Alpha, x86) the code has been tested and key computations such as matrix vector multiplications have shown excellent sustained floating point rates across the board, typically falling between 50% and 80% of the theoretical per processor peak (in the absence of communication). Even with overlapping communication and computation however, current network hardware is not fast enough to hide the cost of communication, and the sustained rate for multi-processor code, including communication, lies between

25% and 40% of the theoretical peak.

A.1 Fundamental Modules

Kinds: Defines the floating point precision of the fundamental types used, and some useful constants.

```
module Kinds

  implicit none

  integer , parameter :: SP = kind(1.0)
  integer , parameter :: DP = kind(1.0D0)
  integer , parameter :: SC = kind((1.0,1.0))
  integer , parameter :: DC = kind((1.0D0,1.0D0))

  real(dp) , parameter :: pi=3.141592653589793238462643383279502884197_dp
  real(dp) , parameter :: two_pi = 6.283185307179586476925286766559005768394_dp

  complex(dc) , parameter :: I = ( 0.0d0, 1.0d0 )

end module Kinds
```

LatticeSize: Defines various parameters relating to the lattice size and field degrees of freedom per site, along with the level of shadowing and padding desired for each dimension.

```
module LatticeSize

  implicit none

  integer , parameter :: nlx=16, nly=16, nlz=16, nlt=32 ! Lattice geometry

  integer , parameter :: nprocx=1, nprocy=1, nprocz=8, nproct=16 ! Processor topology
  integer , parameter :: nproc = nprocx*nprocy*nprocz*nproct

  integer , parameter :: nx=nlx/nprocx , ny=nly/nprocy , nz=nlz/nprocz , nt=nlt/nproct ! Sub-lattice geometry

  integer , parameter :: shdwx = 0, shdwy=0, shdwz=2, shdwt=2 ! Shadowing
  integer , parameter :: nxs =nx+ shdwx , nys =ny+ shdwy , nzs =nz+ shdwz , nts =nt+ shdwt ! Single shadow
  integer , parameter :: nxss=nx+2*shdwx , nyss=ny+2*shdwy , nzss=nz+2*shdwz , ntss=nt+2*shdwt ! Double shadow

  integer , parameter :: padx = 0, pady=0, padz=1, padt=1 ! Dimension padding for caching
  integer , parameter :: nxp=nx+padx , nyp=ny+pady , nzp=nz+padz , ntp=nt+padt

  integer , parameter :: ns=4, nsp = 2, nc=3, nd=4 ! # Spinor , half-spinor , colour and space-time d.o.f.
  integer , parameter :: nplaq = 6 !# of unique plaquettes per site= nd*(nd-1)/2

  integer , parameter :: nlattice = nlx*nly*nlz*nlt ! lattice size
  integer , parameter :: nsublattice = nx*ny*nz*nt ! lattice size per processor
  integer , parameter :: nsublattice_s = nxs*nys*nzs*nts ! shadowed lattice size per processor
  integer , parameter :: nsublattice_ss = nxss*nyss*nzss*ntss ! shadowed lattice size per processor

  integer :: mapx(-1:nx+2) , mapy(-1:ny+2) , mapz(-1:nz+2) , mapt(-1:nt+2) ! shadow mapping

end module LatticeSize
```

ColourTypes: Defines colour vector and matrix types, used to take advantage of data locality, and provides extensions to the intrinsic random_number.

```
module ColourTypes

  use Kinds
  use LatticeSize
  implicit none

  !Cl => Colour

  type real_vector
     sequence
     real(dp) , dimension(nc) :: Cl
  end type real_vector

  type colour_vector
     sequence
```

```
    complex(dc), dimension(nc) :: Cl
  end type colour_vector

  type colour_matrix
    sequence
    complex(dc), dimension(nc,nc) :: Cl
  end type colour_matrix

  type(colour_vector), parameter :: zero_vector = colour_vector( (/ 0,0,0 /) )
  type(colour_matrix), parameter :: zero_matrix = colour_matrix( reshape( (/ 0,0,0,0,0,0,0,0,0 /)
       , (/ 3,3 /) ) )
  type(colour_matrix), parameter :: unit_matrix = colour_matrix( reshape( (/ 1,0,0,0,1,0,0,0,1 /)
       , (/ 3,3 /) ) )

contains

  subroutine random_complex(z)

    complex(dc), intent(out) :: z
    real(dp) :: zr, zi

    call random_number(zr)
    call random_number(zi)

    z = cmplx(zr,zi,dc)

  end subroutine random_complex

  subroutine random_vector(z)

    type(colour_vector), intent(out) :: z
    real(dp) :: zr(nc), zi(nc)

    call random_number(zr)
    call random_number(zi)

    z%Cl = cmplx(zr,zi,dc)

  end subroutine random_vector

  subroutine random_matrix(z)

    type(colour_matrix), intent(out) :: z
    real(dp) :: zr(nc,nc), zi(nc,nc)

    call random_number(zr)
    call random_number(zi)

    z%Cl = cmplx(zr,zi,dc)

  end subroutine random_matrix

end module ColourTypes
```

VectorAlgebra: Provides fundamental routines for manipulating colour vectors.

```
module VectorAlgebra

  use ColourTypes
  implicit none

contains

  pure subroutine normalise_vector(v)

    type(colour_vector), intent(inout) :: v
    real(dp) :: norm

    norm = sqrt(sum(real(v%Cl)**2 + aimag(v%Cl)**2))
    v%Cl = v%Cl/norm

  end subroutine normalise_vector

  pure function vector_inner_product(v,w) result (vdotw)

    type(colour_vector), intent(in) :: v, w
    complex(dc) :: vdotw

    vdotw = sum(conjg(v%Cl)*w%Cl)

  end function vector_inner_product

  pure function real_vector_inner_product(v,w) result (vdotw)

    type(colour_vector), intent(in) :: v, w
```

```
  real(dp) :: vdotw

  vdotw = sum(real(v%Cl)*real(w%Cl) + aimag(v%Cl)*aimag(w%Cl))

end function real_vector_inner_product

pure function vector_norm(v) result (norm)

  type(colour_vector), intent(in) :: v
  real(dp) :: norm

  norm = sqrt(sum(real(v%Cl)**2 + aimag(v%Cl)**2))

end function vector_norm

pure function vector_normsq(v) result (normsq)

  type(colour_vector), intent(in) :: v
  real(dp) :: normsq

  normsq = sum(real(v%Cl)**2 + aimag(v%Cl)**2)

end function vector_normsq

pure subroutine orthogonalise_vectors(w,v)

  type(colour_vector), intent(inout) :: w
  type(colour_vector), intent(in) :: v
  complex(dc) :: vdotw

  vdotw = sum(conjg(v%Cl)*w%Cl)

  w%Cl = w%Cl - v%Cl*vdotw

end subroutine orthogonalise_vectors

pure subroutine vector_product(x,v,w)

  type(colour_vector), intent(out) :: x
  type(colour_vector), intent(in) :: v,w
  integer :: ic,jc,kc

  do ic=1,nc
     jc = modulo(ic,3)+1
     kc = modulo(jc,3)+1

     x%Cl(ic) = conjg(v%Cl(jc)*w%Cl(kc) - v%Cl(kc)*w%Cl(jc))
  end do

end subroutine vector_product

subroutine MultiplyMatClrVec(times,left,right)

  type(colour_vector), intent(out) :: times
  type(colour_matrix), intent(in)  :: left
  type(colour_vector), intent(in) :: right

  times%Cl(ic) = sum(left%Cl(ic,:)*right%Cl(:)) ! *unroll over ic*

end subroutine MultiplyMatClrVec

subroutine MultiplyMatDagClrVec(times,left,right)

  type(colour_vector), intent(out) :: times
  type(colour_matrix), intent(in)  :: left
  type(colour_vector), intent(in) :: right

  times%Cl(ic) = sum(conjg(left%Cl(:,ic))*right%Cl(:)) ! *unroll over ic*

end subroutine MultiplyMatDagClrVec

subroutine MultiplyClrVecDagMat(times,left,right)

  type(colour_vector), intent(out) :: times
  type(colour_vector), intent(in) :: left
  type(colour_matrix), intent(in)  :: right

  times%Cl(ic) = sum(conjg(left%Cl(:))*right%Cl(:,ic)) ! *unroll over ic*

end subroutine MultiplyClrVecDagMat

subroutine MultiplyClrVecDagMatDag(times,left,right)

  type(colour_vector), intent(out) :: times
  type(colour_vector), intent(in) :: left
  type(colour_matrix), intent(in)  :: right

  times%Cl(ic) = sum(conjg(left%Cl(:)*right%Cl(ic,:))) ! *unroll over ic*
```

```
  end subroutine MultiplyClrVecDagMatDag

  pure subroutine VecDag(dag, left)

    type(colour_vector), dimension(ns), intent(out) :: dag
    type(colour_vector), dimension(ns), intent(in) :: left
    integer :: is

    do is=1,ns
       dag(is)%Cl = conjg(left(is)%Cl)
    end do

  end subroutine VecDag

  subroutine MultiplyMatVec(times, left, right)

    type(colour_vector), dimension(ns), intent(out) :: times
    type(colour_matrix), intent(in)  :: left
    type(colour_vector), dimension(ns), intent(in) :: right

    integer :: is

    do is=1,ns
       times(is)%Cl(ic) = sum(left%Cl(ic,:)*right(is)%Cl(:)) ! *unroll over ic*
    end do

  end subroutine MultiplyMatVec

  subroutine MultiplyMatDagVec(times, left, right)

    type(colour_vector), dimension(ns), intent(out) :: times
    type(colour_matrix), intent(in)  :: left
    type(colour_vector), dimension(ns), intent(in) :: right

    integer :: is

    do is=1,ns
       times(is)%Cl(ic) = sum(conjg(left%Cl(:,ic))*right(is)%Cl(:)) ! *unroll over ic*
    end do

  end subroutine MultiplyMatDagVec

  subroutine MultiplyVecDagMat(times, left, right)

    type(colour_vector), dimension(ns), intent(out) :: times
    type(colour_vector), dimension(ns), intent(in) :: left
    type(colour_matrix), intent(in)  :: right

    integer :: is

    do is=1,ns
       times(is)%Cl(ic) = sum(conjg(left(is)%Cl(:))*right%Cl(:,ic)) ! *unroll over ic*
    end do

  end subroutine MultiplyVecDagMat

  subroutine MultiplyVecDagMatDag(times, left, right)

    type(colour_vector), dimension(ns), intent(out) :: times
    type(colour_vector), dimension(ns), intent(in) :: left
    type(colour_matrix), intent(in)  :: right

    integer :: is

    do is=1,ns
       times(is)%Cl(ic) = sum(conjg(left(is)%Cl(:)*right%Cl(ic,:))) ! *unroll over ic*
    end do

  end subroutine MultiplyVecDagMatDag

end module VectorAlgebra
```

MatrixAlgebra: Provides fundamental routines for manipulating colour matrices.

```
module MatrixAlgebra

  use ColourTypes
  implicit none

  private :: real_times_matrix

  interface operator(*)
     module procedure real_times_matrix
  end interface
```

```
contains

  elemental function real_times_matrix(left, right) result (times)

    real(dp), intent(in) :: left
    type(colour_matrix), intent(in) :: right
    type(colour_matrix) :: times

    times%Cl = left*Right%Cl

  end function real_times_matrix

  elemental function real_trace(right) result (Tr)

    real(dp) :: Tr
    type(colour_matrix), intent(in) :: right

    Tr = real(right%Cl(1,1) + right%Cl(2,2) + right%Cl(3,3))

  end function real_trace

  elemental function trace(right) result (Tr)

    complex(dc) :: Tr
    type(colour_matrix), intent(in) :: right

    Tr = right%Cl(1,1) + right%Cl(2,2) + right%Cl(3,3)

  end function trace

  elemental function matrix_normsq(right) result (norm)

    real(dp) :: norm
    type(colour_matrix), intent(in) :: right

    norm = sum(real(conjg(right%Cl)*right%Cl))

  end function matrix_normsq

  elemental subroutine GetIdentityMatrix(I_x)

    type(colour_matrix), intent(out) :: I_x

    I_x%Cl = 0.0d0

    I_x%Cl(1,1) = 1.0d0
    I_x%Cl(2,2) = 1.0d0
    I_x%Cl(3,3) = 1.0d0

  end subroutine GetIdentityMatrix

  elemental subroutine MatDag(dag, left)

    type(colour_matrix), intent(in) :: left
    type(colour_matrix), intent(out) :: dag

    dag%Cl(ic,jc) = conjg(left%Cl(ic,jc)) ! *unroll*

  end subroutine MatDag

  elemental subroutine MultiplyMatMat(MM, left, right)

    type(colour_matrix), intent(in) :: left
    type(colour_matrix), intent(in) :: right
    type(colour_matrix), intent(out) :: MM

    MM%Cl(ic,jc) = sum(left%Cl(ic,:)*right%Cl(:,jc)) ! *unroll*

  end subroutine MultiplyMatMat

  elemental subroutine MultiplyMatMatdag(MM, left, right)

    type(colour_matrix), intent(in) :: left
    type(colour_matrix), intent(in) :: right
    type(colour_matrix), intent(out) :: MM

    MM%Cl(ic,jc) = sum(left%Cl(ic,:)*conjg(right%Cl(jc,:))) ! *unroll*

  end subroutine MultiplyMatMatdag

  elemental subroutine MultiplyMatdagMat(MM, left, right)

    type(colour_matrix), intent(in) :: left
    type(colour_matrix), intent(in) :: right
    type(colour_matrix), intent(out) :: MM

    MM%Cl(ic,jc) = sum(conjg(left%Cl(:,ic))*right%Cl(:,jc)) ! *unroll*
```

```
  end subroutine MultiplyMatdagMat

  elemental subroutine MultiplyMatdagMatdag(MM, left, right)

    type(colour_matrix), intent(in) :: left
    type(colour_matrix), intent(in) :: right
    type(colour_matrix), intent(out) :: MM

    MM%Cl(ic,jc) = sum(conjg(left%Cl(:,ic)*right%Cl(jc,:))) ! *unroll*

  end subroutine MultiplyMatdagMatdag

  elemental subroutine RealTraceMultMatMat(TrMM, left, right)

    type(colour_matrix), intent(in) :: left
    type(colour_matrix), intent(in) :: right
    real(dp), intent(out) :: TrMM

    TrMM = real(sum( left%Cl(:,:)*transpose(right%Cl(:,:)) )) ! *unroll*

  end subroutine RealTraceMultMatMat

  elemental subroutine MultiplyMatDiagMat(MM, left, right)

    type(colour_matrix), intent(in) :: left
    type(colour_vector), intent(in) :: right
    type(colour_matrix), intent(out) :: MM

    MM%Cl(ic,jc) = left%Cl(ic,jc)*right%Cl(jc) ! *unroll*

  end subroutine MultiplyMatDiagMat

  elemental subroutine MultiplyMatRealDiagMat(MM, left, right)

    type(colour_matrix), intent(in) :: left
    type(real_vector), intent(in) :: right
    type(colour_matrix), intent(out) :: MM

    MM%Cl(ic,jc) = left%Cl(ic,jc)*right%Cl(jc) ! *unroll*

  end subroutine MultiplyMatRealDiagMat

  elemental subroutine MatPlusMatTimesMat(MM, left, right)

    type(colour_matrix), intent(in) :: left
    type(colour_matrix), intent(in) :: right
    type(colour_matrix), intent(inout) :: MM

    MM%Cl(ic,jc) = MM%Cl(ic,jc) + sum(left%Cl(ic,:)*right%Cl(:,jc)) ! *unroll*

  end subroutine MatPlusMatTimesMat

  elemental subroutine MatPlusMatTimesMatdag(MM, left, right)

    type(colour_matrix), intent(in) :: left
    type(colour_matrix), intent(in) :: right
    type(colour_matrix), intent(inout) :: MM

    MM%Cl(ic,jc) = MM%Cl(ic,jc) + sum(left%Cl(ic,:)*conjg(right%Cl(jc,:))) ! *unroll*

  end subroutine MatPlusMatTimesMatdag

  elemental subroutine MatPlusMatdagTimesMat(MM, left, right)

    type(colour_matrix), intent(in) :: left
    type(colour_matrix), intent(in) :: right
    type(colour_matrix), intent(inout) :: MM

    MM%Cl(ic,jc) = MM%Cl(ic,jc) + sum(conjg(left%Cl(:,ic))*right%Cl(:,jc)) ! *unroll*

  end subroutine MatPlusMatdagTimesMat

  elemental subroutine MatPlusMatdagTimesMatdag(MM, left, right)

    type(colour_matrix), intent(in) :: left
    type(colour_matrix), intent(in) :: right
    type(colour_matrix), intent(inout) :: MM

    MM%Cl(ic,jc) = MM%Cl(ic,jc) + sum(conjg(left%Cl(:,ic)*right%Cl(jc,:))) ! *unroll*

  end subroutine MatPlusMatdagTimesMatdag

end module MatrixAlgebra
```

A.2 Communication Modules

MPIInterface: Provides a higher level interface to MPI.

```
module MPIInterface

  use ColourTypes
  implicit none

  include "mpif.h"

  integer, parameter :: mpi_comm = MPI_COMM_WORLD, mpi_root_rank = 0
  integer :: mpi_size, mpi_rank
  integer, dimension(nd) :: mpi_coords
  integer :: i_nx, j_nx, i_ny, j_ny, i_nt, j_nt, i_nz, j_nz !sublattice boundaries
  logical :: i_am_root

  integer, parameter :: mpi_dp = MPI_DOUBLE_PRECISION, mpi_dc = MPI_DOUBLE_COMPLEX
  integer, parameter :: nmpi_status = MPI_STATUS_SIZE
  integer, parameter :: nmpi_header = MPI_BSEND_OVERHEAD

#define mpiprint if (i_am_root) print

contains

  function coords_to_rank(coords) result (rank)

    integer, dimension(nd) :: coords(nd)
    integer :: rank

    rank = coords(4) + nproct*coords(3) + nproct*nprocz*coords(2)

  end function coords_to_rank

  function rank_to_coords(rank) result (coords)

    integer :: rank
    integer, dimension(nd) :: coords(nd)

    coords(4) = mod(rank,nproct)
    coords(3) = mod(rank/nproct,nprocz)
    coords(2) = rank/(nproct*nprocz)
    coords(1) = 0

  end function rank_to_coords

  subroutine GetSubLattice(irank,ix,jx,iy,jy,iz,jz,it,jt)

    integer :: irank,ix,jx,iy,jy,iz,jz,it,jt
    integer, dimension(nd) :: coords(nd)

    coords = rank_to_coords(irank)

    ix = nx*coords(1)+ 1
    jx = nx*(coords(1) + 1)

    iy = ny*coords(2)+ 1
    jy = ny*(coords(2) + 1)

    iz = nz*coords(3)+ 1
    jz = nz*(coords(3) + 1)

    it = nt*coords(4)+ 1
    jt = nt*(coords(4) + 1)

  end subroutine GetSubLattice

  function site_is_mine(ix,iy,iz,it)

    integer :: ix,iy,iz,it
    logical :: site_is_mine

    site_is_mine = (i_nx <= ix) .and. (ix <= j_nx) .and. (i_ny <= iy) .and. (iy <= j_ny) .and. &
       &          (i_nz <= iz) .and. (iz <= j_nz) .and. (i_nt <= it) .and. (it <= j_nt)

  end function site_is_mine

  subroutine LatticeToSubLattice(ix,iy,iz,it,jx,jy,jz,jt)

    integer :: ix,jx,iy,jy,iz,jz,it,jt

    jx = ix - i_nx + 1
    jy = iy - i_ny + 1
    jz = iz - i_nz + 1
    jt = it - i_nt + 1
```

```
end subroutine LatticeToSubLattice

subroutine InitialiseMPI

  integer :: mpierror

  integer :: ix,iy,iz,it,is,id

  call MPI_Init(mpierror)
  call MPI_ErrorCheck(mpierror)

  call MPI_Comm_size(mpi_comm, mpi_size, mpierror)
  call MPI_ErrorCheck(mpierror)

  call MPI_Comm_rank(mpi_comm, mpi_rank, mpierror)
  call MPI_ErrorCheck(mpierror)

  i_am_root = (mpi_rank == mpi_root_rank )
  mpi_coords = rank_to_coords(mpi_rank)

  call GetSubLattice(mpi_rank,i_nx,j_nx,i_ny,j_ny,i_nz,j_nz,i_nt,j_nt)

  mapx(-1) = nx-1
  mapx(0) = nx
  do ix=1,nx
     mapx(ix) = ix
  end do
  mapx(nx+1) = 1
  mapx(nx+2) = 2

  do iy=1,ny
     mapy(iy) = iy
  end do

  if ( nprocy == 1 ) then
     mapy(-1) = ny-1
     mapy(0) = ny
     mapy(ny+1) = 1
     mapy(ny+2) = 2
  else
     mapy(-1) = ny+4
     mapy(0) = ny+2
     mapy(ny+1) = ny+1
     mapy(ny+2) = ny+3
  end if

  do iz=1,nz
     mapz(iz) = iz
  end do

  if ( nprocz == 1 ) then
     mapz(-1) = nz-1
     mapz(0) = nz
     mapz(nz+1) = 1
     mapz(nz+2) = 2
  else
     mapz(-1) = nz+4
     mapz(0) = nz+2
     mapz(nz+1) = nz+1
     mapz(nz+2) = nz+3
  end if

  mapt(-1) = nt+4
  mapt(0) = nt+2
  do it=1,nt+1
     mapt(it) = it
  end do
  mapt(nt+2) = nt+3

end subroutine InitialiseMPI

subroutine AbortMPI(error)

  character(len=*) :: error
  integer :: ierror

  print *, "User abort: ", error
  call MPI_Abort(mpi_comm,MPI_ERR_OTHER,ierror)

end subroutine AbortMPI

subroutine BroadcastLogical(l,root_rank)

  logical :: l
  integer :: root_rank
  integer :: mpierror

  call MPI_BCast(l,1,mpi_logical,root_rank,mpi_comm,mpierror)
```

```
    call MPI_ErrorCheck(mpierror)

  end subroutine BroadcastLogical

  subroutine BroadcastInteger(n, root_rank)

    integer :: n, root_rank
    integer :: mpierror

    call MPI_BCast(n,1,mpi_integer,root_rank,mpi_comm,mpierror)
    call MPI_ErrorCheck(mpierror)

  end subroutine BroadcastInteger

  subroutine BroadcastReal(x, root_rank)

    real(DP) :: x
    integer :: mpierror, root_rank

    call MPI_BCast(x,1,mpi_dp,root_rank,mpi_comm,mpierror)
    call MPI_ErrorCheck(mpierror)

  end subroutine BroadCastReal

  subroutine BroadcastComplex(z, root_rank)

    complex(DC) :: z
    integer :: mpierror, root_rank

    call MPI_BCast(z,1,mpi_dc,root_rank,mpi_comm,mpierror)
    call MPI_ErrorCheck(mpierror)

  end subroutine BroadCastComplex

  subroutine BroadcastString(s, root_rank)

    character(len=*) :: s
    integer :: mpierror, root_rank

    call MPI_BCast(s,len(s),mpi_character,root_rank,mpi_comm,mpierror)
    call MPI_ErrorCheck(mpierror)

  end subroutine BroadcastString

  subroutine AllSumInteger(summand, sum)

    real(DP) :: summand, sum
    integer :: mpierror

    call MPI_AllReduce(summand,sum,1,mpi_integer,MPI_SUM,mpi_comm,mpierror)
    call MPI_ErrorCheck(mpierror)

  end subroutine AllSumInteger

  subroutine AllSumReal(summand, sum)

    real(DP) :: summand, sum
    integer :: mpierror

    call MPI_AllReduce(summand,sum,1,mpi_dp,MPI_SUM,mpi_comm,mpierror)
    call MPI_ErrorCheck(mpierror)

  end subroutine AllSumReal

  subroutine AllSumComplex(summand, sum)

    complex(DC) :: summand, sum
    integer :: mpierror

    call MPI_AllReduce(summand,sum,1,mpi_dc,MPI_SUM,mpi_comm,mpierror)
    call MPI_ErrorCheck(mpierror)

  end subroutine AllSumComplex

  subroutine Wait(mpi_request)

    integer :: mpi_request
    integer, dimension(nmpi_status) :: mpi_status
    integer :: mpierror

    call MPI_Wait(mpi_request, mpi_status, mpierror)
    call MPI_ErrorCheck(mpierror)

  end subroutine Wait

  subroutine Start(mpi_request)

    integer :: mpi_request
```

```
  integer, dimension(nmpi_status) :: mpi_status
  integer :: mpierror

  call MPI_Start(mpi_request, mpierror)
  call MPI_ErrorCheck(mpierror)

end subroutine Start

subroutine MPIBarrier

  integer :: mpierror

  call MPI_Barrier(mpi_comm, mpierror)
  call MPI_ErrorCheck(mpierror)

end subroutine MPIBarrier

subroutine FinaliseMPI

  integer :: mpierror

  call MPI_Finalize(mpierror)
  call MPI_ErrorCheck(mpierror)

end subroutine FinaliseMPI

subroutine MPI_ErrorCheck(mpierror)

  integer :: mpierror
  integer :: errorcode, errorclass
  character(len=MPI_MAX_ERROR_STRING) :: errorstring
  integer :: length, ierror

  if (mpierror /= MPI_SUCCESS ) then
     call MPI_ERROR_CLASS(mpierror, errorclass, ierror)
     call MPI_ERROR_STRING( mpierror, errorstring, length, ierror )
     mpiprint *, "Errorcode", mpierror, errorstring(1:length)
     call MPI_ERROR_STRING( errorclass, errorstring, length, ierror )
     mpiprint *, "Errorclass", mpierror, errorstring(1:length)
     call MPI_Abort(mpi_comm, mpierror, ierror)
  end if

end subroutine MPI_ErrorCheck

end module MPIInterface
```

RealFieldMPIComms: Provides high level communication routines for complex fields. The communication details should be optimised for the underlying hardware. For this reason, only subprogram outlines have been given here.

```
module RealFieldMPIComms

  use MPIInterface
  implicit none

  interface SendRealField
     module procedure SendRealField_0
     module procedure SendRealField_1
     module procedure SendRealField_2
     module procedure SendRealField_3
     module procedure SendRealField_4
     module procedure SendRealField_5
     module procedure SendRealField_6
     module procedure SendRealField_7
  end interface

  interface RecvRealField
     module procedure RecvRealField_0
     module procedure RecvRealField_1
     module procedure RecvRealField_2
     module procedure RecvRealField_3
     module procedure RecvRealField_4
     module procedure RecvRealField_5
     module procedure RecvRealField_6
     module procedure RecvRealField_7
  end interface

contains

  subroutine SendRealField_*(phi, dest_rank)

    real(dp), dimension(rank=*) :: phi
    integer :: dest_rank
```

```
    !This procedure is an abstraction of MPI_Send.

  end subroutine SendRealField_*

  subroutine RecvRealField_*(phi, src_rank)

    real(dp), dimension(:) :: phi
    integer :: src_rank

    !This procedure is an abstraction of MPI_Recv.

  end subroutine RecvRealField_*

end module RealFieldMPIComms
```

ComplexFieldMPIComms: Provides high level communication routines for complex fields. The communication details should be optimised for the underlying hardware. For this reason, only subprogram outlines have been given here.

```
module ComplexFieldMPIComms

  use MPIInterface
  implicit none

  interface SendComplexField
     module procedure SendComplexField_1
     module procedure SendComplexField_2
     module procedure SendComplexField_3
     module procedure SendComplexField_4
     module procedure SendComplexField_5
     module procedure SendComplexField_6
     module procedure SendComplexField_7
  end interface

  interface RecvComplexField
     module procedure RecvComplexField_1
     module procedure RecvComplexField_2
     module procedure RecvComplexField_3
     module procedure RecvComplexField_4
     module procedure RecvComplexField_5
     module procedure RecvComplexField_6
     module procedure RecvComplexField_7
  end interface

contains

  subroutine SendComplexField_*(phi, dest_rank)

    complex(dc), dimension(rank=*) :: phi
    integer :: dest_rank

    !This procedure is an abstraction of MPI_Send.

  end subroutine SendComplexField_*

  subroutine RecvComplexField_i(phi, src_rank)

    complex(dc), dimension(rank=*) :: phi
    integer :: src_rank

    !This procedure is an abstraction of MPI_Recv.

  end subroutine RecvComplexField_*

end module ComplexFieldMPIComms
```

GaugeFieldMPIComms: Provides high level communication routines for gauge fields. The communication details should be optimised for the underlying hardware. For this reason, only subprogram outlines have been given here.

```
module GaugeFieldMPIComms

  use MPIInterface
  use ColourTypes
  implicit none

  interface SendMatrixField
     module procedure SendMatrixField_1
     module procedure SendMatrixField_2
     module procedure SendMatrixField_3
     module procedure SendMatrixField_4
```

```
    module procedure SendMatrixField_5
    module procedure SendMatrixField_6
    module procedure SendMatrixField_7
  end interface

  interface RecvMatrixField
    module procedure RecvMatrixField_1
    module procedure RecvMatrixField_2
    module procedure RecvMatrixField_3
    module procedure RecvMatrixField_4
    module procedure RecvMatrixField_5
    module procedure RecvMatrixField_6
    module procedure RecvMatrixField_7
  end interface

  interface SendRecvMatrixField
    module procedure SendRecvMatrixField_1
    module procedure SendRecvMatrixField_2
    module procedure SendRecvMatrixField_3
    module procedure SendRecvMatrixField_4
    module procedure SendRecvMatrixField_5
    module procedure SendRecvMatrixField_6
    module procedure SendRecvMatrixField_7
  end interface

contains

  subroutine InitShadowGaugeField

    integer :: ishdw, src_rank, dest_rank, iy, iz, it

    ! Perform any initialisation necessary for ShadowGaugeField.

  end subroutine InitShadowGaugeField

  subroutine ShadowGaugeField(U_xd,ishdw)

    type(colour_matrix), dimension(:,:,:,:,:) :: U_xd
    integer :: ishdw

    ! This procedure performs communication dependent upon the underlying processor topology.
    ! Its purpose is to "shadow" the per processor sub-lattice with slices from the neighbouring
    ! processors, to minimise communication costs. The slices which are shadowed are determined by ishdw.

  end subroutine ShadowGaugeField

  subroutine SendMatrixField_*(phi, dest_rank)

    type(colour_matrix), dimension(rank=*) :: phi
    integer :: dest_rank
    integer :: mpierror, tag
    tag = size(phi)

    ! This procedure is an abstraction of MPI_Send.

  end subroutine SendMatrixField_*

  subroutine RecvMatrixField_*(phi, src_rank)

    type(colour_matrix), dimension(rank=*) :: phi
    integer :: src_rank

    ! This procedure is an abstraction of MPI_Recv.

  end subroutine RecvMatrixField_*

  subroutine SendRecvMatrixField_*(phi_send, dest_rank, phi_recv, src_rank)

    type(colour_matrix), dimension(rank=*) :: phi_send, phi_recv
    integer :: dest_rank, src_rank

    ! This procedure is an abstraction of MPI_SendRecv.

  end subroutine SendRecvMatrixField_*

end module GaugeFieldMPIComms
```

FermionFieldMPIComms: Provides high level communication routines for fermion fields. The communication details should be optimised for the underlying hardware. For this reason, only subprogram outlines have been given here.

```
module FermionFieldMPIComms

  use MPIInterface
  use ColourTypes
```

```
  use Timer
  implicit none

  interface SendFermionField
     module procedure SendFermionField_1
     module procedure SendFermionField_2
     module procedure SendFermionField_3
     module procedure SendFermionField_4
     module procedure SendFermionField_5
     module procedure SendFermionField_6
     module procedure SendFermionField_7
  end interface

  interface RecvFermionField
     module procedure RecvFermionField_1
     module procedure RecvFermionField_2
     module procedure RecvFermionField_3
     module procedure RecvFermionField_4
     module procedure RecvFermionField_5
     module procedure RecvFermionField_6
     module procedure RecvFermionField_7
  end interface

contains

  subroutine InitShadowFermionField

    integer :: ishdw, src_rank, dest_rank, iy, iz, it

    ! Perform any initialisation necessary for ShadowFermionField.

  end subroutine InitShadowFermionField

  subroutine ShadowFermionField(phi,ishdw)

    type(colour_vector), dimension(nxs,nys,nzs,nts,ns) :: phi
    integer :: ishdw

    ! This procedure performs communication dependent upon the underlying processor topology.
    ! Its purpose is to "shadow" the per processor sub-lattice with slices from the neighbouring
    ! processors, to minimise communication costs. The slices which are shadowed are determined by ishdw.

  end subroutine ShadowFermionField

  subroutine SendFermionField_*(phi, dest_rank)

    type(colour_vector), dimension(rank=*) :: phi
    integer :: dest_rank

    ! This procedure is an abstraction of MPI_Send.

  end subroutine SendFermionField_*

  subroutine RecvFermionField_*(phi, src_rank)

    type(colour_vector), dimension(rank=*) :: phi
    integer :: src_rank

    ! This procedure is an abstraction of MPI_Recv.

  end subroutine RecvFermionField_*

  subroutine SendRecvFermionField_*(phi_send, dest_rank, phi_recv, src_rank)

    type(colour_vector), dimension(:,:,:,:) :: phi_send, phi_recv
    integer :: dest_rank, src_rank

    ! This procedure is an abstraction of MPI_SendRecv.

  end subroutine SendRecvFermionField_*

end module FermionFieldMPIComms
```

MPIRandomSeed: Provides a parallel random number seeder.

```
module MPIRandomSeed

  use Kinds
  use MPIInterface
  implicit none

contains

  subroutine mpi_random_seed(iseed)

    implicit none
    integer :: iseed
```

```
    integer :: i,N
    integer, dimension(:), allocatable :: seeder_array

    ! Adapted to MPI from the HPF routine by Paul Coddington.
    ! Find out the size of the seed array used in the Fortran 90 random number generator
    call random_seed(size=N)

    allocate ( seeder_array(N) )

    !This code executes identically in each process,
    !so we must include a rank dependent seed to make sure that
    !we get different random numbers on each process.

    call bitrand_seeder(iseed+mpi_rank,N,seeder_array)

    call random_seed( put=seeder_array(1:N) )

    deallocate( seeder_array )

  end subroutine mpi_random_seed

  subroutine bitrand_seeder(seed,N,seed_array)

    integer :: seed,N
    integer :: seed_array(N)

    ! Initialize each bit of the 32-bit integer seed array using a simple linear congruential generator,
    ! or some other portable random number generator, see for example Numerical Recipes.

  end subroutine bitrand_seeder

end module MPIRandomSeed
```

Timer: Provides support for execution time statistics.

```
module Timer

  use Kinds
  use MPIInterface

  logical, parameter :: timing = .true.

  real(dp) :: clockres
  real :: Mflops

contains

  subroutine InitTimer

    clockres = mpi_wtick()

  end subroutine InitTimer

  subroutine TimingUpdate(ncalls,outtime,intime,mintime,maxtime,meantime)

    integer :: ncalls
    real(DP) :: intime,outtime,maxtime, mintime,meantime

    ncalls = ncalls + 1
    if ( ncalls == 1 ) mintime = outtime - intime
    if ( outtime - intime < mintime ) mintime = outtime - intime
    if ( outtime - intime > maxtime ) maxtime = outtime - intime
    meantime = (ncalls - 1)*meantime/ncalls + (outtime - intime)/ncalls

  end subroutine TimingUpdate

end module Timer
```

A.3 Common Modules

GaugeField: Provide the fundamental routines relating to gauge fields, including input and output.

```
module GaugeField

  use GaugeFieldMPIComms
  use VectorAlgebra
  use MatrixAlgebra
  implicit none
```

```fortran
  type(colour_matrix), dimension(nxs,nys,nzs,nts,nd) :: U_xd

  real(DP) :: beta,lastPlaq,plaqbarAvg,uzero,u0_bar = 1.0d0

contains

  subroutine ReadGaugeField(filename, U_xd)

    character(len=*) :: filename
    type(colour_matrix), dimension(nxs,nys,nzs,nts,nd), intent(out) :: U_xd

    real(DP), dimension(:,:,:,:,:,:,:), allocatable :: ReU, ImU
    integer :: ic, jc, irank
    integer :: ix, jx, iy, jy, iz, jz, it, jt

    type(colour_matrix), dimension(nx,ny,nz,nt,nd) :: V_xd
    integer    :: nconfig,nxdim,nydim,nzdim,ntdim

    if ( i_am_root ) then

       allocate(ReU(nlx,nly,nlz,nlt,nd,nc,nc))
       allocate(ImU(nlx,nly,nlz,nlt,nd,nc,nc))

       ReU = 0.0d0
       ImU = 0.0d0

       open(101,file=filename,form='unformatted',status='old',action='read')

       read (101) nconfig,beta,nxdim,nydim,nzdim,ntdim

       if ( (nxdim /= nlx) .or. (nydim /= nly) .or. (nzdim /= nlz) .or. (ntdim /= nlt) ) then
          call AbortMPI("ReadGaugeField: Lattice size mismatch")
       end if

       do ic = 1, nc-1
          read (101) ReU(:,:,:,:,:,ic,:)
          read (101) ImU(:,:,:,:,:,ic,:)
       end do

       read (101) lastPlaq, plaqbarAvg, uzero
       close(101)

       do irank=0,nproc-1
          call GetSubLattice(irank,ix,jx,iy,jy,iz,jz,it,jt)
          do jc=1,nc; do ic=1,nc
             V_xd(1:nx,1:ny,1:nz,1:nt,:)%Cl(ic,jc) = cmplx(ReU(ix:jx,iy:jy,iz:jz,it:jt,:,ic,jc),&
                  & ImU(ix:jx,iy:jy,iz:jz,it:jt,:,ic,jc),dc)
          end do; end do
          if ( irank /= mpi_root_rank ) then
             call SendMatrixField(V_xd, irank)
          else
             U_xd(1:nx,1:ny,1:nz,1:nt,:)   = V_xd
          end if
       end do

       deallocate(ReU)
       deallocate(ImU)

    else
       call RecvMatrixField(V_xd, mpi_root_rank)
       U_xd(1:nx,1:ny,1:nz,1:nt,:)   = V_xd
    end if

    call BroadcastReal(beta,mpi_root_rank)
    call BroadcastReal(lastPlaq,mpi_root_rank)
    call BroadcastReal(plaqbarAvg,mpi_root_rank)
    call BroadcastReal(uzero,mpi_root_rank)

    call FixSU3(U_xd)

    call ShadowGaugeField(U_xd,1)

  end subroutine ReadGaugeField

  subroutine WriteGaugeField(filename, U_xd, icfg)

    character(len=*) :: filename
    type(colour_matrix), dimension(nxs,nys,nzs,nts,nd) :: U_xd
    integer :: icfg

    integer :: ic, jc, irank
    integer :: ix, jx, iy, jy, iz, jz, it, jt

    real(DP), dimension(:,:,:,:,:,:,:), allocatable :: ReU, ImU

    type(colour_matrix), dimension(nx,ny,nz,nt,nd) :: V_xd

    if ( i_am_root ) then
```

```
      allocate(ReU(nlx,nly,nlz,nlt,nd,nc,nc))
      allocate(ImU(nlx,nly,nlz,nlt,nd,nc,nc))

      do irank=0,nproc-1
         if ( irank /= mpi_root_rank ) then
            call RecvMatrixField(V_xd, irank)
         else
            V_xd = U_xd(1:nx,1:ny,1:nz,1:nt,:)
         end if
         call GetSubLattice(irank,ix,jx,iy,jy,iz,jz,it,jt)
         do jc=1,nc; do ic=1,nc
            ReU(ix:jx,iy:jy,iz:jz,it:jt,:,ic,jc) =  real(V_xd(1:nx,1:ny,1:nz,1:nt,:)%Cl(ic,jc))
            ImU(ix:jx,iy:jy,iz:jz,it:jt,:,ic,jc) = aimag(V_xd(1:nx,1:ny,1:nz,1:nt,:)%Cl(ic,jc))
         end do; end do

      end do

      open(101,file=filename,form='unformatted',status='new',action='write')

      write (101) icfg,beta,nlx,nly,nlz,nlt

      do ic = 1, nc-1
         write (101) ReU(:,:,:,:,:,ic,:)
         write (101) ImU(:,:,:,:,:,ic,:)
      end do

      write (101) lastPlaq, plaqbarAvg, uzero
      close(101)

      deallocate(ReU)
      deallocate(ImU)

   else
      V_xd = U_xd(1:nx,1:ny,1:nz,1:nt,:)
      call SendMatrixField(V_xd, mpi_root_rank)
   end if

end subroutine WriteGaugeField

subroutine FixSU3(U_xd)

   type(colour_matrix), dimension(:,:,:,:,:), intent(inout) :: U_xd
   type(colour_vector) :: v1,v2,v3
   integer :: ix, iy, iz, it
   integer :: sx, sy, sz, st
   integer :: id

   do id=1,nd
      do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx

         v1%Cl(:) = U_xd(ix,iy,iz,it,id)%Cl(1,:)

         call normalise_vector(v1)

         v2%Cl(:) = U_xd(ix,iy,iz,it,id)%Cl(2,:)

         call orthogonalise_vectors(v2,v1)

         call normalise_vector(v2)

         call vector_product(v3,v1,v2)

         U_xd(ix,iy,iz,it,id)%Cl(1,:) = v1%Cl(:)
         U_xd(ix,iy,iz,it,id)%Cl(2,:) = v2%Cl(:)
         U_xd(ix,iy,iz,it,id)%Cl(3,:) = v3%Cl(:)

      end do; end do; end do; end do
   end do

end subroutine FixSU3

subroutine GetUZero(U_xd, uzero)

   type(colour_matrix), dimension(:,:,:,:,:) :: U_xd
   real(dp) :: uzero

   integer :: mu, nu
   real(dp) :: plaqbar, plaqbarpp

   plaqbarpp = 0.0d0

   do mu=1,nd
      do nu=mu+1,nd

         call AddPlaqBar_munu(plaqbarpp,U_xd,mu,nu)

      end do
```

```
    end do

    call AllSumReal(plaqbarpp, plaqbar)
    !u_0 = (Mean (1/3)ReTr U_munu)^(1/4)
    uzero = (plaqbar/(nlattice*nplaq*nc))**0.25d0

  end subroutine GetUZero

  function GetPlaqBar(U_xd) result(plaqbar)

    type(colour_matrix), dimension(:,:,:,:,:) :: U_xd
    real(dp) :: uzero

    integer :: mu, nu
    real(dp) :: plaqbar, plaqbarpp

    plaqbarpp = 0.0d0

    do mu=1,nd
       do nu=mu+1,nd

          call AddPlaqBar_munu(plaqbarpp,U_xd,mu,nu)

       end do
    end do

    call AllSumReal(plaqbarpp, plaqbar)

  end function GetPlaqBar

  subroutine SetBoundaryConditions(U_xd, bcx, bcy, bcz, bct)

    type(colour_matrix), dimension(:,:,:,:,:), intent(out) :: U_xd
    real(dp) :: bcx, bcy, bcz, bct

    if ( j_nx == nlx ) U_xd(nx,:,:,:,1) = bcx*U_xd(nx,:,:,:,1)
    if ( j_ny == nly ) U_xd(:,ny,:,:,2) = bcy*U_xd(:,ny,:,:,2)
    if ( j_nz == nlz ) U_xd(:,:,nz,:,3) = bcz*U_xd(:,:,nz,:,3)
    if ( j_nt == nlt ) U_xd(:,:,:,nt,4) = bct*U_xd(:,:,:,nt,4)

  end subroutine SetBoundaryConditions

  subroutine AddPlaqBar_munu(plaqbar,U_xd,mu,nu)

    real(dp) :: plaqbar
    type(colour_matrix), dimension(:,:,:,:,:) :: U_xd
    integer :: mu, nu

    type(colour_matrix) :: UmuUnu,UmudagUnudag
    real(dp) :: plaq_x

    integer, dimension(nd) :: dmu, dnu
    integer :: ix,iy,iz,it
    integer :: jx,jy,jz,jt
    integer :: kx,ky,kz,kt

#define U_mux U_xd(ix,iy,iz,it,mu)
#define U_nux U_xd(ix,iy,iz,it,nu)

#define U_muxpmu U_xd(jx,jy,jz,jt,mu)
#define U_nuxpmu U_xd(jx,jy,jz,jt,nu)

#define U_muxpnu U_xd(kx,ky,kz,kt,mu)
#define U_nuxpnu U_xd(kx,ky,kz,kt,nu)

    dmu = 0
    dnu = 0

    dmu(mu) = 1
    dnu(nu) = 1

    do it=1,nt
       jt = mapt(it + dmu(4))
       kt = mapt(it + dnu(4))
       do iz=1,nz
          jz = mapz(iz + dmu(3))
          kz = mapz(iz + dnu(3))
          do iy=1,ny
             jy = mapy(iy + dmu(2))
             ky = mapy(iy + dnu(2))
             do ix=1,nx
                jx = mapx(ix + dmu(1))
                kx = mapx(ix + dnu(1))

                call MultiplyMatMat(UmuUnu,U_mux,U_nuxpmu)
                call MultiplyMatDagMatDag(UmudagUnudag,U_muxpnu,U_nux)
                call RealTraceMultMatMat(plaq_x,UmuUnu,UmudagUnudag)
```

```
            plaqbar = plaqbar + plaq_x

          end do
        end do
      end do
    end do

  end subroutine AddPlaqBar_munu

end module GaugeField
```

FermionField: Provide the fundamental routines relating to fermion fields, including input and output.

```
module FermionField

  use ColourTypes
  use FermionFieldMPIComms
  use GaugeField
  use RealFieldMPIComms
  use ComplexFieldMPIComms
  implicit none

contains

  subroutine random_fermion_field(psi)

    type(colour_vector), dimension(:,:,:,:,:) :: psi
    integer :: ix,iy,iz,it,is,ns

    ns = size(psi,5)

    do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
       call random_vector(psi(ix,iy,iz,it,is))
    end do; end do; end do; end do; end do

  end subroutine random_fermion_field

  function inner_product(psi,phi) result (psidotphi)

    type(colour_vector), dimension(:,:,:,:,:) :: psi,phi

    complex(dc) :: psidotphi, psidotphipp
    integer :: ix,iy,iz,it,is,ns

    ns = size(psi,5)

    psidotphipp = 0.0d0
    do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
       psidotphipp = psidotphipp + sum(conjg(psi(ix,iy,iz,it,is)%Cl)*phi(ix,iy,iz,it,is)%Cl)
    end do; end do; end do; end do; end do

    call AllSumComplex(psidotphipp,psidotphi)

  end function inner_product

  function real_inner_product(psi,phi) result (psidotphi)

    type(colour_vector), dimension(:,:,:,:,:) :: psi,phi
    real(dp) :: psidotphi
    real(dp) :: psidotphipp
    integer :: ix,iy,iz,it,is,ns

    ns = size(psi,5)

    psidotphipp = 0.0d0
    do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
       psidotphipp = psidotphipp + sum(real(conjg(psi(ix,iy,iz,it,is)%Cl)*phi(ix,iy,iz,it,is)%Cl))
    end do; end do; end do; end do; end do

    call AllSumReal(psidotphipp,psidotphi)

  end function real_inner_product

  function fermion_norm(psi) result (norm_psi)

    type(colour_vector), dimension(:,:,:,:,:) :: psi
    real(dp) :: norm_psi
    real(dp) :: norm_psipp
    integer :: ix,iy,iz,it,is,ns

    ns = size(psi,5)

    norm_psipp = 0.0d0
    do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
       norm_psipp = norm_psipp + sum(real(conjg(psi(ix,iy,iz,it,is)%Cl)*psi(ix,iy,iz,it,is)%Cl))
```

```
    end do; end do; end do; end do; end do

    call AllSumReal(norm_psipp, norm_psi)
    norm_psi = sqrt(norm_psi)

  end function fermion_norm

  function fermion_normsq(psi) result (normsq_psi)

    type(colour_vector), dimension(:,:,:,:,:) :: psi
    real(dp) :: normsq_psi
    real(dp) :: normsq_psipp
    integer :: ix,iy,iz,it,is,ns

    ns = size(psi,5)

    normsq_psipp = 0.0d0
    do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
       normsq_psipp = normsq_psipp + sum(real(conjg(psi(ix,iy,iz,it,is)%Cl)*psi(ix,iy,iz,it,is)%Cl))
    end do; end do; end do; end do; end do

    call AllSumReal(normsq_psipp, normsq_psi)

  end function fermion_normsq

  subroutine normalise(phi, norm_phi)

    type(colour_vector), dimension(:,:,:,:,:) :: phi
    real(dp) :: norm_phi

    integer :: ix,iy,iz,it,is,ns

    ns = size(phi,5)

    do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
       phi(ix,iy,iz,it,is)%Cl = phi(ix,iy,iz,it,is)%Cl/norm_phi
    end do; end do; end do; end do; end do

  end subroutine normalise

  subroutine ProjectVector(v_lambda, phi)

    type(colour_vector), dimension(:,:,:,:,:) :: v_lambda, phi
    complex(dc) :: v_lambdadotphi
    integer :: ix,iy,iz,it,is,ns

    ns = size(phi,5)

    v_lambdadotphi = inner_product(v_lambda, phi)
    do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
       phi(ix,iy,iz,it,is)%Cl = phi(ix,iy,iz,it,is)%Cl - v_lambda(ix,iy,iz,it,is)%Cl*v_lambdadotphi
    end do; end do; end do; end do; end do

  end subroutine ProjectVector

  subroutine orthogonalise(phi, psi, psidotphi)

    type(colour_vector), dimension(:,:,:,:,:) :: phi, psi
    complex(dc) :: psidotphi
    integer :: ix,iy,iz,it,is,ns

    ns = size(phi,5)

    do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
       phi(ix,iy,iz,it,is)%Cl = phi(ix,iy,iz,it,is)%Cl - psi(ix,iy,iz,it,is)%Cl*psidotphi
    end do; end do; end do; end do; end do

  end subroutine orthogonalise

  subroutine ProjectVectorSpace(n_lambda, v_i, phi)

    integer, intent(in) :: n_lambda
    type(colour_vector), dimension(:,:,:,:,:) :: phi
    type(colour_vector), dimension(:,:,:,:,:,:), target :: v_i

    type(colour_vector), dimension(:,:,:,:,:), pointer :: v_lambda
    integer :: i_lambda

    do i_lambda = 1,n_lambda
       v_lambda => v_i(:,:,:,:,:,i_lambda)
       call ProjectVector(v_lambda, phi)
    end do

  end subroutine ProjectVectorSpace

  subroutine WriteFermionField(phi, OutFileId)

    type(colour_vector), dimension(:,:,:,:,:) :: phi
```

```
  integer :: OutFileId
  type(colour_vector), dimension(nx,ny,nz,nt,ns) :: chi
  complex(dc), dimension(:,:,:,:,:,:), allocatable :: chi_out
  integer :: irank, is, ic
  integer :: ix, jx, iy, jy, iz, jz, it, jt

  if ( i_am_root ) then
     allocate(chi_out(nlx,nly,nlz,nlt,ns,nc))

     do irank=0,mpi_size-1
        if ( irank == mpi_root_rank ) then
           chi = phi(1:nx,1:ny,1:nz,1:nt,:)
        else
           call RecvFermionField(chi,irank)
        end if

        call GetSubLattice(irank,ix,jx,iy,jy,iz,jz,it,jt)
        do is=1,ns; do ic=1,nc
           chi_out(ix:jx,iy:jy,iz:jz,it:jt,is,ic) = chi(:,:,:,:,is)%Cl(ic)
        end do; end do
     end do

     write(OutFileId) chi_out(:,:,:,:,:,:)

     deallocate(chi_out)
  else
     chi = phi(1:nx,1:ny,1:nz,1:nt,:)
     call SendFermionField(chi,mpi_root_rank)
  end if

end subroutine WriteFermionField

subroutine ReadFermionField(phi,InFileId)

  type(colour_vector), dimension(:,:,:,:,:) :: phi
  integer :: InFileId
  type(colour_vector), dimension(nx,ny,nz,nt,ns) :: chi
  complex(dc), dimension(:,:,:,:,:,:), allocatable :: chi_in
  integer :: it, irank, is, ic
  integer :: ix, jx, iy, jy, iz, jz, it, jt

  if ( i_am_root ) then
     allocate(chi_in(nlx,nly,nlz,nlt,ns,nc))

     read(InFileId) chi_in(:,:,:,:,:,:)

     do irank=0,mpi_size-1
        call GetSubLattice(irank,ix,jx,iy,jy,iz,jz,it,jt)
        do is=1,ns; do ic=1,nc
           chi(:,:,:,:,is)%Cl(ic) = chi_in(ix:jx,iy:jy,iz:jz,it:jt,is,ic)
        end do; end do

        if ( irank == mpi_root_rank ) then
           phi(1:nx,1:ny,1:nz,1:nt,:) = chi
        else
           call SendFermionField(chi,irank)
        end if
     end do

     deallocate(chi_in)

  else
     call RecvFermionField(chi,mpi_root_rank)
     phi(1:nx,1:ny,1:nz,1:nt,:) = chi
  end if

end subroutine ReadFermionField

subroutine WriteEigenSpace(n_ev,n_dummy,lambda_i,v_i,tolerance,m_f,bct,OutputFile)

  integer, intent(in) :: n_ev, n_dummy
  real(dp), dimension(n_ev) :: lambda_i
  type(colour_vector), dimension(nx,ny,nz,nt,ns,n_ev) :: v_i
  real(dp) :: tolerance, m_f, bct
  character(len=*) :: OutputFile

  integer :: it, i_ev, irank

  if ( i_am_root ) then
     open (100, file=OutputFile, status = "unknown", form = "unformatted", action = "write")
     write (100) n_ev, n_dummy, tolerance, m_f, bct
  end if

  do i_ev=1,n_ev
     if ( i_am_root ) then
        write(100) lambda_i(i_ev)
     end if
     call WriteFermionField(v_i(:,:,:,:,:,i_ev),100)
```

```
  end do

  if ( i_am_root ) then
     close(100)
  end if

  call MPIBarrier

end subroutine WriteEigenSpace

subroutine ReadEigenSpace(n_ev,n_dummy,lambda_i,v_i,tolerance,m_f,bct,InputFile)

  integer, intent(in) :: n_ev
  integer :: n_dummy
  real(dp), dimension(n_ev) :: lambda_i
  type(colour_vector), dimension(nx,ny,nz,nt,ns,n_ev) :: v_i
  real(dp) :: tolerance, m_f, bct
  character(len=*) :: InputFile

  integer :: it, i_ev, irank, m_ev

  if ( i_am_root ) then
     open (100, file=InputFile, status = "old", form = "unformatted", action = "read")
     read (100) m_ev, n_dummy, tolerance, m_f, bct
  end if

  do i_ev=1,n_ev
     if ( i_am_root ) then
        read(100) lambda_i(i_ev)
     end if
     call BroadCastReal(lambda_i(i_ev),mpi_root_rank)
     call ReadFermionField(v_i(:,:,:,:,:,i_ev),100)
  end do

  if ( i_am_root ) then
     close(100)
  end if

  call BroadCastInteger(n_dummy,mpi_root_rank)
  call BroadCastReal(tolerance,mpi_root_rank)
  call BroadCastReal(m_f,mpi_root_rank)
  call BroadCastReal(bct,mpi_root_rank)

end subroutine ReadEigenSpace

subroutine ReadFermionField_sc(phi,InFileId)

  complex(dc), dimension(nx*ny*nz,nt,ns,nc) :: phi
  integer :: InFileId

  complex(dc), dimension(nx*ny*nz,nt,ns,nc) :: chi
  complex(dc), dimension(:,:,:,:), allocatable :: chi_in
  integer :: irank
  integer :: ix, jx, iy, jy, iz, jz, it, jt
  integer :: kx,ky,kz,kt, i_xyz, k_xyz
  if ( i_am_root ) then
     allocate(chi_in(nlx*nly*nlz,nlt,ns,nc))

     read(InFileId) chi_in(:,:,:,:)

     do irank=0,mpi_size-1
        call GetSubLattice(irank,ix,jx,iy,jy,iz,jz,it,jt)
        do kt=1,nt; do kz=1,nz; do ky=1,ny; do kx=1,nx
           k_xyz =  kx + (ky-1)*nx + (kz-1)*nx*ny
           i_xyz = (ix+(kx-1)) + (iy-1+(ky-1))*nlx + (iz-1+(kz-1))*nlx*nly
           chi(k_xyz,kt,:,:) = chi_in(i_xyz,it+kt-1,:,:)
        end do; end do; end do; end do

        if ( irank == mpi_root_rank ) then
           phi = chi
        else
           call SendComplexField(chi,irank)
        end if
     end do

     deallocate(chi_in)

  else
     call RecvComplexField(chi,mpi_root_rank)
     phi = chi
  end if

end subroutine ReadFermionField_sc

subroutine WriteTimeSlicePropagator(phi,OutFileId)

  complex(dc), dimension(nx*ny*nz,nt,nc,ns,nc,ns) :: phi
  integer :: OutFileId
```

```fortran
  complex(dc), dimension(nx*ny*nz,nt,nc,ns,nc,ns) :: chi
  real(dp), dimension(:,:,:,:,:,:), allocatable :: phir, phii
  integer :: ix, jx, iy, jy, iz, jz, it, jt
  integer :: kx,ky,kz,kt, i_xyz, k_xyz
  integer :: irank

  if ( i_am_root ) then
     allocate(phir(nlx*nly*nlz,nlt,nc,ns,nc,ns))
     allocate(phii(nlx*nly*nlz,nlt,nc,ns,nc,ns))

     do irank=0,mpi_size-1
        if ( irank == mpi_root_rank ) then
           chi = phi
        else
           call RecvComplexField(chi,irank)
        end if
        call GetSubLattice(irank,ix,jx,iy,jy,iz,jz,it,jt)
        do kt=1,nt; do kz=1,nz; do ky=1,ny; do kx=1,nx
           k_xyz =  kx + (ky-1)*nx + (kz-1)*nx*ny
           i_xyz = (ix+(kx-1)) + (iy-1+(ky-1))*nlx + (iz-1+(kz-1))*nlx*nly
           phir(i_xyz,it+kt-1,:,:,:,:) =  real(chi(k_xyz,kt,:,:,:,:))
           phii(i_xyz,it+kt-1,:,:,:,:) = aimag(chi(k_xyz,kt,:,:,:,:))
        end do; end do; end do; end do

     end do

     do it=1,nlt
        write(OutFileld) phir(:,it,:,:,:,:)
        write(OutFileld) phii(:,it,:,:,:,:)
     end do

     deallocate(phir)
     deallocate(phii)
  else
     chi = phi
     call SendComplexField(chi,mpi_root_rank)
  end if

end subroutine WriteTimeSlicePropagator

pure subroutine GammaPlusProject(Gplus_muphi,phi,mu)

  type(colour_vector), dimension(ns), intent(out) :: Gplus_muphi
  type(colour_vector), dimension(ns), intent(in) :: phi
  integer, intent(in) :: mu

  !Apply the spinor projector Gplus = (1 + gamma_mu) to phi.
  !Naturally we are in a chiral basis.

  select case(mu)
  case(1)
     Gplus_muphi(1)%Cl = phi(1)%Cl - cmplx(-aimag(phi(4)%Cl),real(phi(4)%Cl),dc)
     Gplus_muphi(2)%Cl = phi(2)%Cl - cmplx(-aimag(phi(3)%Cl),real(phi(3)%Cl),dc)
     Gplus_muphi(3)%Cl = cmplx(-aimag(Gplus_muphi(2)%Cl),real(Gplus_muphi(2)%Cl),dc)
     Gplus_muphi(4)%Cl = cmplx(-aimag(Gplus_muphi(1)%Cl),real(Gplus_muphi(1)%Cl),dc)
  case(2)
     Gplus_muphi(1)%Cl = phi(1)%Cl - phi(4)%Cl
     Gplus_muphi(2)%Cl = phi(2)%Cl + phi(3)%Cl
     Gplus_muphi(3)%Cl =  Gplus_muphi(2)%Cl
     Gplus_muphi(4)%Cl = -Gplus_muphi(1)%Cl
  case(3)
     Gplus_muphi(1)%Cl = phi(1)%Cl - cmplx(-aimag(phi(3)%Cl),real(phi(3)%Cl),dc)
     Gplus_muphi(2)%Cl = phi(2)%Cl + cmplx(-aimag(phi(4)%Cl),real(phi(4)%Cl),dc)
     Gplus_muphi(3)%Cl = cmplx(-aimag(Gplus_muphi(1)%Cl),real(Gplus_muphi(1)%Cl),dc)
     Gplus_muphi(4)%Cl = cmplx(aimag(Gplus_muphi(2)%Cl),-real(Gplus_muphi(2)%Cl),dc)
  case(4)
     Gplus_muphi(1)%Cl = phi(1)%Cl + phi(3)%Cl
     Gplus_muphi(2)%Cl = phi(2)%Cl + phi(4)%Cl
     Gplus_muphi(3)%Cl = Gplus_muphi(1)%Cl
     Gplus_muphi(4)%Cl = Gplus_muphi(2)%Cl
  end select

end subroutine GammaPlusProject

pure subroutine GammaMinusProject(Gminus_muphi,phi,mu)

  type(colour_vector), dimension(ns), intent(out) :: Gminus_muphi
  type(colour_vector), dimension(ns), intent(in) :: phi
  integer, intent(in) :: mu

  !Apply the spinor projector (1 - gamma_mu) to phi.
  !Naturally we are in a chiral basis.

  select case(mu)
  case(1)
     Gminus_muphi(1)%Cl = phi(1)%Cl + cmplx(-aimag(phi(4)%Cl),real(phi(4)%Cl),dc)
     Gminus_muphi(2)%Cl = phi(2)%Cl + cmplx(-aimag(phi(3)%Cl),real(phi(3)%Cl),dc)
     Gminus_muphi(3)%Cl = cmplx(aimag(Gminus_muphi(2)%Cl),-real(Gminus_muphi(2)%Cl),dc)
```

```
      Gminus_muphi(4)%Cl = cmplx(aimag(Gminus_muphi(1)%Cl),-real(Gminus_muphi(1)%Cl),dc)
   case(2)
      Gminus_muphi(1)%Cl = phi(1)%Cl + phi(4)%Cl
      Gminus_muphi(2)%Cl = phi(2)%Cl - phi(3)%Cl
      Gminus_muphi(3)%Cl = -Gminus_muphi(2)%Cl
      Gminus_muphi(4)%Cl =  Gminus_muphi(1)%Cl
   case(3)
      Gminus_muphi(1)%Cl = phi(1)%Cl + cmplx(-aimag(phi(3)%Cl),real(phi(3)%Cl),dc)
      Gminus_muphi(2)%Cl = phi(2)%Cl - cmplx(-aimag(phi(4)%Cl),real(phi(4)%Cl),dc)
      Gminus_muphi(3)%Cl = cmplx(aimag(Gminus_muphi(1)%Cl),-real(Gminus_muphi(1)%Cl),dc)
      Gminus_muphi(4)%Cl = cmplx(-aimag(Gminus_muphi(2)%Cl),real(Gminus_muphi(2)%Cl),dc)
   case(4)
      Gminus_muphi(1)%Cl = phi(1)%Cl - phi(3)%Cl
      Gminus_muphi(2)%Cl = phi(2)%Cl - phi(4)%Cl
      Gminus_muphi(3)%Cl = -Gminus_muphi(1)%Cl
      Gminus_muphi(4)%Cl = -Gminus_muphi(2)%Cl
   end select

end subroutine GammaMinusProject

subroutine GammaPhi(gamma_muphi,phi,mu)

   type(colour_vector), dimension(ns) :: gamma_muphi
   type(colour_vector), dimension(ns) :: phi
   integer :: mu

   !Naturally we are in a chiral basis.

   select case(mu)
   case(1)
      gamma_muphi(1)%Cl = -cmplx(-aimag(phi(4)%Cl),real(phi(4)%Cl),dc)
      gamma_muphi(2)%Cl = -cmplx(-aimag(phi(3)%Cl),real(phi(3)%Cl),dc)
      gamma_muphi(3)%Cl =  cmplx(-aimag(phi(2)%Cl),real(phi(2)%Cl),dc)
      gamma_muphi(4)%Cl =  cmplx(-aimag(phi(1)%Cl),real(phi(1)%Cl),dc)
   case(2)
      gamma_muphi(1)%Cl = -phi(4)%Cl
      gamma_muphi(2)%Cl =  phi(3)%Cl
      gamma_muphi(3)%Cl =  phi(2)%Cl
      gamma_muphi(4)%Cl = -phi(1)%Cl
   case(3)
      gamma_muphi(1)%Cl = -cmplx(-aimag(phi(3)%Cl),real(phi(3)%Cl),dc)
      gamma_muphi(2)%Cl =  cmplx(-aimag(phi(4)%Cl),real(phi(4)%Cl),dc)
      gamma_muphi(3)%Cl =  cmplx(-aimag(phi(1)%Cl),real(phi(1)%Cl),dc)
      gamma_muphi(4)%Cl = -cmplx(-aimag(phi(2)%Cl),real(phi(2)%Cl),dc)
   case(4)
      gamma_muphi(1)%Cl =  phi(3)%Cl
      gamma_muphi(2)%Cl =  phi(4)%Cl
      gamma_muphi(3)%Cl =  phi(1)%Cl
      gamma_muphi(4)%Cl =  phi(2)%Cl
   end select

end subroutine GammaPhi

subroutine SigmaPhi(sigma_munuphi,phi,mu,nu)

   type(colour_vector), dimension(ns) :: sigma_munuphi
   type(colour_vector), dimension(ns) :: phi
   integer :: mu, nu, iplaq, is

   if ( mu < nu ) then
      iplaq = mu+nu-1-1/mu
   else
      iplaq = nu+mu-1-1/nu
   end if

   select case(iplaq)
   case(1) ! mu = 1, nu = 2
      sigma_munuphi(1)%Cl =  cmplx(-aimag(phi(1)%Cl),real(phi(1)%Cl),dc)
      sigma_munuphi(2)%Cl = -cmplx(-aimag(phi(2)%Cl),real(phi(2)%Cl),dc)
      sigma_munuphi(3)%Cl =  cmplx(-aimag(phi(3)%Cl),real(phi(3)%Cl),dc)
      sigma_munuphi(4)%Cl = -cmplx(-aimag(phi(4)%Cl),real(phi(4)%Cl),dc)
   case(2) ! mu = 1, nu = 3
      sigma_munuphi(1)%Cl = -phi(2)%Cl
      sigma_munuphi(2)%Cl =  phi(1)%Cl
      sigma_munuphi(3)%Cl = -phi(4)%Cl
      sigma_munuphi(4)%Cl =  phi(3)%Cl
   case(3) ! mu = 1, nu = 4
      sigma_munuphi(1)%Cl = -cmplx(-aimag(phi(2)%Cl),real(phi(2)%Cl),dc)
      sigma_munuphi(2)%Cl = -cmplx(-aimag(phi(1)%Cl),real(phi(1)%Cl),dc)
      sigma_munuphi(3)%Cl =  cmplx(-aimag(phi(4)%Cl),real(phi(4)%Cl),dc)
      sigma_munuphi(4)%Cl =  cmplx(-aimag(phi(3)%Cl),real(phi(3)%Cl),dc)
   case(4) ! mu = 2, nu = 3
      sigma_munuphi(1)%Cl =  cmplx(-aimag(phi(2)%Cl),real(phi(2)%Cl),dc)
      sigma_munuphi(2)%Cl =  cmplx(-aimag(phi(1)%Cl),real(phi(1)%Cl),dc)
      sigma_munuphi(3)%Cl =  cmplx(-aimag(phi(4)%Cl),real(phi(4)%Cl),dc)
      sigma_munuphi(4)%Cl =  cmplx(-aimag(phi(3)%Cl),real(phi(3)%Cl),dc)
   case(5) ! mu = 2, nu = 4
      sigma_munuphi(1)%Cl = -phi(2)%Cl
```

```
      sigma_munuphi(2)%Cl =  phi(1)%Cl
      sigma_munuphi(3)%Cl =  phi(4)%Cl
      sigma_munuphi(4)%Cl = -phi(3)%Cl
    case(6) ! mu = 3, nu = 4
      sigma_munuphi(1)%Cl = -cmplx(-aimag(phi(1)%Cl),real(phi(1)%Cl),dc)
      sigma_munuphi(2)%Cl =  cmplx(-aimag(phi(2)%Cl),real(phi(2)%Cl),dc)
      sigma_munuphi(3)%Cl =  cmplx(-aimag(phi(3)%Cl),real(phi(3)%Cl),dc)
      sigma_munuphi(4)%Cl = -cmplx(-aimag(phi(4)%Cl),real(phi(4)%Cl),dc)
    end select

    if ( mu > nu ) then
      do is=1,ns
        sigma_munuphi(is)%Cl = -sigma_munuphi(is)%Cl
      end do
    end if

  end subroutine SigmaPhi

  subroutine GetSource(rho,jx,jy,jz,jt,js,jc,SmearedSource,alpha_smear,n_smear,U_xd)

    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: rho
    integer :: jx,jy,jz,jt,js,jc
    logical :: SmearedSource
    real(dp) :: alpha_smear
    integer :: n_smear
    type(colour_matrix), dimension(:,:,:,:,:) :: U_xd

    ! Gauge invariant (spatial) fermion source smearing.
    ! ref: NPB(proc. suppl.) 17, 361 (1990), PRD47, 5128 (1993)

    type(colour_vector), dimension(nxs,nys,nzs) :: rho_pr
    type(colour_vector), dimension(nxp,nyp,nzp) :: phi

    type(colour_vector) :: phi_px, phi_mx

    integer :: mu,ismear
    integer :: ix,iy,iz,it,dmu(nd)
    integer :: lx,ly,lz
    integer :: mx,my,mz
    integer :: mpi_status(nmpi_status), mpierror, mrank, prank, tag

#define U_mux         U_xd(ix,iy,iz,it,mu)
#define U_muxmmu      U_xd(mx,my,mz,it,mu)

#define rho_x     rho_pr(ix,iy,iz)
#define rho_xpmu  rho_pr(lx,ly,lz)
#define rho_xmmu  rho_pr(mx,my,mz)

#define phi_x     phi(ix,iy,iz)

    if ( site_is_mine(jx,jy,jz,jt) ) then
      call LatticeToSubLattice(jx,jy,jz,jt,ix,iy,iz,it)
      rho(ix,iy,iz,it,js)%Cl(jc) = 1.0d0
    end if

    if ( SmearedSource .and. ( n_smear > 0) ) then

      if ( (i_nt <= jt) .and. (jt <= j_nt) ) then
        it = jt - i_nt + 1
        rho_pr(1:nx,1:ny,1:nz) = rho(1:nx,1:ny,1:nz,it,js)

        mrank = mpi_coords(4) + nproct*modulo(mpi_coords(3) - 1, nprocz)
        prank = mpi_coords(4) + nproct*modulo(mpi_coords(3) + 1, nprocz)

        tag = nx*ny

        do ismear=1,n_smear

          call MPI_SendRecv(rho_pr(1,1,1)           ,nc*nx*ny,mpi_dc,mrank,tag,&
               &            rho_pr(1,1,mapz(nz+1)),nc*nx*ny,mpi_dc,prank,tag,&
               &            mpi_comm, mpi_status, mpierror)

          call MPI_SendRecv(rho_pr(1,1,nz)          ,nc*nx*ny,mpi_dc,prank,tag,&
               &            rho_pr(1,1,mapz(0))    ,nc*nx*ny,mpi_dc,mrank,tag,&
               &            mpi_comm, mpi_status, mpierror)

          phi = zero_vector

          do mu=1,nd-1
            dmu = 0
            dmu(mu) = 1

            do iz=1,nz
              lz = mapz(iz + dmu(3))
              mz = mapz(iz - dmu(3))
              do iy=1,ny
                ly = mapy(iy + dmu(2))
                my = mapy(iy - dmu(2))
```

```
              do ix=1,nx
                 lx = mapx(ix + dmu(1))
                 mx = mapx(ix - dmu(1))

                 call MultiplyMatClrVec    (phi_px,U_mux   ,rho_xpmu)
                 call MultiplyMatDagClrVec(phi_mx,U_muxmmu,rho_xmmu)

                 phi_x%Cl = phi_x%Cl + (phi_px%Cl + phi_mx%Cl)/u0_bar

              end do
            end do
          end do

        end do

        do iz=1,nz; do iy=1,ny; do ix=1,nx
           rho_x%Cl = (1.0d0 - alpha_smear)*rho_x%Cl + (alpha_smear/6.0d0)*phi_x%Cl
        end do; end do; end do

      end do

      rho(1:nx,1:ny,1:nz,it,js) = rho_pr(1:nx,1:ny,1:nz)

    end if

  end if

 end subroutine GetSource

end module FermionField
```

GL3Diag: Parallel colour matrix diagonalisation routine, based on the Ritz algorithm.

```
module GL3Diag

  use ColourTypes
  use VectorAlgebra
  implicit none

  !Routines for the 3x3 matrices (General linear group)

contains

  subroutine DiagonaliseMat(H_xd,lambda_xd,V_xd)

    type(colour_matrix) :: H_xd !Hermitian matrix
    type(real_vector)   :: lambda_xd !eigenvalues
    type(colour_matrix) :: V_xd !eigenvector matrix
    integer :: mu

    type(colour_vector) :: psi, Dpsi, grad_mu, search, Dsearch
    real(dp) :: normsq_grad_mu, normsq_grad_mu_old
    real(dp) :: norm_search, norm_psi

    logical :: converged, recalc
    real(dp)  :: tolerance = 1.0d-12
    integer :: iterations, mycount

    complex(dc) :: psidotsearch, grad_mudotsearch
    real(dp) :: psidotDpsi, searchdotDsearch

    real(dp) :: cos_delta, sin_delta, cos_theta, sin_theta
    real(dp) :: a2, a3

    real(dp) :: alpha, beta, mu_psi, mu_phi, mu_chi
    complex(dc) :: phidotDchi, chidotDphi

    type(colour_vector) :: phi, chi
#define Dphi grad_mu
#define Dchi search

    complex(dc) :: phidotv_p, chidotv_p, phidotv_m, chidotv_m

    real(dp) :: lambda_p, lambda_m
    complex(dc) :: nu_p, nu_m

    integer :: t_in, t_out
    integer :: ix,iy,iz,it

    psi%Cl = 1.0d0/sqrt(3.0d0)

    call normalise_vector(psi)

    call MultiplyMatClrVec(Dpsi,H_xd,psi)
```

```
mu_psi = real_vector_inner_product(psi,Dpsi)

grad_mu%CI = Dpsi%CI - mu_psi*psi%CI

normsq_grad_mu = vector_normsq(grad_mu)

search = grad_mu
norm_search = vector_norm(search)

call MultiplyMatClrVec(Dsearch,H_xd,search)

converged = .false.
recalc = .false.
iterations = 0

psidotsearch = vector_inner_product(psi,search)

do
   converged = ( sqrt(normsq_grad_mu) < tolerance ) .or. ( norm_search**2 < tolerance**2 )

   if ( converged ) exit
   norm_psi = vector_norm(psi)
   if ( iterations >= 60 ) print *, "Diag Mat ", iterations
   if ( iterations >= 60 ) exit

   iterations = iterations + 1
   recalc = ( modulo(iterations,3) == 0 )

   normsq_grad_mu_old = normsq_grad_mu
   psidotDpsi = real_vector_inner_product(psi,Dpsi)
   searchdotDsearch = real_vector_inner_product(search,Dsearch)/(norm_search**2)

   a2 = 0.5d0*(psidotDpsi - searchdotDsearch)
   a3 = normsq_grad_mu/norm_search
   alpha = sqrt( a2**2 + a3**2)

   cos_delta = a2/alpha
   sin_delta = a3/alpha

   if (cos_delta <= 0.0d0 ) then
      cos_theta = sqrt(0.5d0*(1.0d0-cos_delta))
      sin_theta = -0.5d0*sin_delta/cos_theta
   else
      sin_theta = -sqrt(0.5d0*(1.0d0+cos_delta))
      cos_theta = -0.5d0*sin_delta/sin_theta
   end if

   psi%CI = cos_theta*psi%CI + sin_theta*(search%CI/norm_search)

   if ( recalc ) then

      call normalise_vector(psi)
      call MultiplyMatClrVec(Dpsi,H_xd,psi)

   else

      Dpsi%CI = cos_theta*Dpsi%CI + sin_theta*(Dsearch%CI/norm_search)

   end if

   mu_psi =  real_vector_inner_product(psi,Dpsi)

   grad_mu%CI = Dpsi%CI - mu_psi*psi%CI

   normsq_grad_mu = vector_normsq(grad_mu)

   converged = ( sqrt(normsq_grad_mu) < tolerance )
   if ( converged ) exit

   beta = cos_theta*(normsq_grad_mu/normsq_grad_mu_old)

   if ( recalc ) beta = 0.0d0

   psidotsearch = vector_inner_product(psi,search)
   search%CI = grad_mu%CI + beta*( search%CI - psidotsearch*psi%CI )

   if ( recalc ) then
      psidotsearch = vector_inner_product(psi,search)

      Dsearch%CI = search%CI-grad_mu%CI

      grad_mudotsearch = vector_inner_product(grad_mu,Dsearch)
      grad_mudotsearch = grad_mudotsearch/normsq_grad_mu

      search%CI = search%CI - psidotsearch*psi%CI - grad_mudotsearch*grad_mu%CI
   end if

   norm_search = vector_norm(search)
```

```
        call MultiplyMatClrVec(Dsearch,H_xd,search)

     end do

     call normalise_vector(psi)

     call MultiplyMatClrVec(Dpsi,H_xd,psi)
     mu_psi = real_vector_inner_product(psi,Dpsi)

     phi%Cl = 1.0d0/sqrt(3.0d0)

     call orthogonalise_vectors(phi,psi)
     call normalise_vector(phi)

     call vector_product(chi,psi,phi)
     call normalise_vector(chi)

     call MultiplyMatClrVec(Dphi,H_xd,phi)
     call orthogonalise_vectors(Dphi,psi)

     call MultiplyMatClrVec(Dchi,H_xd,chi)
     call orthogonalise_vectors(Dchi,psi)

     mu_phi = real_vector_inner_product(phi,Dphi)
     mu_chi = real_vector_inner_product(chi,Dchi)
     phidotDchi = vector_inner_product(phi,Dchi)
     chidotDphi = vector_inner_product(chi,Dphi)

     beta = -(mu_phi+mu_chi)
     alpha = mu_phi*mu_chi - real(phidotDchi*chidotDphi)

     lambda_p = 0.5d0*(-beta - sqrt(beta**2 - 4.0d0*alpha))
     lambda_m = 0.5d0*(-beta + sqrt(beta**2 - 4.0d0*alpha))

     nu_p = -phidotDchi/(mu_phi - lambda_p)

     alpha = sqrt(real(nu_p)**2 + aimag(nu_p)**2 + 1.0d0)
     phidotv_p = nu_p/alpha
     chidotv_p = 1.0d0/alpha

     nu_m = -chidotDphi/(mu_chi - lambda_m)

     alpha = sqrt(real(nu_m)**2 + aimag(nu_m)**2 + 1.0d0)

     phidotv_m = 1.0d0/alpha
     chidotv_m = nu_m/alpha

     !Temporarily store the eigenvectors rotated from phi and chi
     Dphi%Cl = phi%Cl*phidotv_p + chi%Cl*chidotv_p
     Dchi%Cl = phi%Cl*phidotv_m + chi%Cl*chidotv_m

     phi = Dphi
     chi = Dchi

     call normalise_vector(phi)
     call normalise_vector(chi)

     call MultiplyMatClrVec(Dphi,H_xd,phi)
     call MultiplyMatClrVec(Dchi,H_xd,chi)

     mu_phi = real_vector_inner_product(phi,Dphi)
     mu_chi = real_vector_inner_product(chi,Dchi)

     lambda_xd%Cl(1) = mu_psi
     V_xd%Cl(:,1) = psi%Cl(:)

     lambda_xd%Cl(2) = mu_phi
     V_xd%Cl(:,2) = phi%Cl(:)

     lambda_xd%Cl(3) = mu_chi
     V_xd%Cl(:,3) = chi%Cl(:)

#undef Dphi
#undef Dchi

   end subroutine DiagonaliseMat

end module GL3Diag
```

FatLinks: Implementation of APE smearing, with unit circle projection.

```
module FatLinks

   use GaugeField
   use GL3Diag
   implicit none
```

```
  type(colour_matrix), dimension(nxs,nys,nzs,nts,nd) :: UFL_xd !Fat links

  real(DP) :: alpha_smear = 0.7d0 !smearing fraction
  integer :: ape_sweeps = 4 !smearing sweeps
  real(dp) :: u0fl_bar = 1.0d0, uzerofl = 1.0d0

contains

  subroutine APESmearLinks(U_xd, UFL_xd, alpha, sweeps)

    type(colour_matrix), dimension(:,:,:,:,:) :: U_xd
    type(colour_matrix), dimension(:,:,:,:,:) :: UFL_xd

    real(dp) :: alpha
    integer :: sweeps

    integer :: isweeps

    UFL_xd(1:nxs,1:nys,1:nzs,1:nts,:) = U_xd(1:nxs,1:nys,1:nzs,1:nts,:)

    do isweeps =1,sweeps
       call APESmear(UFL_xd,alpha)
       if ( isweeps == sweeps) call FixSU3(UFL_xd)
       call ShadowGaugeField(UFL_xd,1)
    end do

  end subroutine APESmearLinks

  subroutine APESmear(U_xd,alpha)

    type(colour_matrix), dimension(:,:,:,:,:) :: U_xd
    real(dp) :: alpha

    type(colour_matrix), dimension(nxp,nyp,nzp,ntp,nd) :: V_xd !Smeared links
    integer :: ix,iy,iz,it,mu,nu,inu

    double precision,dimension(nx,ny,nz,nt,nd,nc,nc) :: urprm,uiprm
    double precision,dimension(nx,ny,nz,nt,nd,nc,nc) :: ur,ui

    call GetSmearedLinks(V_xd,U_xd,alpha)

    U_xd(1:nx,1:ny,1:nz,1:nt,:) = V_xd(1:nx,1:ny,1:nz,1:nt,:)

    call SU3Project(U_xd)

  end subroutine APESmear

  subroutine GetSmearedLinks(V_xd,U_xd,alpha)

    type(colour_matrix), dimension(:,:,:,:,:) :: V_xd
    type(colour_matrix), dimension(:,:,:,:,:) :: U_xd
    real(dp) :: alpha

    integer :: ix,iy,iz,it,mu,nu,inu

    do mu=1,nd
       do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
          V_xd(ix,iy,iz,it,mu)%Cl = 0.0d0
       end do; end do; end do; end do
    end do

    do mu=1,nd
       nu = mu
       do inu=1,nd-1
          nu = modulo(nu,nd)+1
          call GetAPEBlockedLinks(V_xd,U_xd,mu,nu)
       end do
    end do

    do mu=1,nd
       do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
          V_xd(ix,iy,iz,it,mu)%Cl = (1.0d0 - alpha)*U_xd(ix,iy,iz,it,mu)%Cl + &
               & (alpha/6.0d0)*V_xd(ix,iy,iz,it,mu)%Cl
       end do; end do; end do; end do
    end do

  end subroutine GetSmearedLinks

  subroutine SU3Project(U_xd)

    type(colour_matrix), dimension(:,:,:,:,:) :: U_xd
    type(colour_matrix) :: H_xd, V_xd, W_xd

    type(real_vector) :: lambda_xd !eigenvalues

    type(colour_vector) :: WxW !cross product
    complex(dc) :: detW
```

```
    integer :: ix,iy,iz,it,mu,ic,jc

    do mu=1,nd

       do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx

          call MultiplyMatDagMat(H_xd,U_xd(ix,iy,iz,it,mu),U_xd(ix,iy,iz,it,mu))
          call DiagonaliseMat(H_xd,lambda_xd,V_xd)

          lambda_xd%Cl = 1.0d0/sqrt(lambda_xd%Cl)

          call MultiplyMatRealDiagMat(W_xd,V_xd,lambda_xd)

          !Undiagonalise H_xd back to the U_xd basis
          call MultiplyMatMatdag(H_xd,W_xd,V_xd)
          call MultiplyMatMat(W_xd,U_xd(ix,iy,iz,it,mu),H_xd)

          WxW%Cl(1) = W_xd%Cl(2,2)*W_xd%Cl(3,3) - W_xd%Cl(2,3)*W_xd%Cl(3,2)
          WxW%Cl(2) = W_xd%Cl(2,3)*W_xd%Cl(3,1) - W_xd%Cl(2,1)*W_xd%Cl(3,3)
          WxW%Cl(3) = W_xd%Cl(2,1)*W_xd%Cl(3,2) - W_xd%Cl(2,2)*W_xd%Cl(3,1)

          detW = W_xd%Cl(1,1)*WxW%Cl(1) + W_xd%Cl(1,2)*WxW%Cl(2) + W_xd%Cl(1,3)*WxW%Cl(3)

          detW = 1.0d0/(detW**(1.0d0/3.0d0))

          U_xd(ix,iy,iz,it,mu)%Cl(:,1) = detW*W_xd%Cl(:,1)
          U_xd(ix,iy,iz,it,mu)%Cl(:,2) = detW*W_xd%Cl(:,2)
          U_xd(ix,iy,iz,it,mu)%Cl(:,3) = detW*W_xd%Cl(:,3)

       end do; end do; end do; end do

    end do

  end subroutine SU3Project


  subroutine GetAPEBlockedLinks(V_xd,U_xd,mu,nu)

    type(colour_matrix), dimension(:,:,:,:,:) :: V_xd
    type(colour_matrix), dimension(:,:,:,:,:) :: U_xd
    integer :: mu, nu

    type(colour_matrix) :: UnuUmu, UnudagUmu, V_nup, V_num
    integer, dimension(nd) :: dmu, dnu
    integer :: ix,iy,iz,it
    integer :: jx,jy,jz,jt
    integer :: kx,ky,kz,kt
    integer :: lx,ly,lz,lt
    integer :: mx,my,mz,mt

#define U_nux U_xd(ix,iy,iz,it,nu)
#define V_munux V_xd(ix,iy,iz,it,mu)
#define U_nuxpmu     U_xd(jx,jy,jz,jt,nu)
#define U_nuxmnu     U_xd(lx,ly,lz,lt,nu)
#define U_muxpnu     U_xd(kx,ky,kz,kt,mu)
#define U_muxmnu     U_xd(lx,ly,lz,lt,mu)
#define U_nuxmnupmu  U_xd(mx,my,mz,mt,nu)


    dmu = 0
    dnu = 0

    dmu(mu) = 1
    dnu(nu) = 1

    do it=1,nt
       jt = mapt(it + dmu(4))
       kt = mapt(it + dnu(4))
       lt = mapt(it - dnu(4))
       mt = mapt(it - dnu(4) + dmu(4))
       do iz=1,nz
          jz = mapz(iz + dmu(3))
          kz = mapz(iz + dnu(3))
          lz = mapz(iz - dnu(3))
          mz = mapz(iz - dnu(3) + dmu(3))
          do iy=1,ny
             jy = mapy(iy + dmu(2))
             ky = mapy(iy + dnu(2))
             ly = mapy(iy - dnu(2))
             my = mapy(iy - dnu(2) + dmu(2))
             do ix=1,nx
                jx = mapx(ix + dmu(1))
                kx = mapx(ix + dnu(1))
                lx = mapx(ix - dnu(1))
                mx = mapx(ix - dnu(1) + dmu(1))

                call MultiplyMatMat(UnuUmu,U_nux,U_muxpnu)
                call MultiplyMatMatdag(V_nup,UnuUmu,U_nuxpmu)
```

```
              call  MultiplyMatdagMat (UnudagUmu , U_nuxmnu , U_muxmnu )
              call  MultiplyMatMat (V_num , UnudagUmu , U_nuxmnupmu )

              V_munux%Cl = V_munux%Cl + V_nup%Cl + V_num%Cl

           end do
        end do
     enddo
  end do

#undef U_nux
#undef V_munux
#undef U_nuxpmu
#undef U_nuxmnu
#undef U_muxpnu
#undef U_muxmnu
#undef U_nuxmnupmu

  end subroutine GetAPEBlockedLinks

end module FatLinks
```

CloverFmunu: Implementation of the one loop clover term.

```
module CloverFmunu

  use MatrixAlgebra
  use MPIInterface
  implicit none

  type(colour_matrix), dimension(nxp,nyp,nzp,ntp,nplaq) :: F_munu !Clover based field strength tensor

contains

  function GetTopQ(F_munu) result (Q)

    type(colour_matrix), dimension(:,:,:,:,:) :: F_munu
    real(dp) :: Q, Qpp, TrF12F34, TrF13F24, TrF14F23
    integer :: ix,iy,iz,it

    Qpp = 0.0d0

    do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
       call RealTraceMultMatMat(TrF12F34,F_munu(ix,iy,iz,it,1),F_munu(ix,iy,iz,it,6))
       call RealTraceMultMatMat(TrF13F24,F_munu(ix,iy,iz,it,2),F_munu(ix,iy,iz,it,5))
       call RealTraceMultMatMat(TrF14F23,F_munu(ix,iy,iz,it,3),F_munu(ix,iy,iz,it,4))
       Qpp = Qpp + 8.0d0*(TrF12F34 - TrF13F24 + TrF14F23)
    end do; end do; end do; end do

    call AllSumReal(Qpp,Q)

    Q = Q/(32.0d0*(pi**2))

  end function GetTopQ

  subroutine CalculateFmunuClover(F_munu,U_xd)

    type(colour_matrix), dimension(:,:,:,:,:) :: F_munu
    type(colour_matrix), dimension(:,:,:,:,:) :: U_xd

    integer :: mu, nu, iplaq
    integer :: ix,iy,iz,it

    do mu=1,nd
       do nu=mu+1,nd
          iplaq = mu+nu-1-1/mu

          do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
             F_munu(ix,iy,iz,it,iplaq)%Cl = 0.0d0
          end do; end do; end do; end do

          call GetCloverTerm(F_munu,U_xd,mu,nu,iplaq)

          do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
             F_munu(ix,iy,iz,it,iplaq)%Cl = 0.125d0*F_munu(ix,iy,iz,it,iplaq)%Cl
          end do; end do; end do; end do

       end do
    end do

  end subroutine CalculateFmunuClover


  subroutine GetCloverTerm(F_munu,U_xd,mu,nu,iplaq)

    type(colour_matrix), dimension(:,:,:,:,:) :: F_munu
```

```
    type(colour_matrix), dimension(:,:,:,:,:) :: U_xd
    integer :: mu, nu, iplaq

    type(colour_matrix) :: U_munu, UmuUnu, UmudagUnudag, UnuUmudag, UnudagUmu !Gauge field products

    integer, dimension(nd) :: dmu, dnu
    integer :: ix,iy,iz,it
    integer :: jx,jy,jz,jt
    integer :: kx,ky,kz,kt
    integer :: ax,ay,az,at
    integer :: bx,by,bz,bt
    integer :: cx,cy,cz,ct
    integer :: dx,dy,dz,dt
    integer :: ex,ey,ez,et

#define U_mux U_xd(ix,iy,iz,it,mu)
#define U_nux U_xd(ix,iy,iz,it,nu)

#define U_nuxpmu U_xd(jx,jy,jz,jt,nu)
#define U_muxpnu U_xd(kx,ky,kz,kt,mu)

#define U_muxmmu U_xd(ax,ay,az,at,mu)
#define U_nuxmmu U_xd(ax,ay,az,at,nu)

#define U_muxmnu U_xd(bx,by,bz,bt,mu)
#define U_nuxmnu U_xd(bx,by,bz,bt,nu)

#define U_nuxmmumnu U_xd(cx,cy,cz,ct,nu)
#define U_muxmmumnu U_xd(cx,cy,cz,ct,mu)

#define U_nuxmnupmu U_xd(dx,dy,dz,dt,nu)
#define U_muxpnummu U_xd(ex,ey,ez,et,mu)

#define F_munux F_munu(ix,iy,iz,it,iplaq)

    dmu = 0
    dnu = 0
    dmu(mu) = 1
    dnu(nu) = 1

    do it=1,nt
       jt = mapt(it + dmu(4))
       kt = mapt(it + dnu(4))
       at = mapt(it - dmu(4))
       bt = mapt(it - dnu(4))
       ct = mapt(it - dmu(4) - dnu(4))
       dt = mapt(it + dmu(4) - dnu(4))
       et = mapt(it - dmu(4) + dnu(4))
       do iz=1,nz
          jz = mapz(iz + dmu(3))
          kz = mapz(iz + dnu(3))
          az = mapz(iz - dmu(3))
          bz = mapz(iz - dnu(3))
          cz = mapz(iz - dmu(3) - dnu(3))
          dz = mapz(iz + dmu(3) - dnu(3))
          ez = mapz(iz - dmu(3) + dnu(3))
          do iy=1,ny
             jy = mapy(iy + dmu(2))
             ky = mapy(iy + dnu(2))
             ay = mapy(iy - dmu(2))
             by = mapy(iy - dnu(2))
             cy = mapy(iy - dmu(2) - dnu(2))
             dy = mapy(iy + dmu(2) - dnu(2))
             ey = mapy(iy - dmu(2) + dnu(2))
             do ix=1,nx
                jx = mapx(ix + dmu(1))
                kx = mapx(ix + dnu(1))
                ax = mapx(ix - dmu(1))
                bx = mapx(ix - dnu(1))
                cx = mapx(ix - dmu(1) - dnu(1))
                dx = mapx(ix + dmu(1) - dnu(1))
                ex = mapx(ix - dmu(1) + dnu(1))

                !U+mu+nu(x)
                call MultiplyMatMat(UmuUnu,U_mux,U_nuxpmu)
                call MultiplyMatdagMatdag(UmudagUnudag,U_muxpnu,U_nux)
                call MultiplyMatMat(U_munu,UmuUnu,UmudagUnudag)

                call AddCloverLeaf(F_munux,U_munu)

                !U-nu+mu(x+a_nu)
                call MultiplyMatdagMat(UnudagUmu,U_nuxmnu,U_muxmnu)
                call MultiplyMatMatdag(UnuUmudag,U_nuxmnupmu,U_mux)
                call MultiplyMatMat(U_munu,UnudagUmu,UnuUmudag)

                call AddCloverLeaf(F_munux,U_munu)

                !U+nu-mu(x+a_mu)
```

```
                call MultiplyMatMatdag(UnuUmudag, U_nux, U_muxpnummu)
                call MultiplyMatdagMat(UnudagUmu, U_nuxmmu, U_muxmmu)
                call MultiplyMatMat(U_munu, UnuUmudag, UnudagUmu)

                call AddCloverLeaf(F_munux, U_munu)

                !U-mu-nu(x+a_mu+a_nu)
                call MultiplyMatdagMatdag(UmudagUnudag, U_muxmmu, U_nuxmmumnu)
                call MultiplyMatMat(UmuUnu, U_muxmmumnu, U_nuxmnu)
                call MultiplyMatMat(U_munu, UmudagUnudag, UmuUnu)

                call AddCloverLeaf(F_munux, U_munu)

             end do
          end do
       enddo
    end do

#undef U_mux
#undef U_nux
#undef U_nuxpmu
#undef U_muxpnu

#undef F_munux

  contains

    subroutine AddCloverLeaf(F_munux, U_munux)

      type(colour_matrix) :: F_munux, U_munux

      F_munux%Cl(1,1) = F_munux%Cl(1,1) + U_munux%Cl(1,1) - conjg(U_munux%Cl(1,1))
      F_munux%Cl(2,1) = F_munux%Cl(2,1) + U_munux%Cl(2,1) - conjg(U_munux%Cl(1,2))
      F_munux%Cl(3,1) = F_munux%Cl(3,1) + U_munux%Cl(3,1) - conjg(U_munux%Cl(1,3))
      F_munux%Cl(1,2) = F_munux%Cl(1,2) + U_munux%Cl(1,2) - conjg(U_munux%Cl(2,1))
      F_munux%Cl(2,2) = F_munux%Cl(2,2) + U_munux%Cl(2,2) - conjg(U_munux%Cl(2,2))
      F_munux%Cl(3,2) = F_munux%Cl(3,2) + U_munux%Cl(3,2) - conjg(U_munux%Cl(2,3))
      F_munux%Cl(1,3) = F_munux%Cl(1,3) + U_munux%Cl(1,3) - conjg(U_munux%Cl(3,1))
      F_munux%Cl(2,3) = F_munux%Cl(2,3) + U_munux%Cl(2,3) - conjg(U_munux%Cl(3,2))
      F_munux%Cl(3,3) = F_munux%Cl(3,3) + U_munux%Cl(3,3) - conjg(U_munux%Cl(3,3))

    end subroutine AddCloverLeaf

  end subroutine GetCloverTerm

end module CloverFmunu
```

ChiralFLICOperator: Optimised implementation of $H_{\rm flic}$ in the chiral γ_μ basis.

```
module ChiralFLICOperator

  use Timer
  use MPIInterface
  use FatLinks
  use CloverFmunu
  use FermionField
  implicit none

  !shifted gauge fields
  type(colour_matrix), dimension(nxp,nyp,nzp,ntp,nd) :: Up_xd
  type(colour_matrix), dimension(nxp,nyp,nzp,ntp,nd) :: Um_xd

  !MFI Fat Link Clover-Wilson fermions using negative mass term, in Sakurai basis
  integer :: ncalls = 0
  real(dp) :: maxtime = 0, mintime
  real(dp) :: meantime = 0.0
  real(dp) :: m_f, bct = 1.0d0, c_sw = 1.0d0
  real(dp), private :: m_r

  complex(dc), dimension(nc,nx,ny,nz,ns) :: phi_st, phi_rt
  complex(dc), dimension(nc,nx,ny,nt,ns) :: phi_sz, phi_rz

contains

  subroutine InitialiseFLICOperator(U_xd, UFL_xd, u0, u0fl, mass, c_sw, bct)

    type(colour_matrix), dimension(:,:,:,:,:) :: U_xd, UFL_xd

    real(DP) :: mass, c_sw, u0, u0fl, bct
    integer :: ix,iy,iz,it,mu, ic,jc
    real(dp) :: mfic_sw, mfir, mfir_fl !coefficents to be absorbed.

    type(colour_matrix) :: U_mu, Ufl_mu ! temporary storage
    type(colour_matrix), dimension(nplaq) :: F_x
```

```fortran
    m_f = mass

    Up_xd(1:nx,1:ny,1:nz,1:nt,:) = U_xd(1:nx,1:ny,1:nz,1:nt,:)
    Um_xd(1:nx,1:ny,1:nz,1:nt,:) = UFL_xd(1:nx,1:ny,1:nz,1:nt,:)

    if (bct/=1.0d0) then
       call SetBoundaryConditions(UFL_xd,1.0d0,1.0d0,1.0d0,bct)
       call ShadowGaugeField(UFL_xd,1)
    end if

    call CalculateFmunuClover(F_munu,UFL_xd)

    mfic_sw = 0.5d0*c_sw/(u0fl**4)

    !code is in the chiral basis so we only need these sums and differences. (halve multiplies)

    do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx

       F_x(1) = F_munu(ix,iy,iz,it,1)
       F_x(2) = F_munu(ix,iy,iz,it,2)
       F_x(3) = F_munu(ix,iy,iz,it,3)
       F_x(4) = F_munu(ix,iy,iz,it,4)
       F_x(5) = F_munu(ix,iy,iz,it,5)
       F_x(6) = F_munu(ix,iy,iz,it,6)

       F_munu(ix,iy,iz,it,1)%Cl = mfic_sw*(F_x(1)%Cl - F_x(6)%Cl)
       F_munu(ix,iy,iz,it,2)%Cl = mfic_sw*(F_x(2)%Cl + F_x(5)%Cl)
       F_munu(ix,iy,iz,it,3)%Cl = mfic_sw*(F_x(3)%Cl - F_x(4)%Cl)
       F_munu(ix,iy,iz,it,4)%Cl = mfic_sw*(F_x(1)%Cl + F_x(6)%Cl)
       F_munu(ix,iy,iz,it,5)%Cl = mfic_sw*(F_x(2)%Cl - F_x(5)%Cl)
       F_munu(ix,iy,iz,it,6)%Cl = mfic_sw*(F_x(3)%Cl + F_x(4)%Cl)

    end do; end do; end do; end do

    mfir = (0.25d0/u0)
    mfir_fl = (0.25d0/u0fl)

    !Absorb the mean field improvement into the gauge fields.
    !Define symmetrised and antisymmetrised gauge fields for efficiency (halve multiplies)

    do mu=1,nd
       do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx

          U_mu%Cl = mfir*Up_xd(ix,iy,iz,it,mu)%Cl
          Ufl_mu%Cl = mfir_fl*Um_xd(ix,iy,iz,it,mu)%Cl

          Up_xd(ix,iy,iz,it,mu)%Cl = U_mu%Cl + Ufl_mu%Cl

          Um_xd(ix,iy,iz,it,mu)%Cl = U_mu%Cl - Ufl_mu%Cl

       end do; end do; end do; end do
    end do

    if ( bct/=1.0d0 ) then
       call SetBoundaryConditions(Up_xd,1.0d0,1.0d0,1.0d0,bct)
       call SetBoundaryConditions(Um_xd,1.0d0,1.0d0,1.0d0,bct)
    end if

    call ShadowGaugeField(Up_xd,0)
    call ShadowGaugeField(Um_xd,0)

  end subroutine InitialiseFLICOperator

  subroutine FLICOperate(phi, Dphi)

    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: Dphi

    integer :: ix,iy,iz,it,is
    integer :: jx,jy,jz,jt,js
    complex(dc) :: psi_x
    type(colour_vector) :: Gammaphi

    real(dp) :: pm(ns),intime, outtime
    integer :: mrank,prank,send_reqz,recv_reqz,send_reqt,recv_reqt,mpi_status(nmpi_status), mpierror

    if (timing) intime = mpi_wtime()

#define phi_x(is) phi(ix,iy,iz,it,is)
#define Dphi_x(is) Dphi(ix,iy,iz,it,is)
#define F_x(ip) F_munu(ix,iy,iz,it,ip)

    m_r = 4.0d0 - m_f

    mrank = mpi_coords(4) + nproct*modulo(mpi_coords(3) - 1, nprocz)
    prank = mpi_coords(4) + nproct*modulo(mpi_coords(3) + 1, nprocz)
```

```
phi_sz(1,:,:,:,:) = phi(:,:,1,1:nt,:)%Cl(1)
phi_sz(2,:,:,:,:) = phi(:,:,1,1:nt,:)%Cl(2)
phi_sz(3,:,:,:,:) = phi(:,:,1,1:nt,:)%Cl(3)

call MPI_ISSend(phi_sz,nc*nx*ny*nt*ns, mpi_dc, mrank, nx*ny*nt, mpi_comm, send_reqz, mpierror)
call MPI_IRecv (phi_rz,nc*nx*ny*nt*ns, mpi_dc, prank, nx*ny*nt, mpi_comm, recv_reqz, mpierror)


pm(1) = -1.0d0
pm(2) =  1.0d0

do is=1,nsp
   do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx

      ! *unroll*
      psi_x = F_x(1)%Cl(ic,1)*cmplx(-aimag(phi_x(is)%Cl(1)),real(phi_x(is)%Cl(1)),dc) + &
            & F_x(1)%Cl(ic,2)*cmplx(-aimag(phi_x(is)%Cl(2)),real(phi_x(is)%Cl(2)),dc) + &
            & F_x(1)%Cl(ic,3)*cmplx(-aimag(phi_x(is)%Cl(3)),real(phi_x(is)%Cl(3)),dc)
      Dphi_x(is)%Cl(ic) = m_r*phi_x(is)%Cl(ic) + pm(is)*psi_x

   end do; end do; end do; end do
end do

pm(1) =  1.0d0
pm(2) = -1.0d0

do is=1,nsp
   js = 3-is
   do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx

      ! *unroll*
      psi_x = F_x(2)%Cl(ic,1)*phi_x(is)%Cl(1) + &
            & F_x(2)%Cl(ic,2)*phi_x(is)%Cl(2) + &
            & F_x(2)%Cl(ic,3)*phi_x(is)%Cl(3)
      Dphi_x(js)%Cl(ic) = Dphi_x(js)%Cl(ic) + pm(js)*psi_x

   end do; end do; end do; end do
end do

do is=1,nsp
   js = 3-is
   do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx

      ! *unroll*
      psi_x = F_x(3)%Cl(ic,1)*cmplx(aimag(phi_x(is)%Cl(1)),-real(phi_x(is)%Cl(1)),dc) + &
            & F_x(3)%Cl(ic,2)*cmplx(aimag(phi_x(is)%Cl(2)),-real(phi_x(is)%Cl(2)),dc) + &
            & F_x(3)%Cl(ic,3)*cmplx(aimag(phi_x(is)%Cl(3)),-real(phi_x(is)%Cl(3)),dc)
      Dphi_x(js)%Cl(ic) = Dphi_x(js)%Cl(ic) - psi_x

   end do; end do; end do; end do
end do

mrank = modulo(mpi_coords(4) - 1, nproct) + nproct*mpi_coords(3)
prank = modulo(mpi_coords(4) + 1, nproct) + nproct*mpi_coords(3)

phi_st(1,:,:,:,:) = phi(:,:,1:nz,1,:)%Cl(1)
phi_st(2,:,:,:,:) = phi(:,:,1:nz,1,:)%Cl(2)
phi_st(3,:,:,:,:) = phi(:,:,1:nz,1,:)%Cl(3)

call MPI_ISSend(phi_st, nc*nx*ny*nz*ns, mpi_dc, mrank, nx*ny*nz, mpi_comm, send_reqt, mpierror)
call MPI_IRecv (phi_rt, nc*nx*ny*nz*ns, mpi_dc, prank, nx*ny*nz, mpi_comm, recv_reqt, mpierror)

pm(3) = -1.0d0
pm(4) =  1.0d0

do is=3,ns
   do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx

      ! *unroll*
      psi_x = F_x(4)%Cl(ic,1)*cmplx(-aimag(phi_x(is)%Cl(1)),real(phi_x(is)%Cl(1)),dc) + &
            & F_x(4)%Cl(ic,2)*cmplx(-aimag(phi_x(is)%Cl(2)),real(phi_x(is)%Cl(2)),dc) + &
            & F_x(4)%Cl(ic,3)*cmplx(-aimag(phi_x(is)%Cl(3)),real(phi_x(is)%Cl(3)),dc)
      Dphi_x(is)%Cl(ic) = m_r*phi_x(is)%Cl(ic) + pm(is)*psi_x

   end do; end do; end do; end do
end do

pm(3) =  1.0d0
pm(4) = -1.0d0

do is=3,ns
   js=5-is/2
   do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx

      ! *unroll*
      psi_x = F_x(5)%Cl(ic,1)*phi_x(is)%Cl(1) + &
            & F_x(5)%Cl(ic,2)*phi_x(is)%Cl(2) + &
            & F_x(5)%Cl(ic,3)*phi_x(is)%Cl(3)
```

```
            Dphi_x(js)%Cl(ic) = Dphi_x(js)%Cl(ic) + pm(js)*psi_x

         end do; end do; end do; end do
      end do

      do is=3,ns
         js=5-is/2
         do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx


            ! *unroll*
            psi_x = F_x(6)%Cl(ic,1)*cmplx(aimag(phi_x(is)%Cl(1)),-real(phi_x(is)%Cl(1)),dc) + &
                 &  F_x(6)%Cl(ic,2)*cmplx(aimag(phi_x(is)%Cl(2)),-real(phi_x(is)%Cl(2)),dc) + &
                 &  F_x(6)%Cl(ic,3)*cmplx(aimag(phi_x(is)%Cl(3)),-real(phi_x(is)%Cl(3)),dc)
            Dphi_x(js)%Cl(ic) = Dphi_x(js)%Cl(ic) + psi_x

         end do; end do; end do; end do
      end do

      !mu = 1

#define phi_xpmu(is)  phi(jx,iy,iz,it,is)
#define Um_mux      Um_xd(ix,iy,iz,it,1)
#define Up_mux      Up_xd(ix,iy,iz,it,1)

      !G_1^+ Um phi_xpmu

      do is=1,nsp
         js = 5-is
         do it=1,nt; do iz=1,nz; do iy=1,ny; ix=nx; do jx=1,nx

            ! *unroll*
            Gammaphi%Cl(ic) = phi_xpmu(is)%Cl(ic) - cmplx(-aimag(phi_xpmu(js)%Cl(ic)),real(phi_xpmu(js)%Cl(
                 ic)),dc)

            psi_x = sum(Um_mux%Cl(ic,:)*Gammaphi%Cl(:))

            Dphi_x(is)%Cl(ic) = Dphi_x(is)%Cl(ic) + psi_x
            Dphi_x(js)%Cl(ic) = Dphi_x(js)%Cl(ic) + cmplx(-aimag(psi_x),real(psi_x),dc)

            ix = jx

         end do; end do; end do; end do
      end do
      !G_1^- Up phi_xpmu

      do is=1,nsp
         js = 5-is
         do it=1,nt; do iz=1,nz; do iy=1,ny; ix=nx; do jx=1,nx

            ! *unroll*
            Gammaphi%Cl(ic) = phi_xpmu(is)%Cl(ic) + cmplx(-aimag(phi_xpmu(js)%Cl(ic)),real(phi_xpmu(js)%Cl(
                 ic)),dc)

            psi_x = sum(Up_mux%Cl(ic,:)*Gammaphi%Cl(:))

            Dphi_x(is)%Cl(ic) = Dphi_x(is)%Cl(ic) - psi_x
            Dphi_x(js)%Cl(ic) = Dphi_x(js)%Cl(ic) + cmplx(-aimag(psi_x),real(psi_x),dc)

            ix = jx

         end do; end do; end do; end do
      end do

      !mu = 2

#define phi_xpmu(is)  phi(ix,jy,iz,it,is)
#define Um_mux      Um_xd(ix,iy,iz,it,2)
#define Up_mux      Up_xd(ix,iy,iz,it,2)

      !G_2^+ Um phi_xpmu

      pm(1) =  1.0d0
      pm(2) = -1.0d0
      pm(3) =  1.0d0
      pm(4) = -1.0d0

      do is=1,nsp
         js = 5-is
         do it=1,nt; do iz=1,nz; iy=ny; do jy=1,ny; do ix=1,nx

            ! *unroll*
            Gammaphi%Cl(ic) = phi_xpmu(is)%Cl(ic) - pm(is)*phi_xpmu(js)%Cl(ic)

            psi_x = sum(Um_mux%Cl(ic,:)*Gammaphi%Cl(:))

            Dphi_x(is)%Cl(ic) = Dphi_x(is)%Cl(ic) + psi_x
```

```
            Dphi_x(js)%Cl(ic) = Dphi_x(js)%Cl(ic) + pm(js)*psi_x

         end do; iy=jy; end do; end do; end do
      end do

      !G_2^- Up phi_xpmu

      do is=1,nsp
         js = 5-is
         do it=1,nt; do iz=1,nz; iy=ny; do jy=1,ny; do ix=1,nx

            ! *unroll*
            Gammaphi%Cl(ic) = phi_xpmu(is)%Cl(ic) + pm(is)*phi_xpmu(js)%Cl(ic)

            psi_x = sum(Up_mux%Cl(ic,:)*Gammaphi%Cl(:))

            Dphi_x(is)%Cl(ic) = Dphi_x(is)%Cl(ic) - psi_x
            Dphi_x(js)%Cl(ic) = Dphi_x(js)%Cl(ic) + pm(js)*psi_x

         end do; iy=jy; end do; end do; end do
      end do

      call MPI_Wait(recv_reqz,mpi_status,mpierror)
      call MPI_Wait(send_reqz,mpi_status,mpierror)

      phi(:,:,nz+1,1:nt,:)%Cl(1) = phi_rz(1,:,:,:,:)
      phi(:,:,nz+1,1:nt,:)%Cl(2) = phi_rz(2,:,:,:,:)
      phi(:,:,nz+1,1:nt,:)%Cl(3) = phi_rz(3,:,:,:,:)

      !mu = 3

#define phi_xpmu(is)  phi(ix,iy,jz,it,is)
#define Um_mux     Um_xd(ix,iy,iz,it,3)
#define Up_mux     Up_xd(ix,iy,iz,it,3)

      !G_3^+ Um phi_xpmu

      pm(1) =  1.0d0
      pm(2) = -1.0d0
      pm(3) =  1.0d0
      pm(4) = -1.0d0

      do is=1,nsp
         js = is+2
         do it=1,nt; do iz=1,nz; jz=iz+1; do iy=1,ny; do ix=1,nx

            ! *unroll*
            Gammaphi%Cl(ic) = phi_xpmu(is)%Cl(ic) - pm(is)*cmplx(-aimag(phi_xpmu(js)%Cl(ic)),real(phi_xpmu(
                 js)%Cl(ic)),dc)

            psi_x = sum(Um_mux%Cl(ic,:)*Gammaphi%Cl(:))

            Dphi_x(is)%Cl(ic) = Dphi_x(is)%Cl(ic) + psi_x
            Dphi_x(js)%Cl(ic) = Dphi_x(js)%Cl(ic) + pm(js)*cmplx(-aimag(psi_x),real(psi_x),dc)

         end do; end do; end do; end do
      end do

      !G_3^- Up phi_xpmu

      do is=1,nsp
         js = is+2
         do it=1,nt; do iz=1,nz; jz=iz+1; do iy=1,ny; do ix=1,nx

            ! *unroll*
            Gammaphi%Cl(ic) = phi_xpmu(is)%Cl(ic) + pm(is)*cmplx(-aimag(phi_xpmu(js)%Cl(ic)),real(phi_xpmu(
                 js)%Cl(ic)),dc)

            psi_x = sum(Up_mux%Cl(ic,:)*Gammaphi%Cl(:))

            Dphi_x(is)%Cl(ic) = Dphi_x(is)%Cl(ic) - psi_x
            Dphi_x(js)%Cl(ic) = Dphi_x(js)%Cl(ic) + pm(js)*cmplx(-aimag(psi_x),real(psi_x),dc)

         end do; end do; end do; end do
      end do

      mrank = mpi_coords(4) + nproct*modulo(mpi_coords(3) - 1, nprocz) + nproct*nprocz*mpi_coords(2)
      prank = mpi_coords(4) + nproct*modulo(mpi_coords(3) + 1, nprocz) + nproct*nprocz*mpi_coords(2)

      phi_sz(1,:,:,:,:) = phi(:,:,nz,1:nt,:)%Cl(1)
      phi_sz(2,:,:,:,:) = phi(:,:,nz,1:nt,:)%Cl(2)
      phi_sz(3,:,:,:,:) = phi(:,:,nz,1:nt,:)%Cl(3)

      call MPI_ISSend(phi_sz,nc*nx*ny*nt*ns, mpi_dc, prank, nx*ny*nt, mpi_comm, send_reqz, mpierror)
      call MPI_IRecv (phi_rz,nc*nx*ny*nt*ns, mpi_dc, mrank, nx*ny*nt, mpi_comm, recv_reqz, mpierror)

      call MPI_Wait(recv_reqt,mpi_status,mpierror)
      call MPI_Wait(send_reqt,mpi_status,mpierror)
```

```
      phi(:,:,1:nz,nt+1,:)%Cl(1) = phi_rt(1,:,:,:,:)
      phi(:,:,1:nz,nt+1,:)%Cl(2) = phi_rt(2,:,:,:,:)
      phi(:,:,1:nz,nt+1,:)%Cl(3) = phi_rt(3,:,:,:,:)

      !mu = 4

#define phi_xpmu(is)  phi(ix,iy,iz,jt,is)
#define Um_mux      Um_xd(ix,iy,iz,it,4)
#define Up_mux      Up_xd(ix,iy,iz,it,4)

      !G_4^+ Um phi_xpmu

      do is=1,nsp
         js = is+2
         do it=1,nt; jt=it+1; do iz=1,nz; do iy=1,ny; do ix=1,nx

            ! *unroll*
            Gammaphi%Cl(ic) = phi_xpmu(is)%Cl(ic) + phi_xpmu(js)%Cl(ic)

            psi_x = sum(Um_mux%Cl(ic,:)*Gammaphi%Cl(:))

            Dphi_x(is)%Cl(ic) = Dphi_x(is)%Cl(ic) + psi_x
            Dphi_x(js)%Cl(ic) = Dphi_x(js)%Cl(ic) + psi_x

         end do; end do; end do; end do
      end do

      !G_4^- Up phi_xpmu

      do is=1,nsp
         js = is+2
         do it=1,nt; jt=it+1; do iz=1,nz; do iy=1,ny; do ix=1,nx

            ! *unroll*
            Gammaphi%Cl(ic) = phi_xpmu(is)%Cl(ic) - phi_xpmu(js)%Cl(ic)

            psi_x = sum(Up_mux%Cl(ic,:)*Gammaphi%Cl(:))

            Dphi_x(is)%Cl(ic) = Dphi_x(is)%Cl(ic) - psi_x
            Dphi_x(js)%Cl(ic) = Dphi_x(js)%Cl(ic) + psi_x

         end do; end do; end do; end do
      end do

      mrank = modulo(mpi_coords(4) - 1, nproct) + nproct*mpi_coords(3)
      prank = modulo(mpi_coords(4) + 1, nproct) + nproct*mpi_coords(3)

      phi_st(1,:,:,:,:) = phi(:,:,1:nz,nt,:)%Cl(1)
      phi_st(2,:,:,:,:) = phi(:,:,1:nz,nt,:)%Cl(2)
      phi_st(3,:,:,:,:) = phi(:,:,1:nz,nt,:)%Cl(3)

      call MPI_ISSend(phi_st, nc*nx*ny*nz*ns, mpi_dc, prank, nx*ny*nz, mpi_comm, send_reqt, mpierror)
      call MPI_IRecv (phi_rt, nc*nx*ny*nz*ns, mpi_dc, mrank, nx*ny*nz, mpi_comm, recv_reqt, mpierror)

      !mu = 1

#define phi_xmmu(is)  phi(jx,iy,iz,it,is)
#define Um_muxmmu Um_xd(jx,iy,iz,it,1)
#define Up_muxmmu Up_xd(jx,iy,iz,it,1)

      !G_1^- Um phi_xmmu

      do is=1,nsp
         js = 5-is
         do it=1,nt; do iz=1,nz; do iy=1,ny; jx=nx; do ix=1,nx

            ! *unroll*
            Gammaphi%Cl(ic) = phi_xmmu(is)%Cl(ic) + cmplx(-aimag(phi_xmmu(js)%Cl(ic)),real(phi_xmmu(js)%Cl(
                 ic)),dc)

            psi_x = sum(conjg(Um_muxmmu%Cl(ic,:))*Gammaphi%Cl(:))

            Dphi_x(is)%Cl(ic) = Dphi_x(is)%Cl(ic) + psi_x
            Dphi_x(js)%Cl(ic) = Dphi_x(js)%Cl(ic) - cmplx(-aimag(psi_x),real(psi_x),dc)

            jx = ix

         end do; end do; end do; end do
      end do

      !G_1^+ Up phi_xmmu

      do is=1,nsp
         js = 5-is
         do it=1,nt; do iz=1,nz; do iy=1,ny; jx=nx; do ix=1,nx

            ! *unroll*
```

```
          Gammaphi%Cl(ic) = phi_xmmu(is)%Cl(ic) - cmplx(-aimag(phi_xmmu(js)%Cl(ic)),real(phi_xmmu(js)%Cl(
                ic)),dc)

          psi_x = sum(conjg(Up_muxmmu%Cl(ic,:))*Gammaphi%Cl(:))

          Dphi_x(is)%Cl(ic) = Dphi_x(is)%Cl(ic) - psi_x
          Dphi_x(js)%Cl(ic) = Dphi_x(js)%Cl(ic) - cmplx(-aimag(psi_x),real(psi_x),dc)

          jx = ix

       end do; end do; end do; end do
    end do

    !mu = 2

#define phi_xmmu(is)  phi(ix,jy,iz,it,is)
#define Um_muxmmu Um_xd(ix,jy,iz,it,2)
#define Up_muxmmu Up_xd(ix,jy,iz,it,2)

    !G_2^- Um phi_xmmu

    do is=1,nsp
       js = 5-is
       do it=1,nt; do iz=1,nz; jy=ny; do iy=1,ny; do ix=1,nx

          ! *unroll*
          Gammaphi%Cl(ic) = phi_xmmu(is)%Cl(ic) + pm(is)*phi_xmmu(js)%Cl(ic)

          psi_x = sum(conjg(Um_muxmmu%Cl(ic,:))*Gammaphi%Cl(:))

          Dphi_x(is)%Cl(ic) = Dphi_x(is)%Cl(ic) + psi_x
          Dphi_x(js)%Cl(ic) = Dphi_x(js)%Cl(ic) - pm(js)*psi_x

       end do; jy=iy; end do; end do; end do
    end do

    !G_2^+ Up phi_xmmu

    do is=1,nsp
       js = 5-is
       do it=1,nt; do iz=1,nz; jy=ny; do iy=1,ny; do ix=1,nx

          ! *unroll*
          Gammaphi%Cl(ic) = phi_xmmu(is)%Cl(ic) - pm(is)*phi_xmmu(js)%Cl(ic)

          psi_x = sum(conjg(Up_muxmmu%Cl(ic,:))*Gammaphi%Cl(:))

          Dphi_x(is)%Cl(ic) = Dphi_x(is)%Cl(ic) - psi_x
          Dphi_x(js)%Cl(ic) = Dphi_x(js)%Cl(ic) - pm(js)*psi_x

       end do; jy=iy; end do; end do; end do
    end do

    call MPI_Wait(recv_reqz,mpi_status,mpierror)
    call MPI_Wait(send_reqz,mpi_status,mpierror)

    phi(:,:,nz+1,1:nt,:)%Cl(1) = phi_rz(1,:,:,:,:)
    phi(:,:,nz+1,1:nt,:)%Cl(2) = phi_rz(2,:,:,:,:)
    phi(:,:,nz+1,1:nt,:)%Cl(3) = phi_rz(3,:,:,:,:)

    !mu = 3

#define phi_xmmu(is)  phi(ix,iy,jz,it,is)
#define Um_muxmmu Um_xd(ix,iy,jz,it,3)
#define Up_muxmmu Up_xd(ix,iy,jz,it,3)

    !G_3^- Um phi_xmmu

    do is=1,nsp
       js = is+2
       do it=1,nt; jz=nz+1; do iz=1,nz; do iy=1,ny; do ix=1,nx

          ! *unroll*
          Gammaphi%Cl(ic) = phi_xmmu(is)%Cl(ic) + pm(is)*cmplx(-aimag(phi_xmmu(js)%Cl(ic)),real(phi_xmmu(
                js)%Cl(ic)),dc)

          psi_x = sum(conjg(Um_muxmmu%Cl(ic,:))*Gammaphi%Cl(:))

          Dphi_x(is)%Cl(ic) = Dphi_x(is)%Cl(ic) + psi_x
          Dphi_x(js)%Cl(ic) = Dphi_x(js)%Cl(ic) - pm(js)*cmplx(-aimag(psi_x),real(psi_x),dc)

       end do; end do; jz=iz; end do; end do
    end do

    !G_3^+ Up phi_xmmu

    do is=1,nsp
       js = is+2
```

```
    do it=1,nt; jz=nz+1; do iz=1,nz; do iy=1,ny; do ix=1,nx

       ! *unroll*
       Gammaphi%Cl(ic) = phi_xmmu(is)%Cl(ic) - pm(is)*cmplx(-aimag(phi_xmmu(js)%Cl(ic)),real(phi_xmmu(
            js)%Cl(ic)),dc)

       psi_x = sum(conjg(Up_muxmmu%Cl(ic,:))*Gammaphi%Cl(:))

       Dphi_x(is)%Cl(ic) = Dphi_x(is)%Cl(ic) - psi_x
       Dphi_x(js)%Cl(ic) = Dphi_x(js)%Cl(ic) - pm(js)*cmplx(-aimag(psi_x),real(psi_x),dc)

    end do; end do; jz=iz; end do; end do
 end do

 call MPI_Wait(recv_reqt, mpi_status, mpierror)
 call MPI_Wait(send_reqt, mpi_status, mpierror)

 phi(:,:,1:nz,nt+1,:)%Cl(1) = phi_rt(1,:,:,:,:)
 phi(:,:,1:nz,nt+1,:)%Cl(2) = phi_rt(2,:,:,:,:)
 phi(:,:,1:nz,nt+1,:)%Cl(3) = phi_rt(3,:,:,:,:)

 !mu = 4

#define phi_xmmu(is)  phi(ix,iy,iz,jt,is)
#define Um_muxmmu Um_xd(ix,iy,iz,jt,4)
#define Up_muxmmu Up_xd(ix,iy,iz,jt,4)

 !G_4^- Um phi_xmmu

 do is=1,nsp
    js = is+2
    jt=nt+1
    do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx

       ! *unroll*
       Gammaphi%Cl(ic) = phi_xmmu(is)%Cl(ic) - phi_xmmu(js)%Cl(ic)

       psi_x = sum(conjg(Um_muxmmu%Cl(1,:))*Gammaphi%Cl(:))
       Dphi_x(is)%Cl(ic) = Dphi_x(is)%Cl(ic) + psi_x
       Dphi_x(js)%Cl(ic) = Dphi_x(js)%Cl(ic) - psi_x

    end do; end do; end do; jt=it; end do
 end do

 !G_4^+ Up phi_xmmu

 do is=1,nsp
    js = is+2
    jt=nt+1
    do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx

       ! *unroll*
       Gammaphi%Cl(ic) = phi_xmmu(is)%Cl(ic) + phi_xmmu(js)%Cl(ic)

       psi_x = sum(conjg(Up_muxmmu%Cl(1,:))*Gammaphi%Cl(:))

       Dphi_x(is)%Cl(ic) = Dphi_x(is)%Cl(ic) - psi_x
       Dphi_x(js)%Cl(ic) = Dphi_x(js)%Cl(ic) - psi_x

    end do; end do; end do; jt=it; end do
 end do

 !Multiply by gamma_5
 do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
    Dphi_x(3)%Cl = -Dphi_x(3)%Cl
    Dphi_x(4)%Cl = -Dphi_x(4)%Cl
 end do; end do; end do; end do


#undef phi_xmmu
#undef phi_xpmu
#undef Um_mux
#undef Um_muxmmu
#undef Up_mux
#undef Up_muxmmu

  if (timing) then
     outtime = mpi_wtime()
     call TimingUpdate(ncalls,outtime,intime,mintime,maxtime,meantime)
  end if

end subroutine FLICOperate

subroutine SqFLICOperate(phi,Dphi)

  type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi
  type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: Dphi
```

```
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: Dsqphi

    call FLICOperate(phi, Dphi)

    call FLICOperate(Dphi, Dsqphi)

    Dphi = Dsqphi

  end subroutine SqFLICOperate

  subroutine Dflic(phi,Dphi)

    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: Dphi
    integer :: ix,iy,iz,it

    call FLICOperate(phi,Dphi)

    do it=1,nt; do iz=1,nz; do iy=1,ny; do ix = 1,nx
       Dphi_x(3)%Cl = -Dphi_x(3)%Cl
       Dphi_x(4)%Cl = -Dphi_x(4)%Cl
    end do; end do; end do; end do

  end subroutine Dflic

  subroutine Dflicdag(phi,Dphi)

    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: Dphi
    integer :: ix,iy,iz,it

    do it=1,nt; do iz=1,nz; do iy=1,ny; do ix = 1,nx
       phi_x(3)%Cl = -phi_x(3)%Cl
       phi_x(4)%Cl = -phi_x(4)%Cl
    end do; end do; end do; end do

    call FLICOperate(phi,Dphi)

    do it=1,nt; do iz=1,nz; do iy=1,ny; do ix = 1,nx
       phi_x(3)%Cl = -phi_x(3)%Cl
       phi_x(4)%Cl = -phi_x(4)%Cl
    end do; end do; end do; end do

  end subroutine Dflicdag

#undef Dphi_x
#undef phi_x

end module ChiralFLICOperator
```

A.4 Overlap Code

ZolotarevApprox: Defines the coefficients used in the Zolotarev approximation to the sign function.

```
module ZolotarevApprox

  use ColourTypes
  implicit none

  integer :: n_poles

  real(dp) :: x_min, x_max
  real(dp) :: c_2n, d_n
  real(dp), dimension(:), allocatable :: b_l, c_l

  integer :: n_lambda !number of eigenvectors to project out

  real(dp), dimension(:), allocatable :: epsilon_lambda_i !eigenvalues
  type(colour_vector), dimension(:,:,:,:,:,:), allocatable :: v_lambda !eigenvectors to project out
  real(DP) :: delta_lambda !the precision to which the eigenvalues are known

end module ZolotarevApprox
```

SignFunction: Implements the multi-shift conjugate gradient solver, as applied to the Zolotarev sign function approximation.

```
module SignFunction
```

```
  use ColourTypes
  use Timer
  use FermionField
  use ZolotarevApprox
  use ConjGradSolvers
  use ChiralFLICOperator
  implicit none

  logical, parameter :: debug = .false.

  integer :: ncalls1 = 0
  real(dp) :: maxtime1 = 0, mintime1
  real(dp) :: meantime1 = 0.0

  logical :: project_eigenspace = .true.

  real(dp) :: c_1

contains

#define chi_x(is) chi(ix,iy,iz,it,is)
#define phi_x(is) phi(ix,iy,iz,it,is)
#define psi_x(is) psi(ix,iy,iz,it,is)
#define Mphi_x(is) Mphi(ix,iy,iz,it,is)
#define Dphi_x(is) Dphi(ix,iy,iz,it,is)
#define phi_ix(is) phi_i(ix,iy,iz,it,is,i_v)
#define residue_x(is) residue(ix,iy,iz,it,is)
#define chi_ix(is) chi_i(ix,iy,iz,it,is,i_v)
#define xi_x(is) xi(ix,iy,iz,it,is)
#define eta_x(is) eta(ix,iy,iz,it,is)

  subroutine MatrixSignFunctionOperate(n_v, psi, chi, tolerance, iterations_i, MatrixOperate,
       SqMatrixOperate, &
       & n_lambda, lambda_i, v_lambda, delta_lambda)

    !A routine to solve the matrix equation (M+sigma)v = psi for a system of shifts sigma
    !sigma must be ordered such that sigma(1) contains the least convergent system
    ! (i.e. the system that converges slowest).

    !M _must_ be hermitian and positive definite.
    !Implements the sign function through the Zolotarev rational approximation

    integer, intent(in) :: n_v !number of shifts
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: psi !source vector
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: chi !solution vector
    real(DP) :: tolerance !the precision desired for the solution
    integer, dimension(n_v) :: iterations_i !convergence information
    interface
       subroutine MatrixOperate(phi, Dphi)
         use ColourTypes
         type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi
         type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: Dphi
       end subroutine MatrixOperate
    end interface
    interface
       subroutine SqMatrixOperate(phi, Dphi)
         use ColourTypes
         type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi
         type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: Dphi
       end subroutine SqMatrixOperate
    end interface

    integer, intent(in) :: n_lambda !number of eigenvectors to project out of M
    real(DP), dimension(n_lambda) :: lambda_i !The sign function acting on the eigenvalues.
    type(colour_vector), dimension(nx,ny,nz,nt,ns,n_lambda), target :: v_lambda !eigenvectors to project
         out of M
    type(colour_vector), dimension(:,:,:,:,:), pointer :: v_l
    real(DP) :: delta_lambda !The precision to which the eigenvectors are correct.
    logical :: ProjectEV
    integer :: ProjectFreq=10, i_lambda

    integer :: iterations !the total number of iterations required
    real(DP), dimension(n_v) :: sigma_i !offset matrix shifts
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: residue
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi, Mphi
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns,n_v) :: phi_i
    real(DP) :: normsq_r, prevnormsq_r, beta, prevbeta, alpha, phidotMphi, offset, tau = 1.0d-1, normpsi,
         normphi
    real(DP), dimension(n_v) :: alpha_i, beta_i, prevbeta_i, zeta_i, prevzeta_i, nextzeta_i
    logical, dimension(n_v) :: converged_i, prevconverged_i
    real(dp) :: intime, outtime
    complex(dc) :: v_ldotphi
    integer :: ix,iy,iz,it,is, i_v

    ProjectEV = project_eigenspace

    if ( ProjectEV ) ProjectFreq = 10 - log10(delta_lambda)
```

```
!translate the sigma shifts so that sigma_i(1)=0

offset = c_l(1)
if (offset /= 0.0d0) then
   sigma_i = c_l - offset
else
   sigma_i = c_l
end if

residue = psi

chi = zero_vector

if ( ProjectEV ) call ProjectVectorSpace(n_lambda,v_lambda,residue)

do i_v=1,n_v
   phi_i(:,:,:,:,i_v) = residue
end do

prevbeta = 1.0d0
prevzeta_i = 1.0d0
zeta_i= 1.0d0
alpha_i = 0

normsq_r = fermion_normsq(residue)
normpsi = sqrt(normsq_r)

if (normsq_r == 0.0d0) return !source was contained in projected orthogonal space

phi = residue

call SqMatrixOperate(phi,Mphi)
if ( ProjectEV ) call ProjectVectorSpace(n_lambda,v_lambda,Mphi)
if (offset /= 0) then
   do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
      Mphi_x(is)%Cl = Mphi_x(is)%Cl + offset*phi_x(is)%Cl
   end do; end do; end do; end do; end do
end if

phidotMphi = real_inner_product(phi,Mphi)
if (phidotMphi == 0.0d0) return  !source was contained in projected orthogonal space

iterations = 0
iterations_i = 0
converged_i = .false.

do
   if (timing) intime = mpi_wtime()

   !If the slowest convergence is extremely slow compared to the other shifts, we need to have a
   ! "safety net" that prevents underflow in zeta_i for the already converged values.
   prevconverged_i = converged_i
   converged_i = (sqrt(normsq_r)*zeta_i/normpsi < tau*tolerance )

   do i_v=1,n_v
      if (converged_i(i_v)) zeta_i(i_v) = epsilon(1.0d0)
      if (converged_i(i_v) .and. .not. prevconverged_i(i_v)) iterations_i(i_v) = iterations
   end do

   if (sqrt(normsq_r)/normpsi < tolerance) iterations_i(1) = iterations
   if (sqrt(normsq_r)/normpsi < tolerance) exit

   iterations = iterations + 1
   ProjectEV = ( modulo(iterations,ProjectFreq)==0 ) .and. project_eigenspace

   beta = -normsq_r/phidotMphi

   nextzeta_i = zeta_i*prevzeta_i*prevbeta/( beta*alpha*(prevzeta_i-zeta_i) + &
        & prevzeta_i*prevbeta*(1.0d0-sigma_i*beta) )
   beta_i = beta*nextzeta_i/zeta_i

   do i_v=1,n_v
      if (.not. converged_i(i_v)) then
         do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
            chi_x(is)%Cl = chi_x(Is)%Cl - (b_l(i_v)*beta_i(i_v))*phi_ix(is)%Cl
         end do; end do; end do; end do; end do
      end if
   end do
   if ( ProjectEV ) call ProjectVectorSpace(n_lambda,v_lambda,chi)

   do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
      residue_x(is)%Cl = residue_x(is)%Cl + beta*Mphi_x(is)%Cl
   end do; end do; end do; end do; end do
   if ( ProjectEV ) call ProjectVectorSpace(n_lambda,v_lambda,residue)

   prevnormsq_r = normsq_r
   normsq_r = fermion_normsq(residue)
```

```
        alpha = normsq_r/prevnormsq_r
        alpha_i = alpha*nextzeta_i*beta_i/(zeta_i*beta)

        do i_v=1,n_v
           if (.not. converged_i(i_v) ) then
              do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
                 phi_ix(is)%Cl = nextzeta_i(i_v)*residue_x(is)%Cl + alpha_i(i_v)*phi_ix(is)%Cl
              end do; end do; end do; end do; end do
           end if
        end do

        phi = phi_i(:,:,:,:,:,1)
        call SqMatrixOperate(phi,Mphi)
        if (offset /= 0) then
           do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
              Mphi_x(is)%Cl = Mphi_x(is)%Cl + offset*phi_x(is)%Cl
           end do; end do; end do; end do; end do
        end if
        if ( ProjectEV ) call ProjectVectorSpace(n_lambda,v_lambda,Mphi)
        phidotMphi = real_inner_product(phi,Mphi)

        prevzeta_i = zeta_i
        zeta_i = nextzeta_i

        prevbeta = beta

        prevbeta_i = beta_i

        if (timing) then
           outtime = mpi_wtime()
           call TimingUpdate(ncalls1,outtime,intime,mintime1,maxtime1,meantime1)
        end if

     end do

     ProjectEV = project_eigenspace

     call SqMatrixOperate(chi,Mphi)
     do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
        chi_x(is)%Cl = Mphi_x(is)%Cl + c_2n*chi_x(is)%Cl
     end do; end do; end do; end do; end do
     if ( ProjectEV ) call ProjectVectorSpace(n_lambda,v_lambda,chi)

     call MatrixOperate(chi,Mphi)
     do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
        chi_x(is)%Cl = d_n*Mphi_x(is)%Cl
     end do; end do; end do; end do; end do
     if ( ProjectEV ) call ProjectVectorSpace(n_lambda,v_lambda,chi)

     if ( ProjectEV ) then
        do i_lambda = 1, n_lambda
           v_l => v_lambda(:,:,:,:,:,i_lambda)
           v_ldotphi = inner_product(v_l,psi)
           do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
              chi_x(is)%Cl = chi_x(is)%Cl + &
                   & (lambda_i(i_lambda)*v_ldotphi)*v_lambda(ix,iy,iz,it,is,i_lambda)%Cl
           end do; end do; end do; end do; end do
        end do
     end if

     normpsi = fermion_norm(psi)

  end subroutine MatrixSignFunctionOperate

#undef chi_x
#undef phi_x
#undef Mphi_x
#undef phi_ix

end module SignFunction
```

FLICOverlap: Implements the FLIC Overlap operator, and makes use of the Ginsparg-Wilson relation to implement its squared version.

```
module FLICOverlap

  use ColourTypes
  use SignFunction
  use ChiralFLICOperator
  use ZolotarevApprox
  implicit none


  real(dp) :: epsilon_tol = 1.0d-8 !sign function tolerance
  integer, dimension(:), allocatable :: epsilon_iter_i !sign function iterations
```

```
  integer , dimension(:) , allocatable :: miniter_i , maxiter_i
  real , dimension(:) , allocatable :: meaniter_i

  logical , parameter :: VerboseOverlap = .true. , DebugOverlap = .false.

  real(DP) :: lambda_scale

  integer :: dummy

  integer :: ncalls2 = 0
  real(dp) :: maxtime2 = 0, mintime2
  real(dp) :: meantime2 = 0.0

  character(len=256) :: ZoloFile , InputHiMode , InputLoMode

contains

  subroutine GetEigenSpace(lambda_min , lambda_max)

    real(dp) :: lambda_min , lambda_max

    real(dp) :: m_w, bct , error
    integer :: m_ev , m_dummy, i_lambda

    type(colour_vector) , dimension(nxp ,nyp ,nzp ,ntp ,ns) :: psi , Dpsi
    integer :: ix ,iy ,iz ,it ,is

    mpiprint *, "Reading Hi Modes"
    if ( i_am_root ) then
       open (100, file=InputHiMode , status = "old" , form = "unformatted" , action = "read")
       read (100) m_ev , m_dummy, delta_lambda , m_w, bct
       read (100) lambda_max
       close(100)
    end if

    call BroadCastReal(lambda_max , mpi_root_rank)

    mpiprint *, "Reading Lo Modes"
    call ReadEigenspace(n_lambda ,m_dummy, epsilon_lambda_i , v_lambda , delta_lambda ,m_w, bct , InputLoMode)

    do i_lambda=1,n_lambda
       call BroadCastReal(epsilon_lambda_i(i_lambda) ,mpi_root_rank)
    end do
    call BroadCastReal(delta_lambda , mpi_root_rank)

    lambda_min = epsilon_lambda_i(n_lambda)

    epsilon_lambda_i = sign(1.0d0, epsilon_lambda_i)

  end subroutine GetEigenSpace

  subroutine ReadZolotarevApprox

    integer :: ipole

    if ( i_am_root ) then
       open(110, file=ZoloFile , status = "old" , form = "formatted" , action = "read")
       read(110,*) n_poles
       read(110,*) x_min , x_max
       read(110,*) c_2n , d_n
    end if

    call BroadCastInteger(n_poles , mpi_root_rank)
    call BroadCastReal(x_min , mpi_root_rank)
    call BroadCastReal(x_max , mpi_root_rank)
    call BroadCastReal(c_2n , mpi_root_rank)
    call BroadCastReal(d_n , mpi_root_rank)

    do ipole=1,n_poles
       if ( i_am_root) read(110,*) b_l(ipole) , c_l(ipole)
       call BroadCastReal(b_l(ipole) ,mpi_root_rank)
       call BroadCastReal(c_l(ipole) ,mpi_root_rank)
    end do

    if ( i_am_root) close(110)

  end subroutine ReadZolotarevApprox

  subroutine InitialiseFLICOverlap(m_w, c_sw , bct , U_xd , UFL_xd)

    !Must call Initialise Operators each time the fermion mass is changed
    !or the gauge field is changed

    real(DP) :: m_w, c_sw , bct !clover-wilson mass, clover coefficient , time boundary conditions
    type(colour_matrix) , dimension(:,:,:,:,:) :: U_xd , UFL_xd
    real(DP) :: lambda_min , lambda_max
```

```fortran
    real(DP) :: kappa, delta, dx, x, err, err2
    integer :: i_pole

    call InitialiseFLICOperator(U_xd,UFL_xd,u0_bar,u0fl_bar,m_w,c_sw,bct)

    if ( project_eigenspace ) then
       call GetEigenSpace(lambda_min,lambda_max)
    else
       lambda_max = 8.0d0 - m_w
       lambda_min = 1.0d0 - abs(m_w - 1.0d0)
    end if

    call ReadZolotarevApprox

    kappa = abs(lambda_max/lambda_min)
    !zolotarev approximation is good between [1.0d0,kappa]
    lambda_scale = abs(kappa/lambda_max)

    do i_pole=1,n_poles
       !b_l(i_pole) = b_l(i_pole)
       c_l(i_pole) = c_l(i_pole)/(lambda_scale**2)
    end do
    c_2n = c_2n/(lambda_scale**2)
    d_n = lambda_scale*d_n

    delta = 2.0d-7

    if (VerboseOverlap) then
       mpiprint *, "kappa", kappa
       mpiprint *, "lambda_min", lambda_min
       mpiprint *, "scaled_min", lambda_min*lambda_scale
       mpiprint *, "min_error", 1.0d0 - zolotarev(abs(lambda_min))
       mpiprint *, "lambda_max", lambda_max
       mpiprint *, "scaled_max", lambda_max*lambda_scale
       mpiprint *, "max_error", 1.0d0 - zolotarev(abs(lambda_max))
       mpiprint *, "lambda_scale", lambda_scale
       mpiprint *, "n_poles", n_poles
       mpiprint *, "polar order needed",  -0.25d0*sqrt(kappa)*log(delta/2.0d0)
       dx = abs(lambda_max-lambda_min)/(nlx*nly*nlz*nlt*ns*nc)

       err = 0.0d0
       err2 = 0.0d0
       do x=abs(lambda_min),abs(lambda_max),dx
          err = err + abs(1.0d0 - zolotarev(x))
          err2 = max(err2,abs(zolotarev(x) - 1.0d0))
       end do
       err = err/(nlx*nly*nlz*nlt*ns*nc)
       mpiprint *, "mean error", err, err2

    end if

  contains

    function zolotarev(x)
      real(DP) :: zolotarev
      real(DP) :: x

      zolotarev = d_n*x*(x**2+c_2n)*(sum(b_l/(x**2+c_l)))

    end function zolotarev

  end subroutine InitialiseFLICOverlap

  subroutine FinaliseOverlap

    deallocate(b_l)
    deallocate(c_l)

    deallocate(epsilon_iter_i)
    deallocate(miniter_i)
    deallocate(maxiter_i)
    deallocate(meaniter_i)

    deallocate(epsilon_lambda_i)
    deallocate(v_lambda)

  end subroutine FinaliseOverlap

#define phi_x(is) phi(ix,iy,iz,it,is)
#define Dphi_x(is) Dphi(ix,iy,iz,it,is)
#define epsilon_Hphi_x(is) epsilon_Hphi(ix,iy,iz,it,is)

  subroutine OverlapDiracOperate(phi, Dphi)

    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: Dphi
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: epsilon_Hphi
```

```
  integer :: ix,iy,iz,it,is
  real(dp) :: intime, outtime

  if (timing) intime = mpi_wtime()

  !Evaluate the sign function
  call MatrixSignFunctionOperate(n_poles, phi, epsilon_Hphi, epsilon_tol, epsilon_iter_i, &
       & FLICOperate, SqFLICOperate, n_lambda, epsilon_lambda_i, v_lambda, delta_lambda )

  !Multiply by gamma_5
  do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
     epsilon_Hphi_x(3)%Cl = -epsilon_Hphi_x(3)%Cl
     epsilon_Hphi_x(4)%Cl = -epsilon_Hphi_x(4)%Cl
  end do; end do; end do; end do


  !Dphi = (1/2)*(1 + gamma_5*epsilon(H))*phi
  do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
     Dphi_x(is)%Cl = 0.5d0*(phi_x(is)%Cl + epsilon_Hphi_x(is)%Cl)
  end do; end do; end do; end do; end do

  if (timing) then
     ncalls2 = ncalls2 + 1
     outtime = mpi_wtime()
     call TimingUpdate(ncalls2,outtime,intime,mintime2,maxtime2,meantime2)
     where ( epsilon_iter_i == 0 ) epsilon_iter_i = epsilon_iter_i(1)
     where ( epsilon_iter_i < miniter_i ) miniter_i = epsilon_iter_i
     where ( epsilon_iter_i > maxiter_i ) maxiter_i = epsilon_iter_i
     meaniter_i = (ncalls2 - 1)*meaniter_i/ncalls2 + epsilon_iter_i/real(ncalls2)
  end if

end subroutine OverlapDiracOperate

subroutine SqOverlapDiracOperate(phi, Dphi, lambda_pm)

  !assumes phi is chiral, with chirality lambda_pm
  type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi
  type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: Dphi
  integer :: lambda_pm

  type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: epsilon_Hphi

  integer :: ix,iy,iz,it
  real(dp) :: intime, outtime

  if (timing) intime = mpi_wtime()
  !Evaluate the sign function

  call MatrixSignFunctionOperate(n_poles, phi, epsilon_Hphi, epsilon_tol, epsilon_iter_i, &
       & FLICOperate, SqFLICOperate, n_lambda, epsilon_lambda_i, v_lambda, delta_lambda )

  !Dphi = (1/2)*phi + (1/4)*(gamma_5*epsilon(H)*phi + lambda_pm*epsilon(H)*phi)

  if ( lambda_pm == +1 ) then
     do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
        Dphi_x(1)%Cl = 0.5d0*( phi_x(1)%Cl + epsilon_Hphi_x(1)%Cl )
        Dphi_x(2)%Cl = 0.5d0*( phi_x(2)%Cl + epsilon_Hphi_x(2)%Cl )
        Dphi_x(3)%Cl = 0.0d0
        Dphi_x(4)%Cl = 0.0d0
     end do; end do; end do; end do
  else
     do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
        Dphi_x(1)%Cl = 0.0d0
        Dphi_x(2)%Cl = 0.0d0
        Dphi_x(3)%Cl = 0.5d0*( phi_x(3)%Cl - epsilon_Hphi_x(3)%Cl )
        Dphi_x(4)%Cl = 0.5d0*( phi_x(4)%Cl - epsilon_Hphi_x(4)%Cl )
     end do; end do; end do; end do
  end if

  if (timing) then
     ncalls2 = ncalls2 + 1
     outtime = mpi_wtime()
     call TimingUpdate(ncalls2,outtime,intime,mintime2,maxtime2,meantime2)
     where ( epsilon_iter_i == 0 ) epsilon_iter_i = epsilon_iter_i(1)
     where ( epsilon_iter_i < miniter_i ) miniter_i = epsilon_iter_i
     where ( epsilon_iter_i > maxiter_i ) maxiter_i = epsilon_iter_i
     meaniter_i = (ncalls2 - 1)*meaniter_i/ncalls2 + epsilon_iter_i/real(ncalls2)
  end if

end subroutine SqOverlapDiracOperate

subroutine SqChiralOverlapOperate(phi, Dphi, lambda_pm)

  !assumes phi is chiral, with chirality lambda_pm
  type(colour_vector), dimension(nxp,nyp,nzp,ntp,nsp) :: phi
  type(colour_vector), dimension(nxp,nyp,nzp,ntp,nsp) :: Dphi
  integer :: lambda_pm
```

```
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi_s, epsilon_Hphi

    real(dp) :: intime, outtime
    integer :: ix,iy,iz,it,is,js

    if (timing) intime = mpi_wtime()
    !Evaluate the sign function

    is = 2 - lambda_pm
    js = 2 + lambda_pm

    phi_s(:,:,:,:,is  ) = phi(:,:,:,:,1)
    phi_s(:,:,:,:,is+1) = phi(:,:,:,:,2)
    phi_s(:,:,:,:,js  ) = zero_vector
    phi_s(:,:,:,:,js+1) = zero_vector

    call MatrixSignFunctionOperate(n_poles, phi_s, epsilon_Hphi, epsilon_tol, epsilon_iter_i, &
         & FLICOperate, SqFLICOperate, n_lambda, epsilon_lambda_i, v_lambda, delta_lambda )

    if ( lambda_pm == 1 ) then
       do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
          Dphi_x(1)%Cl = 0.5d0*(phi_x(1)%Cl + epsilon_Hphi_x(1)%Cl)
          Dphi_x(2)%Cl = 0.5d0*(phi_x(2)%Cl + epsilon_Hphi_x(2)%Cl)
       end do; end do; end do; end do
    else
       do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
          Dphi_x(1)%Cl = 0.5d0*(phi_x(1)%Cl - epsilon_Hphi_x(3)%Cl)
          Dphi_x(2)%Cl = 0.5d0*(phi_x(2)%Cl - epsilon_Hphi_x(4)%Cl)
       end do; end do; end do; end do
    end if

    if (timing) then
       ncalls2 = ncalls2 + 1
       outtime = mpi_wtime()
       call TimingUpdate(ncalls2,outtime,intime,mintime2,maxtime2,meantime2)
       where ( epsilon_iter_i == 0 ) epsilon_iter_i = epsilon_iter_i(1)
       where ( epsilon_iter_i < miniter_i ) miniter_i = epsilon_iter_i
       where ( epsilon_iter_i > maxiter_i ) maxiter_i = epsilon_iter_i
       meaniter_i = (ncalls2 - 1)*meaniter_i/ncalls2 + epsilon_iter_i/real(ncalls2)
    end if

  end subroutine SqChiralOverlapOperate

  subroutine HermitianOverlapOperate(phi, Dphi)

    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: Dphi

    integer :: ix,iy,iz,it,is
    call OverlapDiracOperate(phi, Dphi)

    !Multiply by gamma_5
    do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
       Dphi_x(3)%Cl = -Dphi_x(3)%Cl
       Dphi_x(4)%Cl = -Dphi_x(4)%Cl
    end do; end do; end do; end do

  end subroutine HermitianOverlapOperate

  subroutine SqHermitianOverlapOperate(phi, Dphi)

    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: Dphi
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: Dsqphi

    integer :: ix,iy,iz,it,is

    call OverlapDiracOperate(phi, Dphi)

    !Multiply by gamma_5
    do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
       Dphi_x(3)%Cl = -Dphi_x(3)%Cl
       Dphi_x(4)%Cl = -Dphi_x(4)%Cl
    end do; end do; end do; end do

    call OverlapDiracOperate(Dphi, Dsqphi)

    !Multiply by gamma_5
    do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
       Dphi_x(3)%Cl = -Dphi_x(3)%Cl
       Dphi_x(4)%Cl = -Dphi_x(4)%Cl
    end do; end do; end do; end do

    Dphi = Dsqphi

  end subroutine SqHermitianOverlapOperate

end module FLICOverlap
```

A.5 Quark Propagator Code

ConjGradSolvers: Provides two conjugate gradient inverter routines, standard CG for Hermitian positive definite matrices, and the more general BiCGStab routine for other matrices.

```
module ConjGradSolvers

  use FermionField
  implicit none

contains

#define residue_x(is) residue(ix,iy,iz,it,is)
#define psi_x(is) psi(ix,iy,iz,it,is)
#define phi_x(is) phi(ix,iy,iz,it,is)
#define Mphi_x(is) Mphi(ix,iy,iz,it,is)
#define chi_x(is) chi(ix,iy,iz,it,is)
#define xi_x(is) xi(ix,iy,iz,it,is)
#define Mxi_x(is) Mxi(ix,iy,iz,it,is)

  subroutine CGInvert(psi, chi, tolerance, iterations, MatrixOperate)

    !A routine to solve the matrix equation M chi = psi, using an initial guess chi
    !M _must_ be hermitian and positive definite.

    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: psi !source vector
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: chi !solution vector and initial guess
    real(dp) :: tolerance !the precision desired for the solution
    integer :: iterations !the total number of iterations required
    interface
       subroutine MatrixOperate(phi, Dphi)
         use ColourTypes
         type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi
         type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: Dphi
       end subroutine MatrixOperate
    end interface

    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: residue
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi, Mphi
    real(dp) :: delta, prevdelta, norm_psi, omega, theta, phidotMphi
    integer :: ix,iy,iz,it,is

    norm_psi = fermion_norm(psi)

    call MatrixOperate(chi, Mphi)

    !residue = psi - Mphi
    do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
       residue_x(is)%Cl = psi_x(is)%Cl - Mphi_x(is)%Cl
    end do; end do; end do; end do; end do

    phi = residue

    delta = fermion_normsq(residue)

    call MatrixOperate(phi,Mphi)

    phidotMphi = real_inner_product(phi,Mphi)

    iterations = 0

    do
       if (sqrt(delta)/norm_psi < tolerance) exit

       iterations = iterations + 1

       omega = delta/phidotMphi

       !residue = residue - omega*Mphi
       do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
          residue_x(is)%Cl = residue_x(is)%Cl - omega*Mphi_x(is)%Cl
       end do; end do; end do; end do; end do

       !chi = chi + omega*phi
       do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
          chi_x(is)%Cl = chi_x(is)%Cl + omega*phi_x(is)%Cl
       end do; end do; end do; end do; end do

       prevdelta = delta
       delta = fermion_normsq(residue)

       theta = -delta/prevdelta
```

```
      !phi = residue - theta*phi
      do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
         phi_x(is)%Cl = residue_x(is)%Cl - theta*phi_x(is)%Cl
      end do; end do; end do; end do; end do

      call MatrixOperate(phi,Mphi)

      phidotMphi = real_inner_product(phi,Mphi)

   end do

end subroutine CGInvert

subroutine BiCGStabInvert(psi, chi, tolerance, iterations, MatrixOperate)

   !A routine to solve the matrix equation M chi = psi, using an initial guess chi

   type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: psi !source vector
   type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: chi !solution vector and initial guess
   real(dp) :: tolerance !the precision desired for the solution
   integer :: iterations !the total number of iterations required
   interface
      subroutine MatrixOperate(phi, Dphi)
        use ColourTypes
        type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi
        type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: Dphi
      end subroutine MatrixOperate
   end interface

   type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: residue, r0
   type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi, Mphi, xi, Mxi
   real(dp) :: epsilon, norm_psi
   complex(dc) :: delta, delta_prime, omega, beta, theta
   integer :: ix,iy,iz,it,is

   norm_psi = fermion_norm(psi)

   call MatrixOperate(chi,Mphi)

   !residue = psi - Mphi
   do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
      residue_x(is)%Cl = psi_x(is)%Cl - Mphi_x(is)%Cl
   end do; end do; end do; end do; end do

   epsilon = fermion_normsq(residue)

   r0 = residue
   delta = inner_product(r0,residue)

   xi = residue
   call MatrixOperate(xi,Mxi)

   delta_prime = inner_product(r0,Mxi)

   iterations = 0

   do

      if (sqrt(epsilon)/norm_psi < tolerance) exit

      iterations = iterations + 1

      omega = delta/delta_prime

      !phi = residue - omega*Mxi
      do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
         phi_x(is)%Cl = residue_x(is)%Cl - omega*Mxi_x(is)%Cl
      end do; end do; end do; end do; end do

      call MatrixOperate(phi,Mphi)

      theta = inner_product(Mphi,phi)/fermion_normsq(Mphi)

      !residue = phi - theta*Mphi
      do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
         residue_x(is)%Cl = phi_x(is)%Cl - theta*Mphi_x(is)%Cl
      end do; end do; end do; end do; end do

      epsilon = fermion_normsq(residue)

      !chi = chi + omega*xi + theta*phi
      do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
         chi_x(is)%Cl = chi_x(is)%Cl + omega*xi_x(is)%Cl + theta*phi_x(is)%Cl
      end do; end do; end do; end do; end do

      delta = inner_product(r0,residue)

      beta = -delta/(delta_prime*theta)
```

```
      !xi = residue - beta*(xi - theta*Mxi)
      do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
         xi_x(is)%Cl = residue_x(is)%Cl - beta*(xi_x(is)%Cl - theta*Mxi_x(is)%Cl)
      end do; end do; end do; end do; end do

      call MatrixOperate(xi,Mxi)

      delta_prime = inner_product(r0,Mxi)

    end do

  end subroutine BiCGStabInvert

end module ConjGradSolvers
```

QuarkPropagator: Uses the BiCGStab algorithm to solve for the FLIC quark propagator.

```
module QuarkPropagator

  use GaugeField
  use FermionField
  use ConjGradSolvers
  use MPIInterface
  use ChiralFLICOperator
  implicit none

  real(dp) :: alpha_sm
  integer :: n_smear, js_start=1, js_end=1, jc_start=1, jc_end = nc
  logical :: SmearedSource

contains

  subroutine CalculatePropagator(jx,jy,jz,jt,kappa,tolerance,OutputPrefix,Dfermion,SmearedSource,alpha_sm,
       n_smear)

    integer, intent(in) :: jx,jy,jz,jt !source position
    real(DP), dimension(:) :: kappa
    real(DP) :: tolerance !the precision desired for the solution
    character(len=*) :: OutputPrefix
    interface
       subroutine Dfermion(phi, Dphi)
         use ColourTypes
         type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi
         type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: Dphi
       end subroutine Dfermion
    end interface

    logical :: SmearedSource
    real(dp) :: alpha_sm
    integer :: n_smear, nkappa, ikappa

    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: Dchi, rho
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: chi !solution vector

    integer :: js,jc,iterations
    real :: starttime, endtime
    real(dp) :: error, norm_rho
    integer :: ix,iy,iz,it,is,ic

    nkappa = size(kappa)

    do js=1,ns
       do jc=1,nc

          chi = zero_vector
          rho = zero_vector

          call GetSource(rho,jx,jy,jz,jt,js,jc,SmearedSource,alpha_sm,n_smear,U_xd)

          norm_rho = fermion_norm(rho)

          do ikappa=1,nkappa

             m_f = 4.0d0 - 0.5d0/kappa(ikappa)

             starttime = mpi_wtime()
             call BiCGStabInvert(rho, chi, tolerance, iterations, Dfermion)
             endtime = mpi_wtime()

             mpiprint '(A,F20.5,A)', "Inversion took ", endtime-starttime, " seconds"

             call Dfermion(chi,Dchi)
             do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
                Dchi(ix,iy,iz,it,is)%Cl = Dchi(ix,iy,iz,it,is)%Cl - rho(ix,iy,iz,it,is)%Cl
```

```
          end do; end do; end do; end do; end do

          error = fermion_norm(Dchi)

          mpiprint *, "kappa ", kappa(ikappa), " iterations ", iterations, " error ", error, error/
               norm_rho

          call WritePropagatorColumn(OutputPrefix, kappa(ikappa), chi)

       end do

    end do
  end do

end subroutine CalculatePropagator

subroutine WritePropagatorColumn(OutputPrefix, kappa, chi)

  character(len=*) :: OutputPrefix

  character(len=273) :: PropagatorFile
  character(len=7) :: PropagatorSuffix

  real(DP) :: kappa !quark mass
  type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: chi !solution vector

  if ( i_am_root ) then
     write( PropagatorSuffix, '(a,F6.4)') "k", kappa
     PropagatorFile = OutputPrefix(1:len_trim(OutputPrefix)) // PropagatorSuffix // ".ptpp"
     open (100, file=PropagatorFile, status = "unknown", form = "unformatted", &
          & position = "append", action = "write")
  end if

  call WriteFermionField(chi,100)
  if ( i_am_root ) close(100)

end subroutine WritePropagatorColumn

end module QuarkPropagator
```

FLICProp: Main program for the calculation of the FLIC propagator.

```
program FLICPropagator

  use QuarkPropagator
  implicit none

  character(len=256) :: InputConfig
  character(len=256) :: OutputPrefix
  integer :: ix,iy,iz,it, ic, ikappa
  integer, parameter :: nkappa = 3

  real(DP) :: tolerance, kappa_i(nkappa), kappa

  character(len=20) :: startDate, startTime, endDate, endTime

  call InitialiseMPI

  call InitShadowGaugeField
  call InitShadowFermionField

  call InitTimer

  mpiprint *, "Enter the gauge configuration file to read from:"
  read(*,'(A256)') InputConfig
  mpiprint '(A256)', InputConfig

  mpiprint *, "Enter the output file prefix:"
  read(*,'(A256)') OutputPrefix
  mpiprint '(A256)', OutputPrefix

  mpiprint *, "FLIC parameters"
  mpiprint *, "Wilson mass parameter"
  do ikappa=1,nkappa
     read(*,*) kappa
     m_f = 4.0d0 - 0.5d0/kappa
     mpiprint *, "m_w=", m_f, " kappa= ", kappa
     kappa_i(ikappa) = kappa
  end do

  mpiprint *, "Clover term c_sw"
  read(*,*) c_sw
  mpiprint *, "c_sw=", c_sw

  mpiprint *, "Smearing fraction"
  read(*,*) alpha_smear
  mpiprint *, "alpha_smear=", alpha_smear
```

```
  mpiprint *, "Smearing sweeps"
  read(*,*)  ape_sweeps
  mpiprint *, "ape_sweeps", ape_sweeps

  mpiprint *, "Propagator parameters"
  mpiprint *, "Source position (ix,iy,iz,it)"
  read(*,*)  ix
  read(*,*)  iy
  read(*,*)  iz
  read(*,*)  it
  mpiprint *, " ix=",ix," iy=",iy," iz=",iz," it=",it

  mpiprint *, "Smeared source?"
  read(*,*)  SmearedSource
  read(*,*)  alpha_sm
  read(*,*)  n_smear

  mpiprint *, " Source smearing:", SmearedSource, alpha_sm, n_smear

  mpiprint *, "Boundary Conditions (bct)"
  read(*,*)  bct
  mpiprint *, " bct=",bct

  mpiprint *, "Enter the desired solution precision:"
  read(*,*) tolerance
  mpiprint *, tolerance

  mpiprint *, "Starting time"
  call date_and_time(date=startDate, time=startTime)

  mpiprint *, "Reading Gauge Field"
  call ReadGaugeField(InputConfig, U_xd)

  if ( uzero /= 1.0d0 ) then
     u0_bar = uzero
  else
     call GetUZero(U_xd,u0_bar)
  end if

  call APESmearLinks(U_xd,UFL_xd,alpha_smear,ape_sweeps)
  call GetUZero(UFL_xd,u0fl_bar)

  mpiprint *, u0_bar, u0fl_bar, m_f, c_sw, bct
  call InitialiseFLICOperator(U_xd,UFL_xd,u0_bar,u0fl_bar,m_f, c_sw, bct)

  call CalculatePropagator(ix,iy,iz,it,kappa_i,tolerance,OutputPrefix,Dflic,SmearedSource,alpha_sm,n_smear
       )

  Mflops = (145*24*nlattice)*1.0e-6

  mpiprint *, "FLIC timing", ncalls, meantime, mintime, maxtime
  mpiprint *, "FLIC rate", ncalls, Mflops/meantime, Mflops/mintime, Mflops/maxtime

  call date_and_time(date=endDate, time=endTime)
  mpiprint *, "Started on ", startDate, " at ", startTime
  mpiprint *, "Finished on ", endDate, " at ", endTime

  call FinaliseMPI

end program FLICPropagator
```

OverlapPropagator: Implements the standard conjugate gradient inversion algorithm, as applied to overlap fermions.

```
module OverlapPropagator

  use FermionField
  use FLICOverlap
  use ConjGradSolvers
  implicit none

  real(dp) :: alpha_sm
  integer :: n_smear, js_start=1, js_end=1, jc_start=1, jc_end = nc
  logical :: SmearedSource

contains

#define chi_x(is)  chi(ix,iy,iz,it,is)
#define Dchi_x(is) Dchi(ix,iy,iz,it,is)
#define chi_ix(is) chi_i(ix,iy,iz,it,is,i_mu)
#define phi_x(is)  phi(ix,iy,iz,it,is)
#define Mphi_x(is) Mphi(ix,iy,iz,it,is)
#define phi_ix(is) phi_i(ix,iy,iz,it,is,i_mu)
#define residue_x(is) residue(ix,iy,iz,it,is)
#define rho_x(is)  rho(ix,iy,iz,it,is)
```

```fortran
subroutine CalculatePropagator(jx,jy,jz,jt,n_mu,mu_i,tolerance,OutputPrefix,SmearedSource,alpha_sm,
    n_smear)

  integer, intent(in) :: jx,jy,jz,jt,n_mu !source position, number of shifts
  real(dp), dimension(n_mu), intent(in) :: mu_i !quark masses
  real(dp) :: tolerance !the precision desired for the solution
  character(len=*) :: OutputPrefix
  logical :: SmearedSource
  real(dp) :: alpha_sm
  integer :: n_smear

  real(dp), dimension(n_mu) :: muprime_i, normchi_i !quark masses
  type(colour_vector), dimension(nxp,nyp,nzp,ntp,nsp) :: rho_sp
  type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: chi, Dchi, rho
  type(colour_vector), dimension(nx,ny,nz,nt,nsp,n_mu) :: phi_i
  type(colour_vector), dimension(nx,ny,nz,nt,ns,n_mu) :: chi_i !solution vectors

  real(dp) :: norm_chi, norm_rho
  integer :: lambda_pm
  complex(dc) :: lambda_test

  character(len=273) :: OutputFile, PropagatorFile
  character(len=6) :: OutputSuffix, PropagatorSuffix

  integer :: js,jc
  integer :: ix,iy,iz,it,is,ic,i_mu,iterations
  integer, dimension(n_mu) :: iterations_i
  real(dp) :: starttime, endtime

  do js=js_start,js_end
     if ( js > js_start ) jc_start = 1
     if ( js > js_start ) jc_end = nc
     do jc=jc_start,jc_end

        rho = zero_vector

        call GetSource(rho,jx,jy,jz,jt,js,jc,SmearedSource,alpha_sm,n_smear,U_xd)

        chi = zero_vector
        norm_rho = fermion_norm(rho)

        call normalise(rho,norm_rho)

        mpiprint *, "Source ", js, jc, " L2 Source norm", norm_rho

        select case(js)
        case(1:2)
           lambda_pm = 1
           rho_sp = rho(:,:,:,:,1:2)
        case(3:4)
           lambda_pm = -1
           rho_sp = rho(:,:,:,:,3:4)
        end select

        muprime_i = mu_i**2/(1.0d0-mu_i**2)

        starttime = mpi_wtime()
        call OverlapMultiMassInvert(n_mu, muprime_i, rho_sp, lambda_pm, phi_i, tolerance, iterations_i )
        endtime = mpi_wtime()

        mpiprint '(A,F20.2,A,F15.5,A)', "Inversion took ", endtime-starttime, " seconds = ",(endtime-
              starttime)/3600.0," hours."

        select case(js)
        case(1:2)
           chi_i(:,:,:,:,1:2,:) = phi_i
           chi_i(:,:,:,:,3:4,:) = zero_vector
        case(3:4)
           chi_i(:,:,:,:,1:2,:) = zero_vector
           chi_i(:,:,:,:,3:4,:) = phi_i
        end select


        mpiprint *, " #Mu ", " Mass ", " Convergence "
        do i_mu=1,n_mu
           mpiprint *, i_mu, mu_i(i_mu), iterations_i(i_mu)
        end do

        !Check Solution
        do i_mu = 1,n_mu
           do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
              chi_ix(is)%Cl = chi_ix(is)%Cl/(1.0d0 - mu_i(i_mu)**2)
           end do; end do; end do; end do; end do
        end do

        !Multiply solution by D^\dagger to get inverse of D
        do i_mu = 1,n_mu
```

```
            chi(1:nx,1:ny,1:nz,1:nt,:) = chi_i(:,:,:,:,:,i_mu)
            call OverlapDiracOperate(chi,Dchi)
            if (js < 3) then
               !+ve chirality, multiply by +gamma_5
               do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
                  Dchi_x(3)%Cl = -Dchi_x(3)%Cl
                  Dchi_x(4)%Cl = -Dchi_x(4)%Cl
               end do; end do; end do; end do
            else
               !-ve chirality, multiply by -gamma_5
               do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
                  Dchi_x(1)%Cl = -Dchi_x(1)%Cl
                  Dchi_x(2)%Cl = -Dchi_x(2)%Cl
               end do; end do; end do; end do
            end if

            do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
               chi_ix(is)%Cl = (1.0d0-mu_i(i_mu))*Dchi_x(is)%Cl + mu_i(i_mu)*chi_ix(is)%Cl
            end do; end do; end do; end do; end do

         end do

         mpiprint *, " Propagator solution quality"
         mpiprint *, " #Mu ", " Error "
         do i_mu=1,n_mu
            chi(1:nx,1:ny,1:nz,1:nt,:) = chi_i(:,:,:,:,:,i_mu)
            call OverlapDiracOperate(chi,Dchi)
            do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
               Dchi_x(is)%Cl = (1.0d0-mu_i(i_mu))*Dchi_x(is)%Cl + mu_i(i_mu)*chi_ix(is)%Cl - rho_x(is)%Cl
            end do; end do; end do; end do; end do

            norm_chi = fermion_norm(Dchi)
            mpiprint *, i_mu, norm_chi, norm_chi/norm_rho
         end do

         call WritePropagatorColumn(OutputPrefix,n_mu,mu_i,chi_i)

      end do
   end do

end subroutine CalculatePropagator

subroutine WritePropagatorColumn(OutputPrefix,n_mu,mu_i,chi_i)

  character(len=*) :: OutputPrefix

  character(len=273) :: PropagatorFile
  character(len=6) :: PropagatorSuffix

  integer, intent(in) :: n_mu !source position, number of shifts
  real(dp), dimension(n_mu), intent(in) :: mu_i !quark masses
  type(colour_vector), dimension(nx,ny,nz,nt,ns,n_mu) :: chi_i !solution vectors

  integer :: i_mu

  do i_mu=1,n_mu
     if ( i_am_root ) then
        write( PropagatorSuffix, '(a,I4.4)') "mu", int(10000*mu_i(i_mu))
        PropagatorFile = OutputPrefix(1:len_trim(OutputPrefix)) // PropagatorSuffix // ".ptpp"
        open (100+i_mu, file=PropagatorFile, status = "unknown", form = "unformatted", &
             & position = "append", action = "write")
     end if
     call WriteFermionField(chi_i(:,:,:,:,:,i_mu),100+i_mu)
     if ( i_am_root ) close(100+i_mu)
  end do

end subroutine WritePropagatorColumn

subroutine OverPropToTimeSlice(n_mu,mu_i,OutputPrefix,TimeSlicePrefix)

  integer, intent(in) :: n_mu !source position, number of shifts
  real(DP), dimension(n_mu) :: mu_i !quark masses
  character(len=256) :: OutputPrefix, TimeSlicePrefix

  character(len=273) :: TimeSliceFile, PropagatorFile
  character(len=6) :: PropagatorSuffix

  complex(dc), dimension(nx*ny*nz,nt,ns,nc) :: chi
  complex(dc), dimension(nx*ny*nz,nt,nc,ns,nc,ns) :: eta, xi

  integer :: i_mu,is,ic,js,jc

  real(dp),dimension(nlt) :: PionCorr, EffMass
  real(dp),dimension(nt)  :: PionCorr_zt, PionCorr_in
  integer :: ix, jx, iy, jy, iz, jz, it, jt
  integer :: irank

  do i_mu = 1,n_mu
```

```
write( PropagatorSuffix, '(a,I4.4)') "mu", int(10000*mu_i(i_mu))
PropagatorFile = OutputPrefix(1:len_trim(OutputPrefix)) // PropagatorSuffix // ".ptpp"

if ( i_am_root ) then
   open (100, file=PropagatorFile, status = "old", form = "unformatted", action = "read")
end if

do is=1,ns
   do ic=1,nc

      call ReadFermionField_sc(chi,100)

      do js=1,ns; do jc=1,nc
         eta(:,:,jc,js,ic,is) = chi(:,:,js,jc)
      end do; end do

   end do
end do

if ( i_am_root ) close(100)

call MPIBarrier

xi(:,:,:,1,:,:) = (1.0d0/sqrt(2.0d0))*(eta(:,:,:,1,:,:) + eta(:,:,:,3,:,:))
xi(:,:,:,2,:,:) = (1.0d0/sqrt(2.0d0))*(eta(:,:,:,2,:,:) + eta(:,:,:,4,:,:))
xi(:,:,:,3,:,:) = (1.0d0/sqrt(2.0d0))*(eta(:,:,:,3,:,:) - eta(:,:,:,1,:,:))
xi(:,:,:,4,:,:) = (1.0d0/sqrt(2.0d0))*(eta(:,:,:,4,:,:) - eta(:,:,:,2,:,:))

eta(:,:,:,:,:,1) = (1.0d0/sqrt(2.0d0))*(xi(:,:,:,:,:,1) + xi(:,:,:,:,:,3))
eta(:,:,:,:,:,2) = (1.0d0/sqrt(2.0d0))*(xi(:,:,:,:,:,2) + xi(:,:,:,:,:,4))
eta(:,:,:,:,:,3) = (1.0d0/sqrt(2.0d0))*(xi(:,:,:,:,:,3) - xi(:,:,:,:,:,1))
eta(:,:,:,:,:,4) = (1.0d0/sqrt(2.0d0))*(xi(:,:,:,:,:,4) - xi(:,:,:,:,:,2))

if ( site_is_mine(1,1,1,1) ) then
   do is=1,ns
      do ic=1,nc
         eta(1,1,ic,is,ic,is) = eta(1,1,ic,is,ic,is) - 1.0d0
      end do
   end do
end if

eta = eta/(1.0d0 - mu_i(i_mu))

do jt=1,nt
   PionCorr_zt(jt) = 0.5d0*sum(abs(eta(:,jt,:,:,:,:))**2)
end do

if ( i_am_root ) then
   PionCorr = 0.0d0
   do irank=0,nproc-1
      call GetSubLattice(irank,ix,jx,iy,jy,iz,jz,it,jt)
      if ( irank /= mpi_root_rank ) then
         call RecvRealField(PionCorr_in,irank)
      else
         PionCorr_in = PionCorr_zt
      end if

      PionCorr(it:jt) = PionCorr(it:jt) + PionCorr_in(1:nt)
   end do
else
   call SendRealField(PionCorr_zt,mpi_root_rank)
end if

if ( i_am_root ) then

   EffMass(1) = 0.0d0
   do jt = 2,nlt
      EffMass(jt) = log(abs( PionCorr(jt-1)/PionCorr(jt) ))
   end do

   mpiprint *, "TimeSlice  Pion Correlator  Effective Mass :mu = ", mu_i(i_mu)
   do jt = 1,nlt
      mpiprint '(I9, E15.7, E15.7)', jt, PionCorr(jt), EffMass(jt)
   end do

end if

TimeSliceFile = TimeSlicePrefix(1:len_trim(TimeSlicePrefix)) // PropagatorSuffix // ".tsp"

if ( i_am_root ) then
   open (200, file=TimeSliceFile, status = "unknown", form = "unformatted", action = "write", &
      & position = "append" )
end if

call WriteTimeSlicePropagator(eta,200)

if ( i_am_root ) close(200)
```

```
  end do

end subroutine OverPropToTimeSlice

subroutine OverlapMultiMassInvert(n_mu, mu_a, psi, lambda_pm, chi_i, tolerance, iterations_i)

   !A routine to solve the matrix equation (M+mu)v = psi for a system of shifts mu
   !mu must be ordered such that mu(1) contains the least convergent system
   ! (i.e. the system that converges slowest).

   !M _must_ be hermitian and positive definite.

   !Assumes a chiral source psi, chirality lambda_pm

   !Assumed: tolerance >= epsilon(1.0d0)

   integer, intent(in) :: n_mu !number of shifts
   real(dp), dimension(n_mu), intent(in) :: mu_a !quark masses
   type(colour_vector), dimension(nxp,nyp,nzp,ntp,nsp) :: psi !source vector
   integer :: lambda_pm !the chirality of psi
   type(colour_vector), dimension(nx,ny,nz,nt,nsp,n_mu) :: chi_i !solution vectors
   real(dp) :: tolerance !the precision desired for the solution
   integer, dimension(n_mu) :: iterations_i

   integer :: iterations !the total number of iterations required
   real(dp), dimension(n_mu) :: mu_i !offset matrix shifts
   type(colour_vector), dimension(nxp,nyp,nzp,ntp,nsp) :: residue, phi, Mphi
   type(colour_vector), dimension(nx,ny,nz,nt,nsp,n_mu) :: phi_i
   real(dp) :: normsq_r, prevnormsq_r, beta, prevbeta, alpha, phidotMphi, offset, normpsi
   real(dp), dimension(n_mu) :: beta_i, prevbeta_i, zeta_i, prevzeta_i, alpha_i, nextzeta_i
   logical, dimension(n_mu) :: converged_i, converged_j
   integer :: ix,iy,iz,it,is,i_mu
   real(dp) :: tau = 1.0d0 !tau = 1.0d-1

   !translate the mu shifts so that mu_i(1)=0
   offset = mu_a(1)
   if (offset /= 0) then
      mu_i = mu_a - offset
   else
      mu_i = mu_a
   end if

   chi_i = zero_vector
   residue = psi

   do i_mu=1,n_mu
      phi_i(:,:,:,:,:,i_mu) = residue(1:nx,1:ny,1:nz,1:nt,:)
   end do

   prevbeta = 1.0d0
   prevzeta_i = 1.0d0
   zeta_i= 1.0d0
   alpha_i = 0
   beta_i = beta

   normsq_r = fermion_normsq(residue)
   normpsi = sqrt(normsq_r)

   phi(1:nx,1:ny,1:nz,1:nt,:) = phi_i(:,:,:,:,:,1)
   call SqChiralOverlapOperate(phi,Mphi, lambda_pm)
   if (offset /= 0) then
      do is=1,nsp; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
         Mphi_x(is)%Cl = Mphi_x(is)%Cl + offset*phi_x(is)%Cl
      end do; end do; end do; end do; end do
   end if
   phidotMphi = real_inner_product(phi,Mphi)

   iterations = 0
   iterations_i = 0

   converged_i = .false.
   converged_j = .false.

   do
      iterations = iterations + 1

      do i_mu=1,n_mu
         if ( ( sqrt(normsq_r)*zeta_i(i_mu)/normpsi < tolerance ) .and. &
              & .not. converged_j(i_mu) ) iterations_i(i_mu) = iterations
      end do
      if (sqrt(normsq_r)/normpsi < tolerance) exit
      mpiprint *, iterations, sqrt(normsq_r)/normpsi, converged_j, epsilon_iter_i(1)
      !mpiprint *, iterations, sqrt(normsq_r), alpha, beta, phidotMphi

      !If the slowest convergence is extremely slow compared to the other shifts, we need to have a "
           safety net"
      !that prevents underflow in zeta_i for the already converged values.
```

```
      converged_i = (sqrt(normsq_r)*zeta_i/normpsi < epsilon(1.0d0) )
      converged_j = (sqrt(normsq_r)*zeta_i/normpsi < tolerance )
      do i_mu=1,n_mu
         if (converged_i(i_mu)) zeta_i(i_mu) = epsilon(1.0d0)
      end do

      beta = -normsq_r/phidotMphi

      nextzeta_i = zeta_i*prevzeta_i*prevbeta/( beta*alpha*(prevzeta_i-zeta_i)+prevzeta_i*prevbeta*(1.0d0
           -mu_i*beta))
      beta_i = beta*nextzeta_i/zeta_i

      do i_mu=1,n_mu
         if (.not. converged_i(i_mu)) then
            do is=1,nsp; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
               chi_ix(is)%Cl = chi_ix(is)%Cl - beta_i(i_mu)*phi_ix(is)%Cl
            end do; end do; end do; end do; end do
         end if
      end do

      do is=1,nsp; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
         residue_x(is)%Cl = residue_x(is)%Cl + beta*Mphi_x(is)%Cl
      end do; end do; end do; end do; end do

      prevnormsq_r = normsq_r
      normsq_r = fermion_normsq(residue)

      alpha = normsq_r/prevnormsq_r
      alpha_i = alpha*nextzeta_i*beta_i/(zeta_i*beta)

      do i_mu=1,n_mu
         if (.not. converged_i(i_mu) ) then
            do is=1,nsp; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
               phi_ix(is)%Cl = nextzeta_i(i_mu)*residue_x(is)%Cl + alpha_i(i_mu)*phi_ix(is)%Cl
            end do; end do; end do; end do; end do
         end if
      end do

      phi(1:nx,1:ny,1:nz,1:nt,:) = phi_i(:,:,:,:,:,1)
      call SqChiralOverlapOperate(phi,Mphi,lambda_pm)
      if (offset /= 0) then
         do is=1,nsp; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
            Mphi_x(is)%Cl = Mphi_x(is)%Cl + offset*phi_x(is)%Cl
         end do; end do; end do; end do; end do
      end if

      phidotMphi = real_inner_product(phi,Mphi)

      prevzeta_i = zeta_i
      zeta_i = nextzeta_i

      prevbeta = beta

      prevbeta_i = beta_i
   end do

   iterations_i(1) = iterations
   where (iterations_i == 0) iterations_i = iterations

 end subroutine OverlapMultiMassInvert

#undef chi_x
#undef Dchi_x
#undef chi_ix
#undef phi_x
#undef Mphi_x
#undef residue_x
#undef rho_x

end module OverlapPropagator
```

FLICOverProp: Main program for the calculation of the FLIC overlap propagator.

```
program FLICPropagator

  use FLICOverlap
  use OverlapPropagator
  implicit none

  character(len=256)  :: InputConfig, InputOverlapMode
  character(len=256)  :: OutputPrefix, TimeSlicePrefix
  integer :: ix,iy,iz,it, n_mu_input, ic

  integer, parameter :: n_mu = 14
  real(DP), dimension(n_mu) :: mu_i
```

```
real(DP) :: tolerance
integer :: i_null, i_ev, j_null, j_ev, m_ev, m_dummy, i_lambda, j_lambda, i_mu

character(len=20) :: startDate, startTime, endDate, endTime

call InitialiseMPI

call InitShadowGaugeField
call InitShadowFermionField

call InitTimer

startDate = ""
startTime = ""

endDate = ""
endTime = ""

mpiprint *, "Enter the gauge configuration file to read from:"
read(*,'(A256)') InputConfig
mpiprint *, InputConfig

mpiprint *, "Enter the low eigenmode file to read from:"
read(*,'(A256)') InputLoMode
mpiprint *, InputLoMode

mpiprint *, "Enter the high eigenmode file to read from:"
read(*,'(A256)') InputHiMode
mpiprint *, InputHiMode

mpiprint *, "Enter the output file prefix:"
read(*,'(A256)') OutputPrefix
mpiprint *, OutputPrefix

mpiprint *, "Enter the timeslice file prefix:"
read(*,'(A256)') TimeSlicePrefix
mpiprint *, TimeSlicePrefix

mpiprint *, "FLIC parameters"
mpiprint *, "Wilson mass parameter"
read(*,*) m_f
mpiprint *, "m_w=", m_f

mpiprint *, "Clover term c_sw"
read(*,*) c_sw
mpiprint *, "c_sw=", c_sw

mpiprint *, "Smearing fraction"
read(*,*) alpha_smear
mpiprint *, "alpha_smear=", alpha_smear

mpiprint *, "Smearing sweeps"
read(*,*) ape_sweeps
mpiprint *, "ape_sweeps", ape_sweeps

mpiprint *, "Propagator parameters"
mpiprint *, "Source position (ix,iy,iz,it)"
read(*,*) ix
read(*,*) iy
read(*,*) iz
read(*,*) it
read(*,*) js_start, js_end
read(*,*) jc_start, jc_end
mpiprint *, " ix=",ix," iy=",iy," iz=",iz," it=",it

mpiprint *, "Source type"
read(*,*) SmearedSource
read(*,*) alpha_sm
read(*,*) n_smear

mpiprint *, "Boundary Conditions (bct)"
read(*,*) bct
mpiprint *, " bct=",bct

mpiprint *, "Number of lo modes to project out:"
read(*,*) n_lambda
mpiprint *, " n_lambda=", n_lambda

mpiprint *, "Number of Fermion masses"
read(*,*) n_mu_input
if (n_mu_input /= n_mu ) then
   mpiprint *, 'Propagator: failed: Mismatch in nmu'
   stop
end if

mpiprint *, "Fermion masses, mu_i"
do i_mu=1,n_mu
   read(*,*) mu_i(i_mu)
```

```
end do
mpiprint *, mu_i

mpiprint *, "Enter the desired solution precision:"
read(*,*) tolerance
mpiprint *, tolerance

mpiprint *, "Starting time"
call date_and_time(date=startDate, time=startTime)

mpiprint *, "Reading Gauge Field"
call ReadGaugeField(InputConfig, U_xd)

if ( uzero /= 1.0d0 ) then
   u0_bar = uzero
else
   call GetUZero(U_xd,u0_bar)
end if

call APESmearLinks(U_xd,UFL_xd,alpha_smear,ape_sweeps)

call GetUZero(UFL_xd, u0fl_bar)

mpiprint *, "uzero", u0_bar, u0fl_bar

ZoloFile = "zolotarev.dat"

allocate(epsilon_lambda_i(n_lambda))
allocate(v_lambda(nx,ny,nz,nt,ns,n_lambda))

if ( i_am_root ) then
   open(110, file=ZoloFile, status = "old", form = "formatted", action = "read")
   read(110,*) n_poles
   close(110)
end if

call BroadCastInteger(n_poles,mpi_root_rank)

allocate(b_l(n_poles))
allocate(c_l(n_poles))

allocate(epsilon_iter_i(n_poles))
allocate(miniter_i(n_poles))
allocate(maxiter_i(n_poles))
allocate(meaniter_i(n_poles))

mpiprint *, "Calculating Operators"
call InitialiseFLICOverlap(m_f, c_sw, bct, U_xd, UFL_xd)

miniter_i = 1000
maxiter_i = 0
meaniter_i = 0.0

call CalculatePropagator(ix,iy,iz,it,n_mu,mu_i,tolerance,OutputPrefix,SmearedSource,alpha_sm,n_smear)

call OverPropToTimeSlice(n_mu,mu_i,OutputPrefix,TimeSlicePrefix)

mpiprint *, "Timing Calls"

Mflops = (145*24*nsublattice)*1.0e-6

mpiprint *, "FLIC timing", ncalls, meantime, mintime, maxtime
mpiprint *, "FLIC rate", ncalls, Mflops/meantime, Mflops/mintime, Mflops/maxtime

mpiprint *, "MultiCG", ncalls1, meantime1, mintime1, maxtime1

mpiprint *, "Overlap", ncalls2, meantime2, mintime2, maxtime2

mpiprint *, "Sign function convergence"
do i_ev=1,n_poles
   mpiprint *, i_ev, meaniter_i(i_ev), miniter_i(i_ev), maxiter_i(i_ev)
end do

call date_and_time(date=endDate, time=endTime)
mpiprint *, "Started on ", startDate, " at ", startTime
mpiprint *, "Finished on ", endDate, " at ", endTime

call FinaliseOverlap

call FinaliseMPI

end program FLICPropagator
```

FLICOverTree: Main program for the calculation of the tree-level FLIC overlap propagator.

```
program FLICPropagator

  use FLICOverlap
  use OverlapPropagator
  implicit none

  character(len=256)  :: InputConfig, InputOverlapMode
  character(len=256)  :: OutputPrefix, TimeSlicePrefix
  integer :: ix,iy,iz,it, n_mu_input, ic

  integer, parameter :: n_mu = 15
  real(DP), dimension(n_mu) :: mu_i
  real(DP) :: tolerance
  integer :: i_null, i_ev, j_null, j_ev, m_ev, m_dummy, i_lambda, j_lambda, i_mu

  character(len=20) :: startDate, startTime, endDate, endTime

  call InitialiseMPI

  call InitShadowGaugeField
  call InitShadowFermionField

  call InitTimer

  startDate = ""
  startTime = ""

  endDate = ""
  endTime = ""

  mpiprint *, "Enter the output file prefix:"
  read(*,'(A256)') OutputPrefix
  mpiprint *, OutputPrefix

  mpiprint *, "Enter the timeslice file prefix:"
  read(*,'(A256)') TimeSlicePrefix
  mpiprint *, TimeSlicePrefix

  mpiprint *, "FLIC parameters"
  mpiprint *, "Wilson mass parameter"
  read(*,*) m_f
  mpiprint *, "m_w=", m_f

  mpiprint *, "Clover term c_sw"
  read(*,*) c_sw
  mpiprint *, "c_sw=", c_sw

  mpiprint *, "Smearing fraction"
  read(*,*) alpha_smear
  mpiprint *, "alpha_smear=", alpha_smear

  mpiprint *, "Smearing sweeps"
  read(*,*) ape_sweeps
  mpiprint *, "ape_sweeps", ape_sweeps

  mpiprint *, "Propagator parameters"
  mpiprint *, "Source position (ix,iy,iz,it)"
  read(*,*) ix
  read(*,*) iy
  read(*,*) iz
  read(*,*) it
  read(*,*) js_start, js_end
  read(*,*) jc_start, jc_end
  mpiprint *, " ix=",ix," iy=",iy," iz=",iz," it=",it

  mpiprint *, "Source type"
  read(*,*) SmearedSource
  read(*,*) alpha_sm
  read(*,*) n_smear

  mpiprint *, "Boundary Conditions (bct)"
  read(*,*) bct
  mpiprint *, " bct=",bct

  mpiprint *, "Number of Fermion masses"
  read(*,*) n_mu_input
  if (n_mu_input /= n_mu ) then
     mpiprint *, 'Propagator: failed: Mismatch in nmu'
     stop
  end if

  mpiprint *, "Fermion masses, mu_i"
  do i_mu=1,n_mu
     read(*,*) mu_i(i_mu)
  end do
  mpiprint *, mu_i

  mpiprint *, "Enter the desired solution precision:"
```

```
read(*,*) tolerance
mpiprint *, tolerance

mpiprint *, "Starting time"
call date_and_time(date=startDate , time=startTime)

mpiprint *, "Reading Gauge Field"
U_xd = unit_matrix
UFL_xd = unit_matrix

u0_bar = 1.0d0
u0fl_bar = 1.0d0

mpiprint *, "uzero", u0_bar, u0fl_bar

ZoloFile = "zolotarev.dat"

n_lambda = 0
project_eigenspace = .false.

allocate(epsilon_lambda_i(n_lambda))
allocate(v_lambda(nx,ny,nz,nt,ns,n_lambda))

if ( i_am_root ) then
   open(110, file=ZoloFile, status = "old", form = "formatted", action = "read")
   read(110,*) n_poles
   close(110)
end if

call BroadCastInteger(n_poles,mpi_root_rank)

allocate(b_l(n_poles))
allocate(c_l(n_poles))

allocate(epsilon_iter_i(n_poles))
allocate(miniter_i(n_poles))
allocate(maxiter_i(n_poles))
allocate(meaniter_i(n_poles))

mpiprint *, "Calculating Operators"
call InitialiseFLICOverlap(m_f, c_sw, bct, U_xd, UFL_xd)

miniter_i = 1000
maxiter_i = 0
meaniter_i = 0.0

call CalculatePropagator(ix,iy,iz,it,n_mu,mu_i,tolerance,OutputPrefix,SmearedSource,alpha_sm,n_smear)

call OverPropToTimeSlice(n_mu,mu_i,OutputPrefix,TimeSlicePrefix)

mpiprint *, "Timing Calls"

Mflops = (145*24*nsublattice)*1.0e-6

mpiprint *, "FLIC timing", ncalls, meantime, mintime, maxtime
mpiprint *, "FLIC rate", ncalls, Mflops/meantime, Mflops/mintime, Mflops/maxtime

mpiprint *, "MultiCG", ncalls1, meantime1, mintime1, maxtime1

mpiprint *, "Overlap", ncalls2, meantime2, mintime2, maxtime2

mpiprint *, "Sign function convergence"
do i_ev=1,n_poles
   mpiprint *, i_ev, meaniter_i(i_ev), miniter_i(i_ev), maxiter_i(i_ev)
end do

call date_and_time(date=endDate, time=endTime)
mpiprint *, "Started on ", startDate, " at ", startTime
mpiprint *, "Finished on ", endDate, " at ", endTime

call FinaliseOverlap

call FinaliseMPI

end program FLICPropagator
```

A.6 Accelerated Ritz Algorithm Code

CmplxJacobiDiag: Implementation of a Jacobi-like routine for the exact diagonalisation of small Hermitian matrices.

```
module CmplxJacobiDiag

  use Kinds
  implicit none

  real(DP) :: JacobiPrecision = 0.1d0
  !Use Jacobi Precision as the precision_factor argument to the routines below
  !Recommended value range: not greater than 1.0d0 and not less than 1.0d-4
  !the smaller the value the more precise the routine

contains

  pure subroutine ComplexDiag(ndim, M_ij, V_ij, sweeps, precision_factor)

    integer, intent(in) :: ndim !Matrix dimensions
    complex(DC), dimension(ndim, ndim), intent(inout) :: M_ij !Matrix to be diagonalised
    complex(DC), dimension(ndim, ndim), intent(inout) :: V_ij !Matrix of eigenvectors
    integer, intent(out) :: sweeps
    real(DP), intent(in)  :: precision_factor

    real(DP), dimension(ndim, ndim) :: ReM, ImM !Matrix to be diagonalised
    real(DP), dimension(ndim, ndim) :: ReV, ImV !Matrix of eigenvectors

    ReM = real(M_ij)
    ImM = aimag(M_ij)
    ReV = real(V_ij)
    ImV = aimag(V_ij)

    call JacobiDiag(ndim, ReM, ImM, ReV, ImV, sweeps, precision_factor)

    M_ij = cmplx(ReM, ImM, dc)
    V_ij = cmplx(ReV, ImV, dc)

  end subroutine ComplexDiag

  pure subroutine JacobiDiag(ndim, ReM, ImM, ReV, ImV, sweeps, precision_factor)

    integer, intent(in) :: ndim !Matrix dimensions
    real(DP), dimension(ndim, ndim), intent(inout) :: ReM, ImM !Matrix to be diagonalised
    real(DP), dimension(ndim, ndim), intent(inout) :: ReV, ImV !Matrix of eigenvectors
    integer , intent(out) :: sweeps
    real(DP), intent(in)  :: precision_factor

    !Upon exit, ReM and ImM contain the real and imaginary parts of the eigenvalues as
    !their diagonal elements, ReV and ImV contain the real and imaginary parts of the
    !eigenvectors, and sweeps contains the number of Jacobi sweeps needed

    !Reference: Wilkinson & Reinsch, Handbook for Automatic Computation
    ! v. 2 Linear Algebra, II/17, by P.J. Eberlein

    real(DP) :: eps, tau
    real(DP) :: ImH, ReH, ModH, G
    real(DP) :: ReB, ImB, ReE, ImE, ReD, ImD.
    real(DP) :: isw, c, s, d, de, b, e, nd, root, root1, root2, eta
    real(DP) :: sig, sa, ca, sx, cx, cotx, cot2x, sin2a, cos2a, cb, ch, sb, sh
    real(DP) :: htan, nc, c1r, s1r, c2r, s2r, c1i, s1i, c2i, s2i
    real(DP) :: tik, tim, uik, uim, aki, ami, aik, aim, zik, zki, zim, zmi
    integer  :: i,j,k,m
    logical  :: mark
    logical, dimension(ndim,ndim) :: OffDiag !sum mask

    eps = tiny(1.0d0)
    mark = .false.

    !set V=identity matrix, set OffDiag mask
    ReV=0.0d0
    ImV=0.0d0
    OffDiag = .true.
    do i=1,ndim
       ReV(i,i) = 1.0d0
       OffDiag(i,i) = .false.
    end do

    sweeps = 0
    do
       !calculate tau=norm of off-diagonal elements
       tau = sum(abs(ReM), mask=OffDiag)+sum(abs(ImM), mask=OffDiag)

       !stopping criterion
       if (tau**2 < eps) exit
       if (mark) exit

       !begin Jacobi sweep
       sweeps = sweeps + 1

       mark = .true.
       KLoop: do k=1,ndim-1
          MLoop: do m=k+1,ndim
```

```
ReH = 0
ImH = 0
ModH = 0
G = 0
do i=1,ndim
   if ( (i/=k) .and. (i/=m) ) then
      ReH = ReH + ReM(k,i)*ReM(m,i) + ImM(k,i)*ImM(m,i) &
            & - ReM(i,k)*ReM(i,m) - ImM(i,k)*ImM(i,m)
      ImH = ImH + ImM(k,i)*ReM(m,i) - ReM(k,i)*ImM(m,i) &
            & - ReM(i,k)*ImM(i,m) + ImM(i,k)*ReM(i,m)
      ModH = ModH - ReM(i,k)**2 - ImM(i,k)**2 &
            & - ReM(m,i)**2 - ImM(m,i)**2 &
            & + ReM(i,m)**2 + ImM(i,m)**2 &
            & + ReM(k,i)**2 + ImM(k,i)**2
      G = G + ReM(i,k)**2 + ImM(i,k)**2 &
            & + ReM(m,i)**2 + ImM(m,i)**2 &
            & + ReM(i,m)**2 + ImM(i,m)**2 &
            & + ReM(k,i)**2 + ImM(k,i)**2
   end if
end do
ReB = ReM(k,m) + ReM(m,k)
ImB = ImM(k,m) + ImM(m,k)
ReE = ReM(k,m) - ReM(m,k)
ImE = ImM(k,m) - ImM(m,k)
ReD = ReM(k,k) - ReM(m,m)
ImD = ImM(k,k) - ImM(m,m)
if ( (ReB**2+ImE**2+ReD**2) >= (ImB**2+ReE**2+ImD**2) ) then
   isw = 1.0d0
   c = ReB
   s = ImE
   d = ReD
   de = ImD
   root2 = sqrt(ReB**2+ImE**2+ReD**2)
else
   isw = -1.0d0
   c = ImB
   s = -ReE
   d = ImD
   de = reD
   root2 = sqrt(ImB**2+ReE**2+ImD**2)
end if
root1 = sqrt(s**2 + c**2)
sig = sign(1.0d0,d)
sa = 0.0d0
ca = sign(1.0d0,c)
if (root1 < eps) then
   sx = 0.0d0
   sa = 0.0d0
   cx = 1.0d0
   ca = 1.0d0
   if (isw > 0.0d0) then
      e = ReE
      b = ImB
   else
      e = ImE
      b = -ReB
   end if
   nd = d**2 + de**2
else
   if (abs(s) > eps) then
      ca = c/root1
      sa = s/root1
   end if
   cot2x = d/root1
   cotx = cot2x + sig*sqrt(1.0d0 + cot2x**2)
   sx = sig/sqrt(1.0d0 + cotx**2)
   cx = sx*cotx
   !find rotated elements
   eta = (ReE*ReB+ImB*ImE)/root1
   nd = root2**2 + ((d*de+ReB*ImB-ReE*ImE)/root2)**2
   sin2a = 2.0d0*ca*sa
   cos2a = ca**2 - sa**2
   ReH = ReH*cx**2 - (ReH*cos2a+ImH*sin2a)*sx**2 - ca*(ModH*cx*sx)
   ImH = ImH*cx**2 + (ImH*cos2a-ReH*sin2a)*sx**2 - sa*(ModH*cx*sx)
   b = isw*(sig*(-root1*de+d*(ReB*ImB-ReE*ImE)/root1)/root2)*ca + eta*sa
   e = ca*eta - isw*(sig*(-root1*de+d*(ReB*ImB-ReE*ImE)/root1)/root2)*sa
end if
s = ReH - sig*root2*e
c = imH - sig*root2*b
root = sqrt(c**2+s**2)
if (root < eps) then
   cb = 1.0d0
   ch = 1.0d0
   sb = 0.0d0
   sh = 0.0d0
else
   cb = -c/root
   sb = s/root
```

```
            nc = (cb*b-e*sb)**2
            htan = root/(g+2.0d0*(nc+nd))
            ch = 1.0d0/sqrt(1.0d0 - htan**2)
            sh = ch*htan
          end if
          !prepare for transformation
          c1r = cx*ch - sx*sh*(sa*cb-sb*ca)
          c2r = cx*ch + sx*sh*(sa*cb-sb*ca)
          c1i = -sx*sh*(ca*cb+sa*sb)
          c2i = c1i
          s1r = sx*ch*ca - cx*sh*sb
          s2r = -sx*ch*ca - cx*sh*sb
          s1i = sx*ch*sa + cx*sh*cb
          s2i = sx*ch*sa - cx*sh*cb
          !decide whether to make transformation
          if ( sqrt(s1r**2+s1i**2) > eps .or. sqrt(s2r**2+s2i**2) > eps) then
             mark = .false.
             !transformation on left
             do i=1,ndim
                aki = ReM(k,i)
                ami = ReM(m,i)
                zki = ImM(k,i)
                zmi = ImM(m,i)
                ReM(k,i) = c1r*aki - c1i*zki + s1r*ami - s1i*zmi
                ImM(k,i) = c1r*zki + c1i*aki + s1r*zmi + s1i*ami
                ReM(m,i) = s2r*aki - s2i*zki + c2r*ami - c2i*zmi
                ImM(m,i) = s2r*zki + s2i*aki + c2r*zmi + c2i*ami
             end do
             !transformation on right
             do i=1,ndim
                aik = ReM(i,k)
                aim = ReM(i,m)
                zik = ImM(i,k)
                zim = ImM(i,m)
                ReM(i,k) = c2r*aik - c2i*zik - s2r*aim + s2i*zim
                ImM(i,k) = c2r*zik + c2i*aik - s2r*zim - s2i*aim
                ReM(i,m) = -s1r*aik + s1i*zik + c1r*aim - c1i*zim
                ImM(i,m) = -s1r*zik - s1i*aik + c1r*zim + c1i*aim
                tik = ReV(i,k)
                tim = ReV(i,m)
                uik = ImV(i,k)
                uim = ImV(i,m)
                ReV(i,k) = c2r*tik - c2i*uik - s2r*tim + s2i*uim
                ImV(i,k) = c2r*uik + c2i*tik - s2r*uim - s2i*tim
                ReV(i,m) = -s1r*tik + s1i*uik + c1r*tim - c1i*uim
                ImV(i,m) = -s1r*uik - s1i*tik + c1r*uim + c1i*tim
             end do

          end if
        end do MLoop
      end do KLoop
    end do

  end subroutine JacobiDiag

end module CmplxJacobiDiag
```

AccMinEVCG: Implementation of the accelerated version of the conjugate gradient eigenvector search algorithm.

```
module AccMinEVCG

  use ColourTypes
  use CmplxJacobiDiag
  use FermionField
  use MPIInterface
  implicit none


  !The only procedure needed to be called is MinEvSpectrum, after which lambda_i and v_i will contain
  !the lowest n_ev eigenvalues and their corresponding eigenvectors, respectively.

  logical :: VerboseAccCGEV = .true. !More or less output
  integer :: CGIter_Max = 100 !Maximum CG Iterations per cycle per eigenvalue
  integer :: StateRenormCount = 0
  integer :: RecalcDpsiCount = 0

contains

  subroutine MinEVSpectrum(n_ev, lambda_i, v_i, tolerance, CGIter_i,  RandomInitialv_i, MatrixOperate,
       n_dummy, lambda_scale )

    integer, intent(in) :: n_ev !number of eigenvectors to calculate
    type(colour_vector), dimension(nx,ny,nz,nt,ns,n_ev), target :: v_i !eigenvectors
    real(dp), dimension(n_ev) :: lambda_i !eigenvalues
    integer, dimension(n_ev) :: CGIter_i !Total eigenvector CG iterations
```

```
    real(dp) :: tolerance !Desired Precision of eigenvectors
    logical :: RandomInitialv_i !Use a random guess to start
    interface
       subroutine MatrixOperate(phi, Dphi)
         use ColourTypes
         type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi
         type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: Dphi
       end subroutine MatrixOperate
    end interface
    integer :: n_dummy !Do not demand convergence for the last n_dummy eigenvectors, n_dummy small
    real(dp) :: lambda_scale !Scale the operator by lambda_scale during MinEVSpectrum.
    !Note: Eigenvalues returned are unscaled.
    !lambda_scale = 1.0d0 : unscaled
    !lambda_scale = 0.0d0 : find only the zero eigenvalues
    !lambda_scale = -1.0d0 : highest eigenvalues found
    !lambda_scale = 1/||M|| : use normalised operator during search
    type(colour_vector), dimension(nx,ny,nz,nt,ns,n_ev), target :: Dv_i !operated upon eigenvectors


    real(dp) :: lambda
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: psi, Dpsi, grad_mu

#define v_lambda psi
#define Dv_lambda Dpsi

    integer :: iterations
    real(dp) :: deltaT, norm_grad_mu

    integer :: sweeps, k_ev, l_ev, j_ev

    type(colour_vector), dimension(:,:,:,:,:), pointer :: v_k, Dv_k, Dv_l, v_l

    complex(DC), dimension(n_ev,n_ev) :: M_kl !Sub-matrix to be diagonalised
    complex(DC), dimension(n_ev,n_ev) :: V_kl !Matrix of sub-eigenvectors

    integer :: JacobiSweeps !Diagonalisation sweeps

    integer :: i_ev !number of eigenvectors calculated so far
    integer :: CGCycles !Number of cycles

    logical, dimension(n_ev) :: converged_i !eigenvectors that have converged
    logical, dimension(n_ev) :: normg_i !||g|| < tolerance?
    logical, dimension(n_ev) :: temple_i !temple < tolerance?
    logical, dimension(n_ev) :: deltaC_i !deltacycle < tolerance?
    logical, dimension(n_ev) :: DeltaCycleExit_i !Did we improve by tau in a CG cycle?

    real(dp), dimension(n_ev) :: mu_i, mu1_i, mu2_i !ritz functional values from 0,1,2 previous
         diagonalizations.

    real(dp), dimension(n_ev) :: deltacycle_i !ritz functional values
    real(dp), dimension(n_ev) :: lower_i !lower bounds
    real(dp), dimension(n_ev) :: gamma_i !decreases
    logical, dimension(n_ev) :: UseTemple_i !temple < tolerance?
    logical, dimension(n_ev) :: UseDeltaCycle_i !Is delta_cycle safe?

    real(dp) :: tau = 0.1d0 !Determines the criterion for stopping the CG search and performing an
         Diagonalisation
    integer :: ix,iy,iz,it,is

    if ( (abs(lambda_scale) /= 1.0d0) .and. (abs(lambda_scale) /= 0.0d0) ) then
       tolerance = tolerance*abs(lambda_scale)
    end if

    VerboseAccCGEV = VerboseAccCGEV .and. i_am_root

    JacobiSweeps = 0
    CGCycles = 0
    i_ev = 0
    deltaT = 0.0d0

    CGIter_i=0
    converged_i = .false.
    normg_i = .false.
    temple_i = .false.
    deltaC_i = .false.
    DeltaCycleExit_i = .false.

    mu_i=0.0d0
    mu1_i=0.0d0
    mu2_i=0.0d0
    deltacycle_i=0.0d0
    lower_i=0.0d0
    UseTemple_i = .false.
    UseDeltaCycle_i = .false.

    if (.not. RandomInitialv_i ) then
       do k_ev=1,n_ev
          psi(1:nx,1:ny,1:nz,1:nt,:) = v_i(:,:,:,:,:,k_ev)
```

```
        call ScaledMatrixOperate(psi, Dpsi)
        Dv_i(:,:,:,:,:,k_ev)=Dpsi(1:nx,1:ny,1:nz,1:nt,:)
    end do

    do k_ev=1,n_ev; do l_ev=1,n_ev
        v_k => v_i(:,:,:,:,:,k_ev)
        Dv_l => Dv_i(:,:,:,:,:,l_ev)
        M_kl(k_ev,l_ev) = inner_product(v_k,Dv_l)
    end do; end do

    if ( VerboseAccCGEV ) print *, "Diagonalising M"
    call ComplexDiag(n_ev, M_kl, V_kl, sweeps, JacobiPrecision)

    JacobiSweeps = JacobiSweeps + sweeps
    if (VerboseAccCGEV ) print *, "Rotating Eigenvectors"
    call RotateEigenVectors(n_ev, M_kl, V_kl)

end if

do i_ev=0,n_ev-1
    call MinimumEV(lambda, v_lambda, Dv_lambda, tolerance, iterations, RandomInitialv_i )
    lambda_i(i_ev+1) = lambda
    v_i(:,:,:,:,:,i_ev+1) = v_lambda(1:nx,1:ny,1:nz,1:nt,:)
    Dv_i(:,:,:,:,:,i_ev+1) = Dv_lambda(1:nx,1:ny,1:nz,1:nt,:)
    CGIter_i(i_ev+1) = iterations

    if (VerboseAccCGEV ) print *, "i_ev=", i_ev+1
    if (VerboseAccCGEV ) print *, "lambda=", lambda
    if (VerboseAccCGEV ) print *, "iterations=", iterations
end do

CGCycles = 1

do
    if ( all(converged_i(1:n_ev-n_dummy)) ) exit

    if (VerboseAccCGEV ) print *, "Cycle #:", CGCycles
    if (VerboseAccCGEV ) print *, "Calculating M"

    if (CGCycles == 1) then
        mu1_i = lambda_i
    end if

    mu2_i = mu1_i
    mu1_i = lambda_i

    !Perform intermediate diagonalisation
    do k_ev=1,n_ev; do l_ev=1,n_ev
        v_k => v_i(:,:,:,:,:,k_ev)
        Dv_l => Dv_i(:,:,:,:,:,l_ev)
        M_kl(k_ev,l_ev) = inner_product(v_k,Dv_l)
    end do; end do

    if (VerboseAccCGEV ) print *, "Diagonalising M"
    call ComplexDiag(n_ev, M_kl, V_kl, sweeps, JacobiPrecision)

    JacobiSweeps = JacobiSweeps + sweeps
    if (VerboseAccCGEV ) print *, "Rotating Eigenvectors"
    call RotateEigenVectors(n_ev, M_kl, V_kl)

    !update the ritz functional values
    do k_ev=1,n_ev
        v_k => v_i(:,:,:,:,:,k_ev)
        Dv_k => Dv_i(:,:,:,:,:,k_ev)
        mu_i(k_ev) = real_inner_product(v_k,Dv_k)
    end do

    if (CGCycles > 1) then
        !Calculate delta cycle
        if ( intermediate .and. i_am_root ) print *, "mu  ", mu_i
        if ( intermediate .and. i_am_root ) print *, "mu1 ", mu1_i
        if ( intermediate .and. i_am_root ) print *, "mu2 ", mu2_i
        deltacycle_i = abs(mu2_i - mu_i)/(1.0d0-tau)
        UseDeltaCycle_i = ( mu2_i >= mu_i - tolerance ) .and. ( DeltaCycleExit_i ) .and. (CGCycles > 3)
        if ( intermediate .and. i_am_root ) print *, UseDeltaCycle_i
    end if

    !Determine whether the Temple Inequality is safe
    do k_ev=1,n_ev
        psi(1:nx,1:ny,1:nz,1:nt,:) = v_i(:,:,:,:,:,k_ev)
        Dpsi(1:nx,1:ny,1:nz,1:nt,:) = Dv_i(:,:,:,:,:,k_ev)

        if (k_ev<n_ev) then
            lower_i(k_ev) = minval(mu_i(k_ev+1:n_ev), mask = mu_i(k_ev+1:n_ev) - mu_i(k_ev) > tolerance)
        end if

        do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
            grad_mu(ix,iy,iz,it,is)%Cl = Dpsi(ix,iy,iz,it,is)%Cl - mu_i(k_ev)*psi(ix,iy,iz,it,is)%Cl
```

```
            end do; end do; end do; end do; end do

            norm_grad_mu = fermion_norm(grad_mu)

            UseTemple_i(k_ev) = abs(lower_i(k_ev)-mu_i(k_ev)) < norm_grad_mu
            if ( intermediate  .and. i_am_root ) print *, "Use Temple?", UseTemple_i(k_ev), &
                 & any(mu_i(k_ev+1:n_ev) - mu_i(k_ev) > tolerance), (CGCycles > 3 )
            UseTemple_i(k_ev) = any(mu_i(k_ev+1:n_ev) - mu_i(k_ev) > tolerance) .and. UseTemple_i(k_ev) .and
                 . (CGCycles > 3 )
         end do
         UseTemple_i(n_ev) = .false. !The temple inequality is not valid for the highest eigenvalue.

         do i_ev=0,n_ev-1
            if (VerboseAccCGEV ) print *, "Searching for Eigenvector ", i_ev+1
            call MinimumEV(lambda, v_lambda, Dv_lambda, tolerance, iterations, .false. )
            lambda_i(i_ev+1) = lambda
            v_i(:,:,:,:,:,i_ev+1) = v_lambda(1:nx,1:ny,1:nz,1:nt,:)
            Dv_i(:,:,:,:,:,i_ev+1) = Dv_lambda(1:nx,1:ny,1:nz,1:nt,:)
            CGIter_i(i_ev+1) = CGIter_i(i_ev+1)+iterations
            if (VerboseAccCGEV ) print *, "lambda=", lambda
            if (VerboseAccCGEV ) print *, "iterations=", iterations
         end do
         CGCycles = CGCycles + 1
      end do

      if (lambda_scale /= 1.0d0) then
         lambda_i = lambda_i / lambda_scale
      end if

      do i_ev=1,n_ev
         mpiprint *, "||g||", normg_i(i_ev), "temple", temple_i(i_ev), "delta", deltaC_i(i_ev)
      end do

      do i_ev=1,n_ev

         psi(1:nx,1:ny,1:nz,1:nt,:) = v_i(:,:,:,:,:,i_ev)
         Dpsi(1:nx,1:ny,1:nz,1:nt,:) = Dv_i(:,:,:,:,:,i_ev)

         do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
            grad_mu(ix,iy,iz,it,is)%Cl = Dpsi(ix,iy,iz,it,is)%Cl - &
                 & (lambda_scale*lambda_i(i_ev))*psi(ix,iy,iz,it,is)%Cl
         end do; end do; end do; end do; end do

         norm_grad_mu = fermion_norm(grad_mu)

         mpiprint *, "||g||", norm_grad_mu, "delta", deltacycle_i(i_ev)
      end do

#undef v_lambda
#undef Dv_lambda

   contains

      subroutine RotateEigenVectors(n_ev, M_kl, V_kl)

        implicit none

        integer, intent(in) :: n_ev
        complex(DC), dimension(n_ev,n_ev) :: M_kl !Sub-matrix to be diagonalised
        complex(DC), dimension(n_ev,n_ev) :: V_kl !Matrix of sub-eigenvectors

        type(colour_vector), dimension(nx,ny,nz,nt,ns,n_ev) :: vprime_i !temp eigenvectors

        real(dp) :: swap, norm_v
        real(dp), dimension(n_ev) :: swap_l
        integer :: k_ev, l_ev

        !Sort sub-eigenvectors into increasing eigenvalue order
        do k_ev=1,n_ev
           do l_ev=k_ev+1,n_ev
              if ( real(M_kl(l_ev,l_ev)) < real(M_kl(k_ev,k_ev)) ) then
                 swap = M_kl(k_ev,k_ev)
                 swap_l = V_kl(:,k_ev)
                 M_kl(k_ev,k_ev) = M_kl(l_ev,l_ev)
                 V_kl(:,k_ev) = V_kl(:,l_ev)
                 M_kl(l_ev,l_ev) = swap
                 V_kl(:,l_ev) = swap_l
              end if
           end do
        end do

        !Normalise sub-eigenvectors
        do l_ev=1,n_ev
           V_kl(:,l_ev) = V_kl(:,l_ev)/sqrt(sum(abs(V_kl(:,l_ev))**2 ))
        end do

        !Re-Normalise old eigenvectors
        do l_ev=1,n_ev
```

```
    v_l => v_i(:,:,:,:,:,l_ev)
    norm_v = fermion_norm(v_l)
    call normalise(v_l,norm_v)
    Dv_l => Dv_i(:,:,:,:,:,l_ev)
    call normalise(Dv_l,norm_v)
  end do

  !Rotate eigenvectors
  do l_ev=1,n_ev
     do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
        vprime_i(ix,iy,iz,it,is,l_ev)%Cl = 0.0d0
     end do; end do; end do; end do; end do
  end do

  do l_ev=1,n_ev; do k_ev=1,n_ev
     do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
        vprime_i(ix,iy,iz,it,is,l_ev)%Cl =  vprime_i(ix,iy,iz,it,is,l_ev)%Cl + &
             & V_kl(k_ev,l_ev)*v_i(ix,iy,iz,it,is,k_ev)%Cl
     end do; end do; end do; end do; end do
  end do; end do

  !update eigenvectors
  v_i = vprime_i

  !Rotate eigenvector derivatives
  do l_ev=1,n_ev
     do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
        vprime_i(ix,iy,iz,it,is,l_ev)%Cl = 0.0d0
     end do; end do; end do; end do; end do
  end do

  do l_ev=1,n_ev; do k_ev=1,n_ev
     do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
        vprime_i(ix,iy,iz,it,is,l_ev)%Cl =  vprime_i(ix,iy,iz,it,is,l_ev)%Cl + &
             & V_kl(k_ev,l_ev)*Dv_i(ix,iy,iz,it,is,k_ev)%Cl
     end do; end do; end do; end do; end do
  end do; end do

  Dv_i = vprime_i

end subroutine RotateEigenVectors

subroutine MinimumEV(lambda, v_lambda, Dv_lambda, tolerance, iterations, initialise_evectors)

  real(dp) :: lambda
  type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: v_lambda
  type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: Dv_lambda
  real(dp)  :: tolerance
  integer :: iterations
  logical, intent(in) :: initialise_evectors

  real(dp) :: mu_psi, mu_min !the ritz functional
  real(dp) :: normsq_grad_mu, normsq_grad_mu_old, normsq_grad_mu_0, norm_search, norm_psi
  real(dp) :: alpha, beta, delta, a1, a2, a3, pi, cos_delta, sin_delta, cos_theta, sin_theta,
       theta_min
  real(dp), parameter :: pio2=1.5707963267948966192313216916397514420985 8_dp

  integer :: j_ev

  type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: psi, Dpsi, grad_mu, search, Dsearch

  complex(DC) :: psidotDpsi, psidotDsearch, searchdotDpsi, searchdotDsearch, psidotsearch,
       grad_mudotsearch

  integer :: t_cycle
  real(dp) :: stop_cycle, delta_c
  logical :: RenormState, CheckState, ProjectState
  integer :: CheckFrequency, LastRenorm, ProjectFrequency

  if (initialise_evectors) then
     do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
        psi(ix,iy,iz,it,is)%Cl = sqrt(1.0d0/(nlattice*ns*nc))
     end do; end do; end do; end do; end do
  else
     psi(1:nx,1:ny,1:nz,1:nt,:) = v_i(:,:,:,:,:,i_ev+1)
  end if

  call ProjectVectorSpace(i_ev,v_i,psi)

  norm_psi = fermion_norm(psi)
  call normalise(psi,norm_psi)

  call ScaledMatrixOperate(psi, Dpsi)
  call ProjectVectorSpace(i_ev,v_i,Dpsi)

  mu_psi = real_inner_product(psi,Dpsi)
  do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
     grad_mu(ix,iy,iz,it,is)%Cl = Dpsi(ix,iy,iz,it,is)%Cl - mu_psi*psi(ix,iy,iz,it,is)%Cl
```

```
end do; end do; end do; end do; end do

normsq_grad_mu = fermion_normsq(grad_mu)
normsq_grad_mu_0 = normsq_grad_mu

search = grad_mu
norm_search = fermion_norm(search)
call ScaledMatrixOperate(search, Dsearch)
call ProjectVectorSpace(i_ev, v_i, Dsearch)

t_cycle = CGCycles + 1
stop_cycle = abs(lambda_scale)*tau**t_cycle
CheckFrequency = min(CgIter_Max/10, CGCycles + 10*(n_ev-i_ev)/n_ev + max(0,int(-log(normsq_grad_mu))
    ) )
ProjectFrequency = 10
RenormState = .true.
DeltaCycleExit_i(i_ev+1) = .false.
converged_i(i_ev+1) = .false.
normg_i(i_ev+1) = .false.
temple_i(i_ev+1) = .false.
deltaC_i(i_ev+1) = .false.

iterations = 0
do
   if ( sqrt(normsq_grad_mu) < tolerance) then
      converged_i(i_ev+1) = .true.
      normg_i(i_ev+1) = .true.
   end if

   if ( lambda_scale == 0.0d0 ) then
      if ( sqrt(normsq_grad_mu) < tau*mu_psi ) then
         converged_i(i_ev+1) = .true.
      end if
   end if

   if ( UseTemple_i(i_ev+1) .and. ( lower_i(i_ev+1) - mu_psi > 0 ) ) then
      if (normsq_grad_mu/(lower_i(i_ev+1) - mu_psi) < tolerance ) then
         converged_i(i_ev+1) = .true.
         temple_i(i_ev+1) = .true.
      end if
   end if

   if ( (CGCycles > 2) .and. UseDeltaCycle_i(i_ev+1) ) then
      if ( deltacycle_i(i_ev+1)*(normsq_grad_mu/normsq_grad_mu_0) < tolerance ) then
         converged_i(i_ev+1) = .true.
         deltaC_i(i_ev+1) = .true.
         DeltaCycleExit_i(i_ev+1) = .true.
      end if
   end if

   delta_c = deltacycle_i(i_ev+1)*(normsq_grad_mu/normsq_grad_mu_0)

   if (converged_i(i_ev+1)) exit

   if ( (iterations >= CGIter_Max) .and. (normsq_grad_mu < normsq_grad_mu_0) ) exit

   if ( iterations >= 3*CGIter_Max/2 ) exit

   if ( (CGCycles <= 3) .and. sqrt(abs(normsq_grad_mu)) < stop_cycle) exit

   if (CgCycles > 1 .and. iterations > 10 ) then
      if ( normsq_grad_mu/normsq_grad_mu_0 < tau) then
         DeltaCycleExit_i(i_ev+1) = .true.
         exit
      end if
   end if

   iterations = iterations + 1

   CheckState = .false.
   if (RenormState) then
      CheckState = .true.
      LastRenorm = 1
   else
      LastRenorm = LastRenorm + 1
   end if
   if (LastRenorm >= CheckFrequency) then
      CheckState = .true.
      LastRenorm = 1
   end if

   if (RenormState) StateRenormCount = StateRenormCount + 1
   RenormState = .false.

   ProjectState = (modulo(iterations, ProjectFrequency)==0)

   normsq_grad_mu_old = normsq_grad_mu
   psidotDpsi = real_inner_product(psi, Dpsi)
```

```
searchdotDsearch = real_inner_product(search, Dsearch)
searchdotDsearch = searchdotDsearch/(norm_search**2)

a1 = 0.5d0*(psidotDpsi + searchdotDsearch)
a2 = 0.5d0*(psidotDpsi - searchdotDsearch)
a3 = normsq_grad_mu/norm_search
alpha = sqrt( a2**2 + a3**2)

cos_delta = a2/alpha
sin_delta = a3/alpha

if (cos_delta <= 0.0d0 ) then
   cos_theta = sqrt(0.5*(1.0-cos_delta))
   sin_theta = -0.5*sin_delta/cos_theta
else
   sin_theta = -sqrt(0.5*(1.0+cos_delta))
   cos_theta = -0.5*sin_delta/sin_theta
end if

do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
   psi(ix,iy,iz,it,is)%Cl = cos_theta*psi(ix,iy,iz,it,is)%Cl + &
        & (sin_theta/norm_search)*search(ix,iy,iz,it,is)%Cl
end do; end do; end do; end do; end do

if (CheckState) then
   call ProjectVectorSpace(i_ev, v_i, psi)

   norm_psi = fermion_norm(psi)
   if (abs(norm_psi -1.0d0)> tolerance) then
      call normalise(psi, norm_psi)
      call ScaledMatrixOperate(psi, Dpsi)
      RecalcDpsiCount = RecalcDpsiCount + 1
      RenormState = .true.
   else
      do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
         Dpsi(ix,iy,iz,it,is)%Cl = cos_theta*Dpsi(ix,iy,iz,it,is)%Cl + &
              & (sin_theta/norm_search)*Dsearch(ix,iy,iz,it,is)%Cl
      end do; end do; end do; end do; end do
   end if
   call ProjectVectorSpace(i_ev, v_i, Dpsi)
else
   do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
      Dpsi(ix,iy,iz,it,is)%Cl = cos_theta*Dpsi(ix,iy,iz,it,is)%Cl + &
           & (sin_theta/norm_search)*Dsearch(ix,iy,iz,it,is)%Cl
   end do; end do; end do; end do; end do
end if

mu_psi = mu_psi - 2*alpha*(sin_theta**2)

if (CheckState) then
   mu_min = real_inner_product(psi, Dpsi)
   if ( abs(mu_psi - mu_min)/max(1.0d0, mu_psi) > tolerance ) then
      mu_psi = mu_min
      RenormState = .true.
   end if
end if

do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
   grad_mu(ix,iy,iz,it,is)%Cl = Dpsi(ix,iy,iz,it,is)%Cl - mu_psi*psi(ix,iy,iz,it,is)%Cl
end do; end do; end do; end do; end do

normsq_grad_mu = fermion_normsq(grad_mu)

if (CheckState) then
   grad_mudotsearch = inner_product(grad_mu, search)
   if (abs(grad_mudotsearch)/max(norm_search,1.0d0) > tolerance) then
      call orthogonalise(search, grad_mu, grad_mudotsearch)
      RenormState = .true.
   end if
end if

beta = cos_theta*(normsq_grad_mu/normsq_grad_mu_old)
psidotsearch = inner_product(psi, search)

do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
   search(ix,iy,iz,it,is)%Cl = grad_mu(ix,iy,iz,it,is)%Cl +  beta*( search(ix,iy,iz,it,is)%Cl - &
        & psi(ix,iy,iz,it,is)%Cl*psidotsearch)
end do; end do; end do; end do; end do

if (beta > 100) then
   norm_search = fermion_norm(search)
   call normalise(search, norm_search)
   psidotsearch = inner_product(psi, search)

   !searchdotgrad_mu = inner_product(grad_mu, search-grad_mu)
   grad_mudotsearch = inner_product(grad_mu, search)
   grad_mudotsearch = (grad_mudotsearch - normsq_grad_mu)
```

```
        do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
           search(ix,iy,iz,it,is)%Cl = search(ix,iy,iz,it,is)%Cl - psidotsearch*psi(ix,iy,iz,it,is)%Cl
                - &
                & (grad_mudotsearch/normsq_grad_mu)*grad_mu(ix,iy,iz,it,is)%Cl
        end do; end do; end do; end do; end do
     end if

     if (CheckState) then
        psidotsearch = inner_product(psi,search)
        if (abs(psidotsearch)/max(norm_search,1.0d0) > tolerance) then
           call orthogonalise(search,psi,psidotsearch)
           RenormState = .true.
        end if
     end if

     norm_search = fermion_norm(search)
     if (spacing(norm_search) > tolerance ) then
        call normalise(search,norm_search)
        psidotsearch = inner_product(psi,search)
        !searchdotgrad_mu = inner_product(grad_mu,search-grad_mu)
        grad_mudotsearch = inner_product(grad_mu,search)
        grad_mudotsearch = (grad_mudotsearch - normsq_grad_mu)

        do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
           search(ix,iy,iz,it,is)%Cl = search(ix,iy,iz,it,is)%Cl - psidotsearch*psi(ix,iy,iz,it,is)%Cl
                - &
                & (grad_mudotsearch/normsq_grad_mu)*grad_mu(ix,iy,iz,it,is)%Cl
        end do; end do; end do; end do; end do
     end if

     call ScaledMatrixOperate(search, Dsearch)
     if (CheckState) then
        call ProjectVectorSpace(i_ev,v_i,Dsearch)
     end if

  end do

  call ScaledMatrixOperate(psi,Dpsi)

  v_lambda = psi
  Dv_lambda = Dpsi
  lambda = mu_psi

  return

end subroutine MinimumEV

subroutine ScaledMatrixOperate(phi, Dphi)

  type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi
  type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: Dphi

  call MatrixOperate(phi, Dphi)

  if ( (lambda_scale /= 1.0d0) .and. (lambda_scale /= 0.0d0) ) then
     do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
        Dphi(ix,iy,iz,it,is)%Cl = lambda_scale * Dphi(ix,iy,iz,it,is)%Cl
     end do; end do; end do; end do; end do
  end if

end subroutine ScaledMatrixOperate

end subroutine MinEVSpectrum

subroutine DiagonaliseBasis(n_ev, v_i, vprime_i, lambdaprime_i, MatrixOperate)

  !Given a set of eigenvectors v_i of one operator (eg. H^2), diagonalise another operator (eg. H)
  !on the span of v_i, and return the new eigenvectors vprime_i

  integer, intent(in) :: n_ev !number of eigenvectors to calculate
  real(dp), dimension(n_ev) :: lambda_i, lambdaprime_i !eigenvalues
  type(colour_vector), dimension(nx,ny,nz,nt,ns,n_ev), target :: v_i !old eigenvectors
  type(colour_vector), dimension(nx,ny,nz,nt,ns,n_ev) :: vprime_i !new eigenvectors

  type(colour_vector), dimension(:,:,:,:,:), pointer :: v_k, Dv_l
  type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi, Dphi

  interface
     subroutine MatrixOperate(phi, Dphi)
       use ColourTypes
       type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi
       type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: Dphi
     end subroutine MatrixOperate
  end interface

  complex(DC), dimension(n_ev,n_ev) :: M_kl !Sub-matrix to be diagonalised
  complex(DC), dimension(n_ev,n_ev) :: V_kl !Matrix of sub-eigenvectors
  type(colour_vector), dimension(nx,ny,nz,nt,ns,n_ev), target :: Dv_i
```

```
    real(dp) :: swap, norm_v
    real(dp), dimension(n_ev) :: swap_l
    integer :: k_ev, l_ev, sweeps
    integer :: ix,iy,iz,it,is

    do k_ev=1,n_ev
       phi(1:nx,1:ny,1:nz,1:nt,:) = v_i(:,:,:,:,:,k_ev)
       call MatrixOperate(phi,Dphi)
       Dv_i(:,:,:,:,:,k_ev) = Dphi(1:nx,1:ny,1:nz,1:nt,:)
    end do

    do k_ev=1,n_ev; do l_ev=1,n_ev
       v_k => v_i(:,:,:,:,:,k_ev)
       Dv_l => Dv_i(:,:,:,:,:,l_ev)
       M_kl(k_ev,l_ev) = inner_product(v_k,Dv_l)
    end do; end do

    call ComplexDiag(n_ev, M_kl, V_kl, sweeps, JacobiPrecision)

    do l_ev=1,n_ev
       lambdaprime_i(l_ev) = M_kl(l_ev,l_ev)
    end do

    if (VerboseAccCGEV) then
       mpiprint *, "nu_i", lambdaprime_i
    end if

    !Normalise sub-eigenvectors
    do l_ev=1,n_ev
       V_kl(:,l_ev) = V_kl(:,l_ev)/sqrt(sum(abs(V_kl(:,l_ev))**2 ))
    end do

    !Rotate eigenvectors and their derivatives
    do l_ev=1,n_ev
       do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
          vprime_i(ix,iy,iz,it,is,l_ev)%Cl = 0.0d0
       end do; end do; end do; end do; end do
    end do

    do l_ev=1,n_ev; do k_ev=1,n_ev
       do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
          vprime_i(ix,iy,iz,it,is,l_ev)%Cl = vprime_i(ix,iy,iz,it,is,l_ev)%Cl + &
               & V_kl(k_ev,l_ev)*v_i(ix,iy,iz,it,is,k_ev)%Cl
       end do; end do; end do; end do; end do
    end do; end do

  end subroutine DiagonaliseBasis

end module AccMinEVCG
```

CGEVSpectrum: Main program for the finding of eigenvectors.

```
program CGEVSpectrum

  use GaugeField
  use AccMinEVCG
  use ChiralFLICOperator
  use Timer
  implicit none

  !timing
  character(len=8) :: startDate, endDate
  character(len=10) :: startTime, endTime

  character(len=256) :: InputConfig, OutputFile, InputLoMode, DataFile
  character(len=3) :: IFig

  type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: psi, Dpsi, grad_mu

  integer :: n_ev !number of eigenvectors to calculate
  type(colour_vector), dimension(:,:,:,:,:,:), allocatable :: v_i !eigenvectors

  real(DP), dimension(:), allocatable :: lambda_i, sqrt_lambda_i, normsqg_i !eigenvalues
  integer, dimension(:), allocatable :: CGIter_i !Total eigenvector CG iterations

  real(DP) :: tolerance = 1.0d-8, lambda_scale
  integer :: i_ev, j_ev, n_dummy = 1, m_ev
  real(DP) :: m_temp, delta_lambda, norm_psi, sum_temp
  real(DP) :: lambda_min, lambda_max, sqrt_lambda, m_w, tol_in, lower, delta_t
  logical :: RandomInitial
  integer :: ix,iy,iz,it,is

  integer :: t0, t1, i0
  complex(DC) :: psidotDpsi

  call InitialiseMPI
```

```
call InitShadowGaugeField
call InitShadowFermionField

call InitTimer

mpiprint *, "Conjugate Gradient eigenvalue routine"
mpiprint *, "Enter the gauge configuration file to read from:"
read(*,'(A256)') InputConfig
mpiprint *, InputConfig

mpiprint *, "Use a random initial guess:"
read(*,*) RandomInitial
mpiprint *, RandomInitial

mpiprint *, "Enter the low eigenmode file to read from:"
read(*,'(A256)') InputLoMode
mpiprint *, InputLoMode

mpiprint *, "Enter the file to write to:"
read(*,'(A256)') OutputFile
mpiprint *, OutputFile

mpiprint *, "Enter the data file to write to:"
read(*,'(A256)') DataFile
mpiprint *, DataFile

mpiprint *, "Enter the configuration no:"
read(*,'(A3)') IFig
mpiprint *, IFig

mpiprint *, "Enter the desired number of eigenvectors:"
read(*,*) n_ev
mpiprint *, n_ev

mpiprint *, "Enter the scale factor (1=low modes,-1=hi modes):"
read(*,*) lambda_scale
mpiprint *, lambda_scale

mpiprint *, "Enter the desired eigenvalue precision:"
read(*,*) tolerance
mpiprint *, tolerance

mpiprint *, "Enter the fermion mass:"
read(*,*) m_f
mpiprint *, m_f

mpiprint *, "Enter the smearing fraction:"
read(*,*) alpha_smear
mpiprint *, alpha_smear

mpiprint *, "Enter the number of APE sweeps:"
read(*,*) ape_sweeps
mpiprint *, ape_sweeps

mpiprint *, "Enter the time boundary condition:"
read(*,*) bct
mpiprint *, bct

mpiprint *, "Starting time"
call date_and_time(date=startDate, time=startTime)

allocate(v_i(nx,ny,nz,nt,ns,n_ev))
allocate(lambda_i(n_ev))
allocate(sqrt_lambda_i(n_ev))
allocate(normsqg_i(n_ev))
allocate(CGIter_i(n_ev))

mpiprint *, "Reading Gauge Field"
call ReadGaugeField(InputConfig, U_xd)

mpiprint *, "Calculating Operators"

if ( uzero /= 1.0d0 ) then
   u0_bar = uzero
else
   call GetUZero(U_xd,u0_bar)
end if

call APESmearLinks(U_xd,UFL_xd,alpha_smear,ape_sweeps)
call GetUZero(UFL_xd,u0fl_bar)

mpiprint *, u0_bar, u0fl_bar, m_f, c_sw, bct
call InitialiseFLICOperator(U_xd,UFL_xd,u0_bar,u0fl_bar,m_f, c_sw, bct)

if ( .not. RandomInitial ) then
   call ReadEigenspace(n_ev,n_dummy,lambda_i,v_i,tolerance,m_w,bct,InputLoMode)
end if
```

```
mpiprint *, "Calculating Eigenvalue Spectrum"
call MinEVSpectrum(n_ev, lambda_i, v_i, tolerance, CGIter_i, RandomInitial, SqFLICOperate, n_dummy,
     lambda_scale )

mpiprint *, "# CG Iterations", sum(CGIter_i), RecalcDpsiCount

mpiprint *, " i_ev "," lambda_i ", " iter ", " norm_g delta_t "

do i_ev=1,n_ev
   psi(1:nx,1:ny,1:nz,1:nt,:) = v_i(:,:,:,:,:,i_ev)
   call SqFLICOperate(psi,Dpsi)
   sqrt_lambda_i(i_ev) = real_inner_product(psi,Dpsi)
   do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
      grad_mu(ix,iy,iz,it,is)%Cl = Dpsi(ix,iy,iz,it,is)%Cl - sqrt_lambda_i(i_ev)*psi(ix,iy,iz,it,is)%Cl
   end do; end do; end do; end do; end do
   normsqg_i(i_ev) = fermion_normsq(grad_mu)
end do

do i_ev=1,n_ev
   lower = minval(sqrt_lambda_i(i_ev+1:n_ev), mask = abs(sqrt_lambda_i(i_ev+1:n_ev) - sqrt_lambda_i(i_ev
        )) > tolerance )
   if (i_ev == n_ev) lower = 0.0d0
   delta_t = normsqg_i(i_ev)/(lower-sqrt_lambda_i(i_ev))
   mpiprint '(I4,F22.17,I8,F15.10,F15.10 )', i_ev, sqrt_lambda_i(i_ev), CGIter_i(i_ev), &
        & sqrt(normsqg_i(i_ev)), delta_t
end do

mpiprint *, " i_ev "," sqrt(lambda_i)", " iter ", " norm_g delta_t "

do i_ev=1,n_ev
   psi(1:nx,1:ny,1:nz,1:nt,:) = v_i(:,:,:,:,:,i_ev)
   call FLICOperate(psi,Dpsi)
   sqrt_lambda_i(i_ev) = real_inner_product(psi,Dpsi)
   do is=1,ns; do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
      grad_mu(ix,iy,iz,it,is)%Cl = Dpsi(ix,iy,iz,it,is)%Cl - sqrt_lambda_i(i_ev)*psi(ix,iy,iz,it,is)%Cl
   end do; end do; end do; end do; end do
   normsqg_i(i_ev) = fermion_normsq(grad_mu)
end do

do i_ev=1,n_ev
   lower = minval(sqrt_lambda_i(i_ev+1:n_ev), mask = abs(sqrt_lambda_i(i_ev+1:n_ev) - sqrt_lambda_i(i_ev
        )) > sqrt(tolerance))
   if (i_ev == n_ev) lower = 0.0d0
   delta_t = normsqg_i(i_ev)/(lower-sqrt_lambda_i(i_ev))
   mpiprint '(I4,F22.17,I8,F15.10,F15.10 )', i_ev, sqrt_lambda_i(i_ev), CGIter_i(i_ev),&
        & sqrt(normsqg_i(i_ev)), delta_t
end do

call WriteEigenspace(n_ev,n_dummy,sqrt_lambda_i,v_i,tolerance,m_f,bct,OutputFile)

if ( i_am_root ) then
   open (200, file=DataFile, status = "unknown", form = "formatted", action = "write", position = "
        append" )
   write (200,'(A,F8.4,3F12.7)') IFig, m_f, sqrt_lambda_i(max(n_ev-2,1):n_ev)
   close(200)
end if

Mflops = (145*24*nsublattice)*1.0e-6

mpiprint *, "FLIC timing", ncalls, meantime, mintime, maxtime
mpiprint *, "FLIC rate", ncalls, Mflops/meantime, Mflops/mintime, Mflops/maxtime

call date_and_time(date=endDate, time=endTime)
mpiprint *, "Started on ", startDate, " at ", startTime
mpiprint *, "Finished on ", endDate, " at ", endTime

deallocate(v_i)
deallocate(lambda_i)
deallocate(sqrt_lambda_i)
deallocate(normsqg_i)
deallocate(CGIter_i)

call FinaliseMPI

end program CGEVSpectrum
```

A.7 Hybrid Monte Carlo Code

TensorAlgebra: Provides efficient implementations of inner and outer products.

```
module TensorAlgebra

  use ColourTypes
  implicit none

contains

  subroutine StarInnerProduct(LinnerR, left, right)

    complex(dc) :: LinnerR
    type(colour_matrix) :: left, right

    LinnerR = sum(left%Cl(:,:)*transpose(right%Cl(:,:))) ! *unroll*

  end subroutine StarInnerProduct

  subroutine AddVectorOuterProduct(LouterR, left, right)

    type(colour_matrix)  :: LouterR
    type(colour_vector), dimension(ns) :: left, right

    LouterR%Cl(ic,jc) = LouterR%Cl(ic,jc) + sum(left(:)%Cl(ic)*right(:)%Cl(jc)) ! *unroll*

  end subroutine AddVectorOuterProduct

  subroutine SubVectorOuterProduct(LouterR, left, right)

    type(colour_matrix)  :: LouterR
    type(colour_vector), dimension(ns) :: left, right

    LouterR%Cl(ic,jc) = LouterR%Cl(ic,jc) - sum(left(:)%Cl(ic)*right(:)%Cl(jc)) ! *unroll*

  end subroutine SubVectorOuterProduct

end module TensorAlgebra
```

GaugeAction: Implementation of the gauge field component of the equations of motion, for both standard and improved gauge actions.

```
module GaugeAction

  use MatrixAlgebra
  use MPIInterface
  use GaugeField
  implicit none

  type(colour_matrix), dimension(nxss,nyss,nzss,ntss,nd) :: U_xdss

  integer  :: ActionType

  integer, parameter :: wilson_glue = 0, twoloop_glue = 1, improved_glue = 2, dbw2_glue = 42

contains

  subroutine ActionParameters(beta,c_R,beta_eff)

    real(dp) :: beta
    real(dp) :: beta_eff, c_R

    select case (actiontype)
    case (wilson_glue)
       beta_eff = beta
       c_R = 0.0d0
    case (twoloop_glue)
       beta_eff = (5.0d0/3.0d0)*beta
       c_R = -1.0/20.0d0
    case (improved_glue)
       beta_eff = (5.0d0/3.0d0)*beta
       c_R = -1.0/(20.0d0*u0_bar**2)
    case (dbw2_glue)
       beta_eff = beta
       c_R = -1.4067d0/12.2536d0
    end select

  end subroutine ActionParameters

  function S_gauge(U_xd,beta)

    type(colour_matrix), dimension(nxss,nyss,nzss,ntss,nd) :: U_xd
    real(DP) :: S_gauge, beta

    real(dp) :: c_R, beta_eff, beta_11, beta_12, beta_21, s_gpp
    integer :: mu,nu

    call ActionParameters(beta,c_R,beta_eff)
```

```
    beta_11 = beta_eff/3.0d0
    beta_12 = beta_eff*c_R/3.0d0
    beta_21 = beta_eff*c_R/3.0d0

    S_gpp = 0.0d0

    do mu=1,nd
       do nu=mu+1,nd
          call GetAction(S_gpp,U_xd,mu,nu,beta_11,beta_12,beta_21)
       end do
    end do

    S_gpp = beta_eff*(1.0d0 + 2.0d0*c_R)*nsublattice*nd*(nd-1)/2.0d0 - S_gpp

    call AllSumReal(S_gpp,S_gauge)

  end function S_gauge

  subroutine GetdS_gbydU(dS_gbydU,U_xd,beta)

    type(colour_matrix), dimension(nxp,nyp,nzp,ntp,nd) :: dS_gbydU
    type(colour_matrix), dimension(nxss,nyss,nzss,ntss,nd) :: U_xd

    real(dp) :: beta, c_R, beta_11, beta_12, beta_21, beta_eff
    integer :: mu,nu
    integer :: ix,iy,iz,it

    call ActionParameters(beta,c_R,beta_eff)

    beta_11 = beta_eff/6.0d0
    beta_12 = beta_eff*c_R/6.0d0
    beta_21 = beta_eff*c_R/6.0d0

    do mu=1,nd
       do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
          dS_gbydU(ix,iy,iz,it,mu)%Cl = 0.0d0
       end do; end do; end do; end do
    end do

    do mu=1,nd
       do nu=mu+1,nd
          call GetStaples(dS_gbydU,U_xd,mu,nu,beta_11,beta_12,beta_21)
       end do
    end do

  end subroutine GetdS_gbydU

  subroutine GetAction(S_g,U_xd,mu,nu,beta_11,beta_12,beta_21)

    type(colour_matrix), dimension(nxss,nyss,nzss,ntss,nd) :: U_xd
    integer :: mu, nu
    real(dp) :: S_g, beta_11, beta_12, beta_21, S_11, S_12, S_21

    type(colour_matrix) :: U2,UmuUnu,UmudagUnudag,UnuUmudag,T_munu

    integer, dimension(nd) :: dmu, dnu
    integer :: ix,iy,iz,it
    integer :: jx,jy,jz,jt
    integer :: kx,ky,kz,kt
    integer :: lx,ly,lz,lt
    integer :: ax,ay,az,at
    integer :: bx,by,bz,bt

#define U_mux U_xd(ix,iy,iz,it,mu)
#define U_nux U_xd(ix,iy,iz,it,nu)

#define U_muxpmu U_xd(jx,jy,jz,jt,mu)
#define U_nuxpmu U_xd(jx,jy,jz,jt,nu)

#define U_muxpnu U_xd(kx,ky,kz,kt,mu)
#define U_nuxpnu U_xd(kx,ky,kz,kt,nu)

#define U_muxpmupnu U_xd(lx,ly,lz,lt,mu)
#define U_nuxpmupnu U_xd(lx,ly,lz,lt,nu)

#define U_nuxp2mu U_xd(ax,ay,az,at,nu)
#define U_muxp2nu U_xd(bx,by,bz,bt,mu)

    dmu = 0
    dnu = 0

    dmu(mu) = 1
    dnu(nu) = 1

    do it=1,nt
       jt = mapt(it + dmu(4))
       kt = mapt(it + dnu(4))
       lt = mapt(it + dmu(4) + dnu(4))
```

```
      at = mapt( it + 2*dmu(4))
      bt = mapt( it + 2*dnu(4))
      do iz=1,nz
         jz = mapz( iz + dmu(3))
         kz = mapz( iz + dnu(3))
         lz = mapz( iz + dmu(3) + dnu(3))
         az = mapz( iz + 2*dmu(3))
         bz = mapz( iz + 2*dnu(3))
         do iy=1,ny
            jy = mapy( iy + dmu(2))
            ky = mapy( iy + dnu(2))
            ly = mapy( iy + dmu(2) + dnu(2))
            ay = mapy( iy + 2*dmu(2))
            by = mapy( iy + 2*dnu(2))
            do ix=1,nx
               jx = mapx( ix + dmu(1))
               kx = mapx( ix + dnu(1))
               lx = mapx( ix + dmu(1) + dnu(1))
               ax = mapx( ix + 2*dmu(1))
               bx = mapx( ix + 2*dnu(1))

               call MultiplyMatMat (UmuUnu, U_mux, U_nuxpmu)
               call MultiplyMatDagMatDag (UmudagUnudag, U_muxpnu, U_nux)
               call RealTraceMultMatMat(S_11, UmuUnu, UmudagUnudag)

               call MultiplyMatMat (U2, U_mux, U_muxpmu)
               call MultiplyMatMatDag(UnuUmudag, U_nuxp2mu, U_muxpmupnu)
               call MultiplyMatMat (T_munu, U2, UnuUmudag)
               call RealTraceMultMatMat(S_21, T_munu, UmudagUnudag)

               call MultiplyMatMat (U2, U_nux, U_nuxpnu)
               call MultiplyMatMatDag(UnuUmudag, U_nuxpmupnu, U_muxp2nu)
               call MultiplyMatMatDag(T_munu, UnuUmudag, U2)
               call RealTraceMultMatMat(S_12, UmuUnu, T_munu)

               S_g = S_g + beta_11*S_11 + beta_21*S_21 + beta_12*S_12

            end do
         end do
      enddo
   end do

#undef U_mu
#undef U_nu

#undef U_muxpmu
#undef U_nuxpmu

#undef U_muxpnu
#undef U_nuxpnu

#undef U_nuxpmupnu
#undef U_muxpmupnu

#undef U_nuxp2mu
#undef U_muxp2nu

  end subroutine GetAction

  subroutine GetStaples(dS_gbydU, U_xd, mu, nu, beta_11, beta_12, beta_21)

    type(colour_matrix), dimension(nxp,nyp,nzp,ntp,nd) :: dS_gbydU
    type(colour_matrix), dimension(nxss,nyss,nzss,ntss,nd) :: U_xd
    integer :: mu, nu
    real(dp) :: S_g, beta_11, beta_12, beta_21

    type(colour_matrix) :: U2, UmuUnu, UnuUmu, UmudagUnudag, UnuUmudag, UnudagUmudag, UnudagUmu, UmuUnudag
    type(colour_matrix) :: T_xp, T_xm, R_xp, R_xm, V_muxp, V_muxm, V_nuxp, V_nuxm

    integer, dimension(nd) :: dmu, dnu
    integer :: ix,iy,iz,it
    integer :: jx,jy,jz,jt
    integer :: kx,ky,kz,kt
    integer :: lx,ly,lz,lt
    integer :: mx,my,mz,mt
    integer :: ax,ay,az,at
    integer :: bx,by,bz,bt
    integer :: cx,cy,cz,ct
    integer :: dx,dy,dz,dt
    integer :: ex,ey,ez,et
    integer :: fx,fy,fz,ft
    integer :: gx,gy,gz,gt
    integer :: hx,hy,hz,ht
    integer :: ox,oy,oz,ot
    integer :: px,py,pz,pt
    integer :: qx,qy,qz,qt
    integer :: rx,ry,rz,rt
```

```
#define U_mux          U_xd(ix,iy,iz,it,mu)
#define U_nux          U_xd(ix,iy,iz,it,nu)
#define dS_gbydU_mux   dS_gbydU(ix,iy,iz,it,mu)
#define dS_gbydU_nux   dS_gbydU(ix,iy,iz,it,nu)
#define U_muxpmu       U_xd(jx,jy,jz,jt,mu)
#define U_nuxpmu       U_xd(jx,jy,jz,jt,nu)
#define U_muxpnu       U_xd(kx,ky,kz,kt,mu)
#define U_nuxpnu       U_xd(kx,ky,kz,kt,nu)
#define U_muxmmu       U_xd(lx,ly,lz,lt,mu)
#define U_nuxmmu       U_xd(lx,ly,lz,lt,nu)
#define U_muxmnu       U_xd(mx,my,mz,mt,mu)
#define U_nuxmnu       U_xd(mx,my,mz,mt,nu)

#define U_nuxp2mu      U_xd(ax,ay,az,at,nu)
#define U_muxp2nu      U_xd(bx,by,bz,bt,mu)
#define U_nuxm2mu      U_xd(cx,cy,cz,ct,nu)
#define U_muxm2mu      U_xd(cx,cy,cz,ct,mu)
#define U_nuxm2nu      U_xd(dx,dy,dz,dt,nu)
#define U_muxm2nu      U_xd(dx,dy,dz,dt,mu)
#define U_nuxpmupnu    U_xd(ex,ey,ez,et,nu)
#define U_muxpmupnu    U_xd(ex,ey,ez,et,mu)
#define U_nuxpmumnu    U_xd(fx,fy,fz,ft,nu)
#define U_muxpmumnu    U_xd(fx,fy,fz,ft,mu)
#define U_nuxmmupnu    U_xd(gx,gy,gz,gt,nu)
#define U_muxmmupnu    U_xd(gx,gy,gz,gt,mu)
#define U_nuxmmumnu    U_xd(hx,hy,hz,ht,nu)
#define U_muxmmumnu    U_xd(hx,hy,hz,ht,mu)
#define U_nuxp2mumnu   U_xd(ox,oy,oz,ot,nu)
#define U_muxmmup2nu   U_xd(px,py,pz,pt,mu)
#define U_nuxm2nupmu   U_xd(qx,qy,qz,qt,nu)
#define U_muxm2mupnu   U_xd(rx,ry,rz,rt,mu)

      dmu = 0
      dnu = 0

      dmu(mu) = 1
      dnu(nu) = 1

      do it=1,nt
         jt = mapt(it + dmu(4))
         kt = mapt(it + dnu(4))
         lt = mapt(it - dmu(4))
         mt = mapt(it - dnu(4))
         at = mapt(it + 2*dmu(4))
         bt = mapt(it + 2*dnu(4))
         ct = mapt(it - 2*dmu(4))
         dt = mapt(it - 2*dnu(4))
         et = mapt(it + dmu(4) + dnu(4))
         ft = mapt(it + dmu(4) - dnu(4))
         gt = mapt(it - dmu(4) + dnu(4))
         ht = mapt(it - dmu(4) - dnu(4))
         ot = mapt(it + 2*dmu(4) - dnu(4))
         pt = mapt(it - dmu(4) + 2*dnu(4))
         qt = mapt(it + dmu(4) - 2*dnu(4))
         rt = mapt(it - 2*dmu(4) + dnu(4))

         do iz=1,nz
            jz = mapz(iz + dmu(3))
            kz = mapz(iz + dnu(3))
            lz = mapz(iz - dmu(3))
            mz = mapz(iz - dnu(3))
            az = mapz(iz + 2*dmu(3))
            bz = mapz(iz + 2*dnu(3))
            cz = mapz(iz - 2*dmu(3))
            dz = mapz(iz - 2*dnu(3))
            ez = mapz(iz + dmu(3) + dnu(3))
            fz = mapz(iz + dmu(3) - dnu(3))
            gz = mapz(iz - dmu(3) + dnu(3))
            hz = mapz(iz - dmu(3) - dnu(3))
            oz = mapz(iz + 2*dmu(3) - dnu(3))
            pz = mapz(iz - dmu(3) + 2*dnu(3))
            qz = mapz(iz + dmu(3) - 2*dnu(3))
            rz = mapz(iz - 2*dmu(3) + dnu(3))

            do iy=1,ny
               jy = mapy(iy + dmu(2))
               ky = mapy(iy + dnu(2))
               ly = mapy(iy - dmu(2))
               my = mapy(iy - dnu(2))
               ay = mapy(iy + 2*dmu(2))
               by = mapy(iy + 2*dnu(2))
               cy = mapy(iy - 2*dmu(2))
               dy = mapy(iy - 2*dnu(2))
               ey = mapy(iy + dmu(2) + dnu(2))
               fy = mapy(iy + dmu(2) - dnu(2))
               gy = mapy(iy - dmu(2) + dnu(2))
               hy = mapy(iy - dmu(2) - dnu(2))
               oy = mapy(iy + 2*dmu(2) - dnu(2))
```

```
py = mapy(iy - dmu(2) + 2*dnu(2))
qy = mapy(iy + dmu(2) - 2*dnu(2))
ry = mapy(iy - 2*dmu(2) + dnu(2))

do ix=1,nx
    jx = mapx(ix + dmu(1))
    kx = mapx(ix + dnu(1))
    lx = mapx(ix - dmu(1))
    mx = mapx(ix - dnu(1))
    ax = mapx(ix + 2*dmu(1))
    bx = mapx(ix + 2*dnu(1))
    cx = mapx(ix - 2*dmu(1))
    dx = mapx(ix - 2*dnu(1))
    ex = mapx(ix + dmu(1) + dnu(1))
    fx = mapx(ix + dmu(1) - dnu(1))
    gx = mapx(ix - dmu(1) + dnu(1))
    hx = mapx(ix - dmu(1) - dnu(1))
    ox = mapx(ix + 2*dmu(1) - dnu(1))
    px = mapx(ix - dmu(1) + 2*dnu(1))
    qx = mapx(ix + dmu(1) - 2*dnu(1))
    rx = mapx(ix - 2*dmu(1) + dnu(1))

    !!     _                                  _
    !!    | | +      [ V_muxp +          ] :   |
    !!          |_| [             V_muxm ] :  _|

    call MultiplyMatMatDag(UnuUmudag, U_nuxpmu, U_muxpnu)
    call MultiplyMatDagMatDag(UnudagUmudag, U_nuxpmumnu, U_muxmnu)

    call MultiplyMatMatDag(V_muxp, UnuUmudag, U_nux)
    call MultiplyMatMat(V_muxm, UnudagUmudag, U_nuxmnu)

    dS_gbydU_mux%Cl = dS_gbydU_mux%Cl + beta_11*(V_muxp%Cl + V_muxm%Cl)

    !!   _   _                              _
    !! |_ + _| [ V_nuxm + V_nuxp ] : |  .
    !!

    call MultiplyMatDagMatDag(UmudagUnudag, U_muxmmupnu, U_nuxmmu)
    call MultiplyMatMat(V_nuxm, UmudagUnudag, U_muxmmu)

    call MultiplyMatDagMatDag(V_nuxp, UnuUmudag, U_mux)

    dS_gbydU_nux%Cl = dS_gbydU_nux%Cl + beta_11*(V_nuxp%Cl + V_nuxm%Cl)

    !!   __      __     __
    !! | _| + |_ | :  _|
    !!

    call MultiplyMatMat(UmuUnu, U_muxpmu, U_nuxp2mu)
    call MultiplyMatMat(U2, U_muxpnu, U_muxpmupnu)
    call MultiplyMatMatDag(T_xp, UmuUnu, U2)

    call MultiplyMatMatDag(R_xp, T_xp, U_nux)
    call MultiplyMatMat(R_xm, UnuUmudag, V_nuxm)

    dS_gbydU_mux%Cl = dS_gbydU_mux%Cl + beta_21*(R_xp%Cl + R_xm%Cl)

    !!   __    __
    !! |__ + __|
    !!

    call MultiplyMatMat(U2, U_muxm2mu, U_muxmmu)
    call MultiplyMatMat(UnuUmu, U_nuxm2mu, U_muxm2mupnu)
    call MultiplyMatDagMat(T_xm, UnuUmu, U2)

    call MultiplyMatDagMat(R_xm, U_muxmmupnu, T_xm)
    call MultiplyMatDagMatDag(R_xp, T_xp, U_mux)

    dS_gbydU_nux%Cl = dS_gbydU_nux%Cl + beta_12*(R_xp%Cl + R_xm%Cl)

    !!   _              _
    !! | | +  _   : | |
    !!  _|     |  :   |
    !!        |_|

    call MultiplyMatMat(UnuUmu, U_nuxpnu, U_muxp2nu)
    call MultiplyMatMat(U2, U_nuxpmu, U_nuxpmupnu)
    call MultiplyMatMatDag(T_xp, UnuUmu, U2)

    call MultiplyMatMatDag(R_xp, T_xp, U_mux)
    call MultiplyMatDagMat(R_xm, UnuUmudag, V_muxm)

    dS_gbydU_nux%Cl = dS_gbydU_nux%Cl + beta_21*(R_xp%Cl + R_xm%Cl)

    !!   _
    !! | |
```

```
        !! | |
        !!     + | |
        !!       |_|

        call MultiplyMatMat(U2,U_nuxm2nu,U_nuxmnu)
        call MultiplyMatMat(UmuUnu,U_muxm2nu,U_nuxm2nupmu)
        call MultiplyMatDagMat(T_xm,UmuUnu,U2)

        call MultiplyMatDagMatDag(R_xp,T_xp,U_nux)
        call MultiplyMatDagMat(R_xm,U_nuxpmumnu,T_xm)

        dS_gbydU_mux%Cl = dS_gbydU_mux%Cl + beta_12*(R_xp%Cl + R_xm%Cl)

        !!        _
        !!    _  | |
        !! |    + |_  :   .
        !! |_|        : |_

        call MultiplyMatDagMat(UnudagUmu,U_nuxmmumnu,U_muxmmumnu)
        call MultiplyMatMat(T_xm,UmudagUnudag,UnudagUmu)

        call MultiplyMatMatDag(UnuUmudag,U_nuxpnu,U_muxmmup2nu)
        call MultiplyMatMat(U2,U_nuxmmu,U_nuxmmupnu)
        call MultiplyMatMatDag(T_xp,UnuUmudag,U2)

        call MultiplyMatMat(R_xm,T_xm,U_nuxmnu)
        call MultiplyMatMat(R_xp,T_xp,U_muxmmu)

        dS_gbydU_nux%Cl = dS_gbydU_nux%Cl + beta_21*(R_xp%Cl + R_xm%Cl)

        !!
        !!    _   +  _
        !! |__|     |__|

        call MultiplyMatMatDag(UmuUnudag,U_muxpmu,U_nuxp2mumnu)
        call MultiplyMatMat(U2,U_muxmnu,U_muxpmumnu)
        call MultiplyMatMatDag(T_xp,UmuUnudag,U2)

        call MultiplyMatMatDag(T_xm,UnudagUmudag,UnudagUmu)

        call MultiplyMatMat(R_xp,T_xp,U_nuxmnu)
        call MultiplyMatMat(R_xm,T_xm,U_muxmmu)

        dS_gbydU_mux%Cl = dS_gbydU_mux%Cl + beta_21*(R_xp%Cl + R_xm%Cl)

      end do
    end do
  enddo
end do

end subroutine GetStaples

end module GaugeAction
```

FLICFermions: Implementation of the pseudofermion field component of the equations of motion, for the FLIC fermion action.

```
module FLICFermions

  use ChiralFLICOperator
  use ConjGradSolvers
  use GaugeField
  use FatLinks
  use GL3Diag
  use VectorAlgebra
  use MatrixAlgebra
  use TensorAlgebra
  implicit none

  interface FermionMatrixMultiply
     module procedure Dflic
  end interface

  interface FermionMatrixDagMultiply
     module procedure Dflicdag
  end interface

  interface SqFermionMatrixMultiply
     module procedure SqFLICOperate
  end interface

  type(colour_matrix), dimension(:,:,:,:,:,:), allocatable, target :: Un_xd

  integer :: iter_cg
  real(dp) :: tolerance_cg = 1.0d-6
  real(dp) :: r_md = 1.0d0, kappa, traj_minev = 1.0d0
```

```
contains

  subroutine InitialiseFermionMatrix(U_xd)

    type(colour_matrix), dimension(:,:,:,:,:) :: U_xd

    integer :: isweeps
    real(dp) :: u0, u0fl

    UFL_xd = U_xd(1:nxs,1:nys,1:nzs,1:nts,:)

    do isweeps =1,ape_sweeps
       Un_xd(:,:,:,:,:,isweeps) = UFL_xd
       call APESmear(UFL_xd,alpha_smear)
       if ( isweeps == ape_sweeps) call FixSU3(UFL_xd)
       call ShadowGaugeField(UFL_xd,1)
    end do

    call InitialiseFLICOperator(U_xd,UFL_xd, u0_bar, u0fl_bar, m_f, c_sw, bct)

  end subroutine InitialiseFermionMatrix

  subroutine PseudoFermionMatrixOperate(phi,eta)

    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi, eta
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi_pr
    integer :: iter_cg2
    real(dp) :: error, norm
    integer :: ix,iy,iz,it,is

    call BiCGStabInvert(phi, eta, tolerance_cg, iter_cg, Dflicdag)
    phi_pr = eta
    call BiCGStabInvert(phi_pr, eta, tolerance_cg, iter_cg2, Dflic)
    iter_cg = iter_cg + iter_cg2

  end subroutine PseudoFermionMatrixOperate

  subroutine GetdS_pfbydU(dS_pfbydU,U_xd,eta)

    type(colour_matrix), dimension(nxp,nyp,nzp,ntp,nd) :: dS_pfbydU
    type(colour_matrix), dimension(:,:,:,:,:) :: U_xd
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: eta
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: chi

    type(colour_vector), dimension(nxs,nys,nzs,nts,ns) :: eta_pr, chi_pr
    type(colour_matrix), dimension(nxp,nyp,nzp,ntp,nd) :: V_xd
    type(colour_matrix), dimension(nxs,nys,nzs,nts,nd) :: dS_pfbydUfl, Ui_xd

    type(colour_vector), dimension(ns) :: eta_l, chi_l
    type(colour_vector), dimension(ns) :: eta_r, chi_r

    integer, dimension(nd) :: dmu
    integer :: ix,iy,iz,it
    integer :: jx,jy,jz,jt
    integer :: mu, isweeps, t0, t1, tsum, nu, inu
    real(dp) :: c_l, c_r

#define dS_pfbydU_mux dS_pfbydU(ix,iy,iz,it,mu)
#define dS_pfbydUfl_mux dS_pfbydUfl(ix,iy,iz,it,mu)

#define eta_x(is) eta_pr(ix,iy,iz,it,is)
#define eta_xpmu(is) eta_pr(jx,jy,jz,jt,is)

#define chi_x(is) chi_pr(ix,iy,iz,it,is)
#define chi_xpmu(is) chi_pr(jx,jy,jz,jt,is)

    !set chi = M eta = gamma_5 H eta
    call Dflic(eta,chi)

    eta_pr(1:nx,1:ny,1:nz,1:nt,:) = eta(1:nx,1:ny,1:nz,1:nt,:)
    chi_pr(1:nx,1:ny,1:nz,1:nt,:) = chi(1:nx,1:ny,1:nz,1:nt,:)

    call ShadowFermionField(eta_pr,1)
    call ShadowFermionField(chi_pr,1)

    do mu=1,nd
       dmu = 0
       dmu(mu) = 1
       do it=1,nt
          jt = mapt(it + dmu(4))
          do iz=1,nz
             jz = mapz(iz + dmu(3))
             do iy=1,ny
                jy = mapy(iy + dmu(2))
                do ix=1,nx
                   jx = mapx(ix + dmu(1))
```

```
                    dS_pfbydU_mux%Cl = 0.0d0

                    dS_pfbydUfl_mux%Cl = 0.0d0

                    call VecDag(eta_l, eta_x(:))
                    call VecDag(chi_l, chi_x(:))
                    call GammaPhi(chi_r, chi_xpmu(:),mu)
                    call GammaPhi(eta_r, eta_xpmu(:),mu)

                    call AddVectorOuterProduct(dS_pfbydU_mux, eta_l, chi_r)
                    call SubVectorOuterProduct(dS_pfbydU_mux, chi_l, eta_r)

                    chi_r = chi_xpmu(:)
                    eta_r = eta_xpmu(:)
                    call AddVectorOuterProduct(dS_pfbydUfl_mux, eta_l, chi_r)
                    call AddVectorOuterProduct(dS_pfbydUfl_mux, chi_l, eta_r)

                 end do
              end do
           enddo
        end do
     end do

     !Clover term derivative.
     do mu=1,nd
        nu = mu
        do inu=1,nd-1
           nu = modulo(nu,nd)+1
           call dFmunubydU(dS_pfbydUfl, UFL_xd, eta_pr, chi_pr, mu, nu)
        end do
     end do

     do isweeps=ape_sweeps,1,-1

        Ui_xd = Un_xd(:,:,:,:,:,isweeps)

        call GetSmearedLinks(V_xd, Ui_xd, alpha_smear)

        call dSbydU_nStardSbydV(dS_pfbydUfl, V_xd)

        call ShadowGaugeField(dS_pfbydUfl,1)
        call dSbydVStardVbydU(dS_pfbydUfl, Ui_xd, V_xd)

     end do

     do mu=1,nd
        do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
           dS_pfbydU_mux%Cl = - (0.5d0/u0_bar)*dS_pfbydU_mux%Cl - (0.5d0/u0fl_bar)*dS_pfbydUfl_mux%Cl
        end do; end do; end do; end do
     end do

#undef dS_pfbydU_mux
#undef dS_pfbydUfl_mux

#undef eta_x
#undef eta_xpmu
#undef chi_x
#undef chi_xpmu

   end subroutine GetdS_pfbydU

   subroutine dFmunubydU(dSbydU, U_xd, eta, chi, mu, nu)

      type(colour_matrix), dimension(nxs,nys,nzs,nts,nd) :: dSbydU, U_xd
      type(colour_vector), dimension(nxs,nys,nzs,nts,ns) :: eta, chi
      integer :: mu,nu
      type(colour_vector), dimension(ns) :: eta_l, chi_r, sigma_munuchi
      type(colour_matrix)  :: dCbydU, V_mupx, UmudagUnudag, UnuUmudag
      type(colour_matrix)  :: V_mumx, UnudagUmudag, UmudagUnu

!!#define V_mumx V_mupx
!!#define UnudagUmudag UmudagUnudag
!!#define UmudagUnu     UnuUmudag

      integer, dimension(nd) :: dmu, dnu
      integer :: ix,iy,iz,it
      integer :: jx,jy,jz,jt
      integer :: kx,ky,kz,kt
      integer :: lx,ly,lz,lt
      integer :: mx,my,mz,mt
      integer :: ax,ay,az,at
      real(dp) :: alpha_sw

#define dSbydU_mux dSbydU(ix,iy,iz,it,mu)

#define U_nux        U_xd(ix,iy,iz,it,nu)
#define U_nuxpmu     U_xd(jx,jy,jz,jt,nu)
#define U_muxpnu     U_xd(kx,ky,kz,kt,mu)
```

```
#define U_nuxmnu      U_xd(lx,ly,lz,lt,nu)
#define U_muxmnu      U_xd(lx,ly,lz,lt,mu)
#define U_nuxpmumnu   U_xd(mx,my,mz,mt,nu)

#define eta_x(is) eta(ix,iy,iz,it,is)
#define eta_xpmu(is) eta(jx,jy,jz,jt,is)
#define eta_xpnu(is) eta(kx,ky,kz,kt,is)
#define eta_xmnu(is) eta(lx,ly,lz,lt,is)
#define eta_xpmumnu(is) eta(mx,my,mz,mt,is)
#define eta_xpmupnu(is) eta(ax,ay,az,at,is)

#define chi_x(is) chi(ix,iy,iz,it,is)
#define chi_xpmu(is) chi(jx,jy,jz,jt,is)
#define chi_xpnu(is) chi(kx,ky,kz,kt,is)
#define chi_xmnu(is) chi(lx,ly,lz,lt,is)
#define chi_xpmumnu(is) chi(mx,my,mz,mt,is)
#define chi_xpmupnu(is) chi(ax,ay,az,at,is)

   dmu = 0
   dnu = 0

   dmu(mu) = 1
   dnu(nu) = 1

   alpha_sw = 0.125d0/(u0fl_bar**3)

   do it=1,nt
      jt = mapt(it + dmu(4))
      kt = mapt(it + dnu(4))
      lt = mapt(it - dnu(4))
      mt = mapt(it - dnu(4) + dmu(4))
      at = mapt(it + dnu(4) + dmu(4))
      do iz=1,nz
         jz = mapz(iz + dmu(3))
         kz = mapz(iz + dnu(3))
         lz = mapz(iz - dnu(3))
         mz = mapz(iz - dnu(3) + dmu(3))
         az = mapz(iz + dnu(3) + dmu(3))
         do iy=1,ny
            jy = mapy(iy + dmu(2))
            ky = mapy(iy + dnu(2))
            ly = mapy(iy - dnu(2))
            my = mapy(iy - dnu(2) + dmu(2))
            ay = mapy(iy + dnu(2) + dmu(2))
            do ix=1,nx
               jx = mapx(ix + dmu(1))
               kx = mapx(ix + dnu(1))
               lx = mapx(ix - dnu(1))
               mx = mapx(ix - dnu(1) + dmu(1))
               ax = mapx(ix + dnu(1) + dmu(1))

               dCbydU%Cl = 0.0d0

               !Positive terms
               call MultiplyMatDagMatDag(UmudagUnudag, U_muxpnu, U_nux)
               call MultiplyMatMatDag(UnuUmudag, U_nuxpmu, U_muxpnu)
               call MultiplyMatMatDag(V_mupx, UnuUmudag, U_nux)

               !C+mu+nu
               call VecDag(eta_l, eta_x(:))
               call SigmaPhi(sigma_munuchi, chi_x(:),mu,nu)
               call MultiplyMatVec(chi_r, V_mupx, sigma_munuchi)
               call AddVectorOuterProduct(dCbydU, eta_l, chi_r)

               call VecDag(eta_l, chi_x(:))
               call SigmaPhi(sigma_munuchi, eta_x(:),mu,nu)
               call MultiplyMatVec(chi_r, V_mupx, sigma_munuchi)
               call AddVectorOuterProduct(dCbydU, eta_l, chi_r)

               !C+nu-mu
               call MultiplyVecDagMat(eta_l, eta_xpmu(:),V_mupx)
               call SigmaPhi(chi_r, chi_xpmu(:),mu,nu)
               call AddVectorOuterProduct (dCbydU, eta_l, chi_r)

               call MultiplyVecDagMat(eta_l, chi_xpmu(:),V_mupx)
               call SigmaPhi(chi_r, eta_xpmu(:),mu,nu)
               call AddVectorOuterProduct (dCbydU, eta_l, chi_r)

               !C-mu-nu
               call MultiplyVecDagMat(eta_l, eta_xpmupnu(:),UmudagUnudag)
               call SigmaPhi(sigma_munuchi, chi_xpmupnu(:),mu,nu)
               call MultiplyMatVec(chi_r, U_nuxpmu, sigma_munuchi)
               call AddVectorOuterProduct (dCbydU, eta_l, chi_r)

               call MultiplyVecDagMat(eta_l, chi_xpmupnu(:),UmudagUnudag)
               call SigmaPhi(sigma_munuchi, eta_xpmupnu(:),mu,nu)
               call MultiplyMatVec(chi_r, U_nuxpmu, sigma_munuchi)
               call AddVectorOuterProduct (dCbydU, eta_l, chi_r)
```

```
    !C-nu+mu
    call MultiplyVecDagMatDag(eta_l , eta_xpnu (:) ,U_nux)
    call SigmaPhi(sigma_munuchi , chi_xpnu (:) ,mu, nu)
    call MultiplyMatVec(chi_r ,UnuUmudag, sigma_munuchi)
    call AddVectorOuterProduct (dCbydU, eta_l , chi_r)

    call MultiplyVecDagMatDag(eta_l , chi_xpnu (:) ,U_nux)
    call SigmaPhi(sigma_munuchi , eta_xpnu (:) ,mu, nu)
    call MultiplyMatVec(chi_r ,UnuUmudag, sigma_munuchi)
    call AddVectorOuterProduct (dCbydU, eta_l , chi_r)


    !Negative Terms
    call MultiplyMatDagMat(UmudagUnu , U_muxmnu , U_nuxmnu)
    call MultiplyMatDagMatDag (UnudagUmudag , U_nuxpmumnu , U_muxmnu)
    call MultiplyMatMat (V_mumx , UnudagUmudag , U_nuxmnu)

    !C+mu+nu^dag
    call MultiplyVecDagMat ( eta_l , eta_xmnu (:) ,U_nuxmnu)
    call SigmaPhi(sigma_munuchi , chi_xmnu (:) ,mu, nu)
    call MultiplyMatVec(chi_r , UnudagUmudag , sigma_munuchi)
    call SubVectorOuterProduct (dCbydU, eta_l , chi_r)

    call MultiplyVecDagMat ( eta_l , chi_xmnu (:) ,U_nuxmnu)
    call SigmaPhi(sigma_munuchi , eta_xmnu (:) ,mu, nu)
    call MultiplyMatVec(chi_r , UnudagUmudag , sigma_munuchi)
    call SubVectorOuterProduct (dCbydU, eta_l , chi_r)

    !C+nu-mu^dag
    call MultiplyVecDagMat ( eta_l , eta_xpmumnu (:) ,UmudagUnu)
    call SigmaPhi(sigma_munuchi , chi_xpmumnu (:) ,mu, nu)
    call MultiplyMatDagVec (chi_r , U_nuxpmumnu , sigma_munuchi)
    call SubVectorOuterProduct (dCbydU, eta_l , chi_r)

    call MultiplyVecDagMat ( eta_l , chi_xpmumnu (:) ,UmudagUnu)
    call SigmaPhi(sigma_munuchi , eta_xpmumnu (:) ,mu, nu)
    call MultiplyMatDagVec (chi_r , U_nuxpmumnu , sigma_munuchi)
    call SubVectorOuterProduct (dCbydU, eta_l , chi_r)

    !C+mu-nu^dag
    call VecDag(eta_l , eta_x (:))
    call SigmaPhi(sigma_munuchi , chi_x (:) ,mu, nu)
    call MultiplyMatVec(chi_r ,V_mumx, sigma_munuchi)
    call SubVectorOuterProduct(dCbydU, eta_l , chi_r)

    call VecDag(eta_l , chi_x (:))
    call SigmaPhi(sigma_munuchi , eta_x (:) ,mu, nu)
    call MultiplyMatVec(chi_r ,V_mumx, sigma_munuchi)
    call SubVectorOuterProduct(dCbydU, eta_l , chi_r)

    !C-mu-nu^dag
    call MultiplyVecDagMat ( eta_l , eta_xpmu (:) ,V_mumx)
    call SigmaPhi(chi_r , chi_xpmu (:) ,mu, nu)
    call SubVectorOuterProduct (dCbydU, eta_l , chi_r)

    call MultiplyVecDagMat ( eta_l , chi_xpmu (:) ,V_mumx)
    call SigmaPhi(chi_r , eta_xpmu (:) ,mu, nu)
    call SubVectorOuterProduct (dCbydU, eta_l , chi_r)

    dSbydU_mux%Cl = dSbydU_mux%Cl + alpha_sw*dCbydU%Cl

   end do
  end do
 enddo
end do

#undef dSbydU_mux

#undef U_nux
#undef U_nuxpmu
#undef U_muxpnu
#undef U_nuxpmumnu
#undef U_muxmnu
#undef U_nuxmnu

#undef eta_x
#undef eta_xpmu
#undef eta_xpnu
#undef eta_xpmumnu
#undef eta_xmnu
#undef eta_xpmupnu

#undef chi_x
#undef chi_xpmu
#undef chi_xpnu
#undef chi_xpmumnu
#undef chi_xmnu
```

```fortran
#undef chi_xpmupnu

#undef V_mumx
#undef UnudagUmudag
#undef UmudagUnu

  end subroutine dFmunubydU

  subroutine dSbydU_nStardSbydV(dSbydU,V_xd)

    !Accepts dSbydU_n and applies multiple chain rules to return
    !dSbydV reusing the same array.

    type(colour_matrix), dimension(nxs,nys,nzs,nts,nd) :: dSbydU
    type(colour_matrix), dimension(nxp,nyp,nzp,ntp,nd) :: V_xd

    type(colour_matrix) :: W_xd, H_xd, G_xd
    complex(dc) :: alpha
    complex(dc) :: detW, ddetW
    type(colour_matrix) :: ddetWbydW
    type(colour_matrix) :: L_xd, R_xd

    integer, parameter :: npole = 8
    type(real_vector) :: lambda_xd, kappa_xd(0:npole), tau_xd
    type(colour_matrix) :: A_xd, B_xd, M_xd(0:npole)

    !Zolotarev approximation to 1/sqrt(x)
    real(dp), parameter :: c_2n = 11.00084578330601d0, d_n = 0.1893126990716808d0
    real(dp), dimension(npole), parameter :: b_l = (/ 5.324499945347391d-03, 7.142501319184830d-03, &
         & 1.141535786342936d-02, 1.970167120121944d-02, 3.541889548512958d-02, 6.752194317679540d-02, &
         & 1.527670977603933d-01, 7.007080332485007d-01 /)
    real(dp), dimension(npole), parameter :: c_l = (/ 7.184251271695007d-05, 8.020314400574700d-04, &
         & 3.346253435165938d-03, 1.151052580482129d-02, 3.782045395218336d-02, 1.259583793469389d-01, &
         & 4.622293374625976d-01, 2.534100646491083d+00 /)
    integer :: ix,iy,iz,it,mu,ipole

#define dSbydU_mux dSbydU(ix,iy,iz,it,mu)
#define dSbydW_mux dSbydU(ix,iy,iz,it,mu)
#define dSbydV_mux dSbydU(ix,iy,iz,it,mu)

    do mu=1,nd
       do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx

          call MultiplyMatDagMat(H_xd,V_xd(ix,iy,iz,it,mu),V_xd(ix,iy,iz,it,mu))
          call DiagonaliseMat(H_xd,lambda_xd,G_xd)
          traj_minev = min(traj_minev,minval(sqrt(lambda_xd%Cl)))

          tau_xd%Cl = 1.0d0/sqrt(lambda_xd%Cl)

          !Undiagonalise H_xd back to the V_xd basis
          call MultiplyMatRealDiagMat(W_xd,G_xd,tau_xd)
          call MultiplyMatMatdag(H_xd,W_xd,G_xd)

          !Set W_xd = U H \in U(3)
          call MultiplyMatMat(W_xd,V_xd(ix,iy,iz,it,mu),H_xd)

          ! U_n = detW^(-1/3)*W
          ! First Chain rule: dS/dW = dS/dU_n * dU_n/dW + dS/dU_n^dag * dU_n^dag/dW
          ! U_n^dag = conjg(detW)*W^dag => dU_n^dag/dW = 0

          !Note ddetWbydW[j,i] = (d detW)/(d W[i,j])

          ddetWbydW%Cl(1,1) = W_xd%Cl(2,2)*W_xd%Cl(3,3) - W_xd%Cl(2,3)*W_xd%Cl(3,2)
          ddetWbydW%Cl(1,2) = W_xd%Cl(1,3)*W_xd%Cl(3,2) - W_xd%Cl(1,2)*W_xd%Cl(3,3)
          ddetWbydW%Cl(1,3) = W_xd%Cl(1,2)*W_xd%Cl(2,3) - W_xd%Cl(1,3)*W_xd%Cl(2,2)

          ddetWbydW%Cl(2,1) = W_xd%Cl(2,3)*W_xd%Cl(3,1) - W_xd%Cl(2,1)*W_xd%Cl(3,3)
          ddetWbydW%Cl(2,2) = W_xd%Cl(1,1)*W_xd%Cl(3,3) - W_xd%Cl(1,3)*W_xd%Cl(3,1)
          ddetWbydW%Cl(2,3) = W_xd%Cl(1,3)*W_xd%Cl(2,1) - W_xd%Cl(1,1)*W_xd%Cl(2,3)

          ddetWbydW%Cl(3,1) = W_xd%Cl(2,1)*W_xd%Cl(3,2) - W_xd%Cl(2,2)*W_xd%Cl(3,1)
          ddetWbydW%Cl(3,2) = W_xd%Cl(1,2)*W_xd%Cl(3,1) - W_xd%Cl(1,1)*W_xd%Cl(3,2)
          ddetWbydW%Cl(3,3) = W_xd%Cl(1,1)*W_xd%Cl(2,2) - W_xd%Cl(1,2)*W_xd%Cl(2,1)

          detW = W_xd%Cl(1,1)*ddetWbydW%Cl(1,1) + W_xd%Cl(1,2)*ddetWbydW%Cl(2,1) + W_xd%Cl(1,3)*ddetWbydW%
               Cl(3,1)

          !Get detW^(-1/3)
          detW = 1.0d0/(detW**(1.0d0/3.0d0))

          !Get -(1/3)*detW^(-4/3)
          ddetW = -(1.0d0/3.0d0)*detW**4

          L_xd%Cl = detW*dSbydU_mux%Cl

          call StarInnerProduct(alpha,dSbydU_mux,W_xd)

          R_xd%Cl = (ddetW*alpha)*ddetWbydW%Cl
```

```
          dSbydW_mux%Cl = L_xd%Cl + R_xd%Cl

          ! W= V H, W^dag = H V^dag , H= h^(-1/2) , h = [V^dag V]
          ! Second Chain rule: dS/dV = dS/dW * dW/dV + dS/dW^dag * dW^dag/dV
          !           = dS/dW * ( I x H + V dH/dV) + dS/dW^dag * ( dH/dV V^dag )
          ! Intermediate Chain rule: dH/dV = dH/dh * dh/dV

          call MultiplyMatMat(L_xd, H_xd, dSbydW_mux)

          call MultiplyMatMat(A_xd, dSbydW_mux, V_xd(ix, iy, iz, it, mu))
          call MatDag(B_xd, A_xd)

          !Set W_xd = dS/dH
          W_xd%Cl = A_xd%Cl + B_xd%Cl

          kappa_xd(0)%Cl = d_n*(lambda_xd%Cl + c_2n)
          do ipole=1,npole
             kappa_xd(ipole)%Cl = 1.0d0/(lambda_xd%Cl + c_l(ipole))
          end do

          !Undiagonalise H_xd back to the V_xd basis
          do ipole=0,npole
             call MultiplyMatRealDiagMat(A_xd, G_xd, kappa_xd(ipole))
             call MultiplyMatMatdag(M_xd(ipole), A_xd, G_xd)
          end do

          B_xd%Cl = 0.0d0
          do ipole=1,npole
             B_xd%Cl = B_xd%Cl + b_l(ipole)*M_xd(ipole)%Cl
          end do

          A_xd%Cl = 0.0d0

          A_xd%Cl(1,1) = d_n
          A_xd%Cl(2,2) = d_n
          A_xd%Cl(3,3) = d_n

          ! T * (L x R) = R T L
          !StarOuterProduct(R_xd, W_xd, A_xd, B_xd)
          call MultiplyMatMat(G_xd, B_xd, W_xd)
          call MultiplyMatMat(R_xd, G_xd, A_xd)

          call MultiplyMatMat(A_xd, W_xd, M_xd(0))
          do ipole=1,npole
             ! T * (L x R) = R T L
             !StarOuterProduct(B_xd, A_xd, M_xd(ipole), M_xd(ipole))
             call MultiplyMatMat(G_xd, M_xd(ipole), A_xd)
             call MultiplyMatMat(B_xd, G_xd, M_xd(ipole))

             R_xd%Cl = R_xd%Cl - b_l(ipole)*B_xd%Cl
          end do

          call MultiplyMatMatDag(dSbydV_mux, R_xd, V_xd(ix, iy, iz, it, mu))

          dSbydV_mux%Cl = L_xd%Cl + dSbydV_mux%Cl

       end do; end do; end do; end do
    end do

#undef dSbydU_mux
#undef dSbydV_mux
#undef dSbydW_mux

  end subroutine dSbydU_nStardSbydV

  subroutine dSbydVstardVbydU(dSbydU, U_xd, V_xd)

    !Accepts dSbydV and applies dS/dU[mu,x] = dS/dV[nu,y] * dV[nu,y]/dU[mu,x] +
    !                                 dS/dV^dag[nu,y] * dV^dag[nu,y]/dU[mu,x],
    !returns the result in dSbydU, and reuses the storage in V_xd.

    type(colour_matrix), dimension(nxs,nys,nzs,nts,nd) :: dSbydU, U_xd
    type(colour_matrix), dimension(nxp,nyp,nzp,ntp,nd) :: V_xd

    type(colour_matrix) :: V_x, W_xpmu, T_x
    integer :: mu,nu, inu

    integer, dimension(nd) :: dmu, dnu
    integer :: ix,iy,iz,it
    integer :: jx,jy,jz,jt
    integer :: kx,ky,kz,kt
    integer :: lx,ly,lz,lt
    integer :: mx,my,mz,mt

#define U_nux        U_xd(ix,iy,iz,it,nu)
#define U_nuxpmu     U_xd(jx,jy,jz,jt,nu)
#define U_muxpnu     U_xd(kx,ky,kz,kt,mu)
```

```
#define U_nuxmnu        U_xd(lx,ly,lz,lt,nu)
#define U_muxmnu        U_xd(lx,ly,lz,lt,mu)
#define U_nuxpmumnu     U_xd(mx,my,mz,mt,nu)

#define dSbydU_mux           V_xd(ix,iy,iz,it,mu)
#define dSbydV_mux           dSbydU(ix,iy,iz,it,mu)
#define dSbydV_nux           dSbydU(ix,iy,iz,it,nu)
#define dSbydV_nuxpmu        dSbydU(jx,jy,jz,jt,nu)
#define dSbydV_muxpnu        dSbydU(kx,ky,kz,kt,mu)
#define dSbydV_nuxpnu        dSbydU(kx,ky,kz,kt,nu)
#define dSbydV_muxmnu        dSbydU(lx,ly,lz,lt,mu)
#define dSbydV_nuxmnu        dSbydU(lx,ly,lz,lt,nu)
#define dSbydV_nuxpmumnu     dSbydU(mx,my,mz,mt,nu)

   do mu=1,nd
      do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
         dSbydU_mux%Cl = 0.0d0
      end do; end do; end do; end do
   end do

   do mu=1,nd
      nu = mu
      do inu=1,nd-1
         nu = modulo(nu,nd)+1
         dmu = 0
         dnu = 0

         dmu(mu) = 1
         dnu(nu) = 1

         do it=1,nt
            jt = mapt(it + dmu(4))
            kt = mapt(it + dnu(4))
            lt = mapt(it - dnu(4))
            mt = mapt(it - dnu(4) + dmu(4))
            do iz=1,nz
               jz = mapz(iz + dmu(3))
               kz = mapz(iz + dnu(3))
               lz = mapz(iz - dnu(3))
               mz = mapz(iz - dnu(3) + dmu(3))
               do iy=1,ny
                  jy = mapy(iy + dmu(2))
                  ky = mapy(iy + dnu(2))
                  ly = mapy(iy - dnu(2))
                  my = mapy(iy - dnu(2) + dmu(2))
                  do ix=1,nx
                     jx = mapx(ix + dmu(1))
                     kx = mapx(ix + dnu(1))
                     lx = mapx(ix - dnu(1))
                     mx = mapx(ix - dnu(1) + dmu(1))

                     V_x = U_nuxmnu
                     call MatDag(W_xpmu,U_nuxpmumnu)

                     ! T * (L x R) = R T L
                     call MultiplyMatMat(T_x,W_xpmu,dSbydV_muxmnu)
                     call MatPlusMatTimesMat(dSbydU_mux,T_x,V_x)

                     call Matdag(V_x,U_nux)
                     W_xpmu = U_nuxpmu

                     ! T * (L x R) = R T L
                     call MultiplyMatMat(T_x,W_xpmu,dSbydV_muxpnu)
                     call MatPlusMatTimesMat(dSbydU_mux,T_x,V_x)

                     call MultiplyMatMatdag(W_xpmu,U_nuxpmu,U_muxpnu)

                     ! T * (I x R) = R A
                     call MatPlusMatTimesMat(dSbydU_mux,W_xpmu,dSbydV_nux)

                     call MultiplyMatdagMat(V_x,U_muxmnu,U_nuxmnu)

                     ! T * (L x I) = T L
                     call MatPlusMatTimesMat(dSbydU_mux,dSbydV_nuxpmumnu,V_x)

                     ! Vdag

                     call MultiplyMatdagMatdag(W_xpmu,U_nuxpmumnu,U_muxmnu)

                     ! T * (I x R) = R T
                     call MatPlusMatTimesMatDag(dSbydU_mux,W_xpmu,dSbydV_nuxmnu)

                     call MultiplyMatdagMatdag(V_x,U_muxpnu,U_nux)

                     ! T * (L x I) = T L
                     call MatPlusMatDagTimesMat(dSbydU_mux,dSbydV_nuxpmu,V_x)

                  end do
```

```
                end do
             enddo
          end do

       end do
    end do

    do mu=1,nd
       do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx
          dSbydV_mux%Cl = (1.0d0 - alpha_smear)*dSbydV_mux%Cl + (alpha_smear/6.0d0)*dSbydU_mux%Cl
       end do; end do; end do; end do
    end do

#undef dSbydU_mux

#undef U_nux
#undef U_nuxpmu
#undef U_muxpnu
#undef U_nuxpmumnu
#undef U_muxmnu
#undef U_nuxmnu

  end subroutine dSbydVstardVbydU

end module FLICFermions
```

HybridMonteCarlo: Implementation of the multiple time step HMC algorithm.

```
module HybridMonteCarlo

  use GaugeAction
  use GL3Diag
  use FatLinks
  use CloverFmunu
  use FLICFermions
  implicit none

  integer, parameter :: n_a = 8 !number of generators of SU(3)
  logical :: quenched = .false.

  real(dp) :: dt = 0.02d0, dt2 = 0.01d0 !Use multiple time steps, (shorter for the gauge action)

  integer :: n_hmc = 10, n_md = 50, n_md2 = 2

  real(dp) :: beta_md !The beta value to use in updating the configuration.

  !Some global variables to track quantities along the HMC trajectory
  integer, parameter :: nupdate_max = 1000, ntraj_max = 1500
  integer :: ntraj = 0, nupdates = 0, ntherm_hmc, nfixuzero, ngen_hmc

  real(dp), dimension(nupdate_max) :: plaquette, topQ, u0fl
  real(dp), dimension(ntraj_max) :: dH

  logical :: thermalising = .false.

contains

  subroutine HMCTrajectory(U_xd)

    type(colour_matrix), dimension(nxs,nys,nzs,nts,nd) :: U_xd !gauge field
    type(colour_matrix), dimension(nxp,nyp,nzp,ntp,nd) :: P_xd !conjugate momenta
    type(colour_matrix), dimension(nxss,nyss,nzss,ntss,nd) :: Upr_xd !updated gauge field, U'
    type(colour_matrix), dimension(nxp,nyp,nzp,ntp,nd) :: Ppr_xd !updated momenta, P'
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi, eta, eta_pr

    real(DP) :: accept, accept_prob, delta_H, delta_g, delta_KE, delta_pf
    integer :: i_md, i_md2, updates, ic, iterations, cgiter(n_md+1), is

    real(DP) :: tau_int, KEbar, S_gbar, S_pfbar, expdHbar, dHbar, p_acc, tau_intQ
    real(DP) :: S_gU, S_gUpr, S_pf, S_pfpr, KE_P, KE_Ppr, H, Hpr, deltau0, Q
    character(len=6) :: yesno

    !Perform a single HMC trajectory, that is
    !perform n_hmc (accepted) HMC updates of the gauge field
    !such that the final gauge field is uncorrelated with the starting gauge field

    dt2 = dt/n_md2
    beta_md = beta

    iterations = 0
    updates = 0

    call GetUZero(U_xd,uzero)

    mpiprint *, uzero, dt, dt2, n_md, n_md2, traj_minev
```

```
phi = zero_vector

Upr_xd(1:nxs,1:nys,1:nzs,1:nts,:) = U_xd
call ShadowGaugeField(Upr_xd,2)

mpiprint '(2A6,3A10,3A12,3A10)',"#Traj", "#acc", "uzero", "u0bar","u0fl_bar", &
      & "delta KE", "delta S_g", "delta S_pf", "delta H", "Q", "accept?"

do
   !Get the gauge action if starting HMC or if thermalising (as u0_bar is updated)
   if ( (iterations == 0) .or. thermalising ) then
      S_gU = S_gauge(Upr_xd,beta)
   end if

   !Refresh the momenta, for ergodicity.
   call GL3GaussianField(P_xd)
   Ppr_xd = P_xd

   !Perform n_md Molecular Dynamics updates of the coordinates and conjugate momenta
   !To ensure reversibility, we use a leapfrog algorithm, which requires
   !that we perform an initial and final half-step on the momenta.

   eta_pr = zero_vector
   if ( .not. quenched ) then
      call UpdatePseudofermionField(Upr_xd,phi)
      call UpdateFermionicMomenta(Upr_xd, Ppr_xd, phi, eta_pr, dt/2.0d0)
      cgiter(1) = iter_cg
   end if
   eta = eta_pr

   do i_md=2,n_md
      call UpdateGaugeMomenta(Upr_xd, Ppr_xd, dt2/2.0d0)
      do i_md2=2,n_md2
         call UpdateGaugeField(Upr_xd, Ppr_xd, dt2)
         call UpdateGaugeMomenta(Upr_xd, Ppr_xd, dt2)
      end do
      call UpdateGaugeField(Upr_xd, Ppr_xd, dt2)
      call UpdateGaugeMomenta(Upr_xd, Ppr_xd, dt2/2.0d0)

      if ( .not. quenched ) call UpdateFermionicMomenta(Upr_xd, Ppr_xd, phi, eta_pr, dt)

      cgiter(i_md) = iter_cg
   end do

   call UpdateGaugeMomenta(Upr_xd, Ppr_xd, dt2/2.0d0)
   do i_md2=2,n_md2
      call UpdateGaugeField(Upr_xd, Ppr_xd, dt2)
      call UpdateGaugeMomenta(Upr_xd, Ppr_xd, dt2)
   end do
   call UpdateGaugeField(Upr_xd, Ppr_xd, dt2)
   call UpdateGaugeMomenta(Upr_xd, Ppr_xd, dt2/2.0d0)

   if ( .not. quenched ) call UpdateFermionicMomenta(Upr_xd, Ppr_xd, phi, eta_pr, dt/2.0d0)
   cgiter(n_md+1) = iter_cg

   !At the end of each trajectory, enforce the unitarity of U_xd
   call FixSU3(Upr_xd)
   call ShadowGaugeField(Upr_xd,1)
   call ShadowGaugeField(Upr_xd,2)

   KE_P = GetKE(P_xd)
   KE_Ppr = GetKE(Ppr_xd)
   !Average the kinetic energy over the momenta matrices
   KEbar = KE_Ppr/(nlattice*nd)
   delta_KE = KE_Ppr - KE_P

   S_gUpr = S_gauge(Upr_xd,beta)
   !Average the action over the 3 unique plaquettes associated with each link matrix.
   S_gbar = S_gUpr/(nlattice*nd*(nd-1))
   delta_g = S_gUpr - S_gU

   S_pf = real_inner_product(phi,eta)
   S_pfpr = real_inner_product(phi,eta_pr)
   !Average the pseudo fermionic action over each pseudofermionic vector.
   S_pfbar = S_pfpr/(nlattice)
   delta_pf = S_pfpr - S_pf

   Hpr = KE_Ppr + S_gUpr + S_pfpr
   H = KE_P + S_gU + S_pf
   delta_H = Hpr - H

   if ( i_am_root ) call random_number(accept)

   call BroadcastReal(accept,mpi_root_rank)

   if ( delta_H <= 0 ) then
      accept_prob = 1.0d0
```

```
    else
       accept_prob = exp(-delta_H)
    end if

    !Calculate the (gauge) topological charge.
    call GetUZero(Upr_xd, uzero)
    call APESmearLinks(Upr_xd, UFL_xd, alpha_smear, ape_sweeps)
    call GetUZero(UFL_xd, uzerofl)

    call CalculateFmunuClover(F_munu, UFL_xd)
    Q = GetTopQ(F_munu)/(uzerofl**4)

    if ( accept < accept_prob ) then

       U_xd = Upr_xd(1:nxs,1:nys,1:nzs,1:nts,:)
       S_gU = S_gUpr
       yesno = "accept"

       updates = updates + 1
       nupdates = nupdates + 1

       plaquette(nupdates) = uzero**4
       topQ(nupdates) = Q
       u0fl(nupdates) = uzerofl

       if ( thermalising ) then
          if ( nupdates == nfixuzero/3 ) alpha_smear = 0.7d0
          if ( nupdates < nfixuzero ) then
             u0_bar = uzero
             u0fl_bar = uzerofl
          else if ( nupdates == nfixuzero ) then
             u0_bar = sum(plaquette(nupdates-9:nupdates)**0.25d0)/10.0d0
             u0fl_bar = sum(u0fl(nupdates-9:nupdates))/10.0d0
          end if
       end if

    else
       Upr_xd(1:nxs,1:nys,1:nzs,1:nts,:) = U_xd
       call ShadowGaugeField(Upr_xd,2)

       yesno = "reject"
    end if

    iterations = iterations + 1
    ntraj = ntraj + 1

    dH(ntraj) = delta_H

    if (i_am_root) write(*,'(2I5,3F10.7,3F12.4,2F10.5,A8,3I4,F10.7)') iterations,updates,uzero,u0_bar,
         u0fl_bar,&
         & delta_KE,delta_g,delta_pf,delta_H, Q, yesno, &
         & minval(cgiter), maxval(cgiter), sum(cgiter)/(n_md+1), traj_minev

    traj_minev = 1.0d0

    if ( updates >= n_hmc) exit
    if ( iterations >= 3*n_hmc ) exit

  end do

  lastPlaq = m_f
  plaqbarAvg = u0fl_bar
  uzero = u0_bar

  !Output some info about the HMC trajectory

  p_acc = real(nupdates)/ntraj

  dHbar = sum(dH(1+nfixuzero:ntraj))/(ntraj - nfixuzero)
  expdHbar = sum(exp(-dH(1+nfixuzero:ntraj)))/(ntraj - nfixuzero)
  tau_int = IntegratedAutoCorrelation(nupdates,plaquette(1:nupdates))
  tau_intQ = IntegratedAutoCorrelation(nupdates,topQ(1:nupdates))
  deltau0 = maxval(abs(u0_bar - plaquette(1:nupdates)**0.25d0))

  mpiprint '(2A6,7A10)', "traj", "acc", "p_acc", "<dH>", "<e^-dH>", "tau_int^P", "tau_int^Q","u0_bar", "
       delta u0"
  mpiprint '(2I6,5F10.5,2F10.7)', ntraj, nupdates, p_acc, dHbar, expdHbar, tau_int, tau_intQ, u0_bar,
       deltau0

end subroutine HMCTrajectory

function IntegratedAutoCorrelation(t_mc,curlyO) result (tau_int)

  integer :: t_mc
  real(dp), dimension(t_mc) :: curlyO
  real(dp) :: tau_int

  real(dp) :: C_t, C_0
```

```
  integer :: t

  C_0 = AutoCorrelation(t_mc,curlyO,0)
  tau_int = 0.5d0
  do t=1,t_mc-1
     C_t = AutoCorrelation(t_mc,curlyO,t)
     tau_int = tau_int + C_t/C_0
  end do

end function IntegratedAutoCorrelation

function AutoCorrelation(t_mc,curlyO,t) result (C_t)

  integer :: t_mc
  real(dp), dimension(t_mc) :: curlyO
  integer :: t
  real(dp) :: C_t

  real(dp) :: Osq_av, Oav_sq

  Osq_av = sum(curlyO(1:t_mc-t)*curlyO(t+1:t_mc))/(t_mc-t)
  Oav_sq = (sum(curlyO(1:t_mc-t))*sum(curlyO(t+1:t_mc)))/(t_mc-t)**2

  C_t = Osq_av - Oav_sq

end function AutoCorrelation

subroutine UpdateGaugeField(U_xd, P_xd, dt)

  type(colour_matrix), dimension(nxss,nyss,nzss,ntss,nd) :: U_xd
  type(colour_matrix), dimension(nxp,nyp,nzp,ntp,nd) :: P_xd
  real(DP), intent(in) :: dt

  type(colour_matrix) :: V_xd, W_xd, X_xd, I_xd
  type(real_vector)   :: lambda_xd
  type(colour_vector) :: theta_xd

  integer :: ix,iy,iz,it,mu
  real(dp) :: nonU

  !P_xd is a Hermitian field that is the sum of SU(3) generator matrices.
  !To update U_xd, we must multiply by an SU(3) matrix to stay within SU(3).
  !We exponentiate I*dt*P_xd/2 by diagonalising, exponentiating its eigenvalues,
  !lambda, and then undiagonalising the diagonal matrix D[lambda].
  !This exponential is then an SU(3) matrix.

  do mu=1,nd
     do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx

        call DiagonaliseMat(P_xd(ix,iy,iz,it,mu), lambda_xd, V_xd)

        theta_xd%Cl = exp((I*dt)*lambda_xd%Cl)

        !Undiagonalise back to P_xd basis, by setting W_xd = V_xd*D[theta_xd]*V_xd^dag,
        !where V_xd is the matrix of eigenvectors.

        !Note: [M*D]_ij = M_ij*theta_j
        call MultiplyMatDiagMat(W_xd,V_xd,theta_xd)

        call MultiplyMatMatDag(X_xd,W_xd,V_xd)

        V_xd = U_xd(ix,iy,iz,it,mu)
        call MultiplyMatMat(U_xd(ix,iy,iz,it,mu),X_xd,V_xd)

     end do; end do; end do; end do
  end do

  call ShadowGaugeField(U_xd,1)
  call ShadowGaugeField(U_xd,2)

end subroutine UpdateGaugeField

subroutine UpdateGaugeMomenta(U_xd, P_xd, dt)

  type(colour_matrix), dimension(nxss,nyss,nzss,ntss,nd) :: U_xd
  type(colour_matrix), dimension(nxp,nyp,nzp,ntp,nd) :: P_xd
  real(DP), intent(in) :: dt

  type(colour_matrix), dimension(nxp,nyp,nzp,ntp,nd) :: dS_gbydU

  type(colour_matrix) :: V_xd,Pdot_xd

  integer :: ix,iy,iz,it,mu
  integer :: ic, jc

  real(dp) :: TrPdot

  integer :: t0, t1
```

```
    call GetdS_gbydU(dS_gbydU, U_xd, beta_md)

    do mu=1,nd
       do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx

          !P_xd is a Hermitian field that is the sum of SU(3) generator matrices (i.e. traceless).
          !To update P_xd, we must add an Hermitian, traceless matrix, Pdot_xd.
          !Set Pdot_xd = i * U * (dSbydU - U^dag * dSbydU^dag U^dag)
          !             = ( i * U * dSbydU + h.c. )

          call MultiplyMatMat(V_xd, U_xd(ix,iy,iz,it,mu), dS_gbydU(ix,iy,iz,it,mu))


          do jc=1,nc; do ic=1,nc
             Pdot_xd%Cl(ic,jc) = I*( V_xd%Cl(ic,jc) - conjg(V_xd%Cl(jc,ic)) )
          end do; end do

          TrPdot = real_trace(Pdot_xd)

          Pdot_xd%Cl(1,1) = Pdot_xd%Cl(1,1) - TrPdot/3.0d0
          Pdot_xd%Cl(2,2) = Pdot_xd%Cl(2,2) - TrPdot/3.0d0
          Pdot_xd%Cl(3,3) = Pdot_xd%Cl(3,3) - TrPdot/3.0d0

          P_xd(ix,iy,iz,it,mu)%Cl = P_xd(ix,iy,iz,it,mu)%Cl + dt*Pdot_xd%Cl

       end do; end do; end do; end do
    end do

  end subroutine UpdateGaugeMomenta

  subroutine UpdateFermionicMomenta(U_xd, P_xd, phi, eta, dt)

    type(colour_matrix), dimension(nxss,nyss,nzss,ntss,nd) :: U_xd
    type(colour_matrix), dimension(nxp,nyp,nzp,ntp,nd) :: P_xd
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi, eta
    real(DP), intent(in) :: dt

    type(colour_matrix), dimension(nxp,nyp,nzp,ntp,nd) :: dS_pfbydU
    type(colour_matrix) :: V_xd, Pdot_xd
    integer :: ix,iy,iz,it,mu
    integer :: ic, jc

    real(dp) :: TrPdot

    integer :: t0, t1

    call InitialiseFermionMatrix(U_xd)

    call PseudoFermionMatrixOperate(phi, eta)

    call GetdS_pfbydU(dS_pfbydU, U_xd, eta)

    do mu=1,nd
       do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx

          !P_xd is a Hermitian field that is the sum of SU(3) generator matrices (i.e. traceless).
          !To update P_xd, we must add an Hermitian, traceless matrix, Pdot_xd.
          !Set Pdot_xd = i * U * (dSbydU - U^dag * dSbydU^dag U^dag)
          !             = ( i * U * dSbydU + h.c. )

          call MultiplyMatMat(V_xd, U_xd(ix,iy,iz,it,mu), dS_pfbydU(ix,iy,iz,it,mu))

          do jc=1,nc; do ic=1,nc
             Pdot_xd%Cl(ic,jc) = I*( V_xd%Cl(ic,jc) - conjg(V_xd%Cl(jc,ic)) )
          end do; end do

          TrPdot = real_trace(Pdot_xd)

          Pdot_xd%Cl(1,1) = Pdot_xd%Cl(1,1) - TrPdot/3.0d0
          Pdot_xd%Cl(2,2) = Pdot_xd%Cl(2,2) - TrPdot/3.0d0
          Pdot_xd%Cl(3,3) = Pdot_xd%Cl(3,3) - TrPdot/3.0d0

          P_xd(ix,iy,iz,it,mu)%Cl = P_xd(ix,iy,iz,it,mu)%Cl + dt*Pdot_xd%Cl

       end do; end do; end do; end do
    end do

  end subroutine UpdateFermionicMomenta

  subroutine UpdatePseudofermionField(U_xd, phi)

    type(colour_matrix), dimension(nxss,nyss,nzss,ntss,nd) :: U_xd
    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: phi

    type(colour_vector), dimension(nxp,nyp,nzp,ntp,ns) :: xi

    !The pseudo fermion fields have no dynamic, they are simply an auxillary field used to calculate
```

```
  !the fermionic determinant , det(M^dag M)=\int Dphi*Dphi e^(-S_pf), where S_pf=phi* (M^dag M)^-1 phi
  !So we wish to generate phi according to the distribution e^(-S_pf) = e^(-phi* M^-1 (M^dag)^-1 phi )
  !Note that xi = (M^dag)^-1 phi is Gaussian distributed , P(xi) = e^-(xi* xi) and therefore easily
  !generated, and hence to obtain the appropriate distribution for phi, we set phi = M^dag xi.

  call InitialiseFermionMatrix(U_xd)

  call ComplexGaussianField(xi)
  !Calculate phi = M^dagger xi = gamma_5 M gamma_5 xi  = H gamma_5 xi

  call FermionMatrixDagMultiply(xi,phi)

end subroutine UpdatePseudofermionField

function GetKE(P_xd) result (KE)

  type(colour_matrix), dimension(nxp,nyp,nzp,ntp,nd) :: P_xd
  real(dp) :: KE

  real(dp) :: TrPsq, TrPsqpp
  integer :: ix,iy,iz,it,mu

  !Calculate the Kinetic energy (in 5D) of the conjugate momenta.
  TrPsqpp = 0.0d0
  do mu=1,nd
     do it=1,nt; do iz=1,nz; do iy=1,ny; do ix=1,nx

        call RealTraceMultMatMat(TrPsq,P_xd(ix,iy,iz,it,mu),P_xd(ix,iy,iz,it,mu))
        TrPsqpp = TrPsqpp + TrPsq

     end do; end do; end do; end do
  end do

  call AllSumReal(TrPsqpp,TrPsq)
  KE = 0.5d0*TrPsq

end function GetKE

! let x in [0,1] be a random number field , with uniform distribution , p(x)dx = dx
! being the probability of generating a random number between x and x+dx.
! let y(x) be a random number field , then |p(y)dy| = |p(x)dx|, thus
! p(y) = p(x)|dx/dy|. Similarly, p(y1(x_i),y2(x_i)) = p(x1,x2)|d(x_i)/d(y_i)|.
! So we can generate 2 real Gaussian fields by making use of the following 2-d fields
! y1(x1,x2) = sqrt(-2*ln(x1))*cos(2*pi*x2), y2(x1,x2) = sqrt(-2*ln(x1))*sin(2*pi*x2). (Num Rec)

subroutine ComplexGaussianField(xi)

  type(colour_vector), dimension(:,:,:,:,:) :: xi
  real(dp) :: alpha, theta !random number fields
  integer :: ix,iy,iz,it,is,ic

  !Return complex random numbers according to P(xi) = exp(-abs(xi)**2)/pi = P(Re(xi))P(Im(xi))

  do is=1,ns
     do it=1,nt; do iz=1,nz; do iy=1,ny; do ix = 1,nx
        do ic=1,nc

           call random_number(alpha)
           call random_number(theta)

           alpha = sqrt(-log(alpha))
           theta=two_pi*theta
           xi(ix,iy,iz,it,is)%Cl(ic) = cmplx(alpha*cos(theta),alpha*sin(theta),dc)

        end do
     end do; end do; end do; end do
  end do

end subroutine ComplexGaussianField

subroutine GL3GaussianField(P_xd)

  type(colour_matrix), dimension(:,:,:,:,:) :: P_xd

  real(dp) :: alpha, theta !random number fields
  real(dp), dimension(n_a) :: omega_a
  integer :: ix,iy,iz,it,mu,i_a

  do mu=1,nd
     do it=1,nt; do iz=1,nz; do iy=1,ny; do ix = 1,nx

        !Returns real random numbers according to P(tau) = exp(-tau**2)/sqrt(pi)

        do i_a=1,n_a

           call random_number(alpha)
           call random_number(theta)
```

```
        alpha = sqrt(-log(alpha))
        theta=two_pi*theta
        omega_a(i_a) = alpha*cos(theta)
      end do

      !Generate an Hermitian colour matrix field based on the Gaussian momenta omega_a

      !Calculate the exponent, sum_a omega_a*lambda_a
      !where lambda_a are the generators of SU(3), the Gellmann matrices
      P_xd(ix,iy,iz,it,mu)%Cl(1,1) =        omega_a(3) + omega_a(8)/sqrt(3.0d0)
      P_xd(ix,iy,iz,it,mu)%Cl(1,2) = cmplx(omega_a(1), -omega_a(2),dc)
      P_xd(ix,iy,iz,it,mu)%Cl(1,3) = cmplx(omega_a(4), -omega_a(5),dc)

      P_xd(ix,iy,iz,it,mu)%Cl(2,1) = cmplx(omega_a(1),  omega_a(2),dc)
      P_xd(ix,iy,iz,it,mu)%Cl(2,2) =       -omega_a(3) + omega_a(8)/sqrt(3.0d0)
      P_xd(ix,iy,iz,it,mu)%Cl(2,3) = cmplx(omega_a(6), -omega_a(7),dc)

      P_xd(ix,iy,iz,it,mu)%Cl(3,1) = cmplx(omega_a(4),  omega_a(5),dc)
      P_xd(ix,iy,iz,it,mu)%Cl(3,2) = cmplx(omega_a(6),  omega_a(7),dc)
      P_xd(ix,iy,iz,it,mu)%Cl(3,3) = -2.0d0*omega_a(8)/sqrt(3.0d0)

    end do; end do; end do; end do
  end do

end subroutine GL3GaussianField

subroutine RandomSU3Field(U_xd)

  type(colour_matrix), dimension(:,:,:,:,:) :: U_xd

  real(dp), dimension(n_a) :: omega_a
  type(colour_matrix) :: H_xd, V_xd
  type(real_vector)   :: lambda_xd
  type(colour_vector) :: theta_xd
  integer :: ix,iy,iz,it,mu,i_a

  do mu=1,nd
     do it=1,nt; do iz=1,nz; do iy=1,ny; do ix = 1,nx

        do i_a=1,n_a
           call random_number(omega_a(i_a))
        end do

        !Generate an Hermitian colour matrix field based on omega_a

        !Calculate the exponent, sum_a omega_a*lambda_a
        !where lambda_a are the generators of SU(3), the Gellmann matrices
        H_xd%Cl(1,1) =        omega_a(3) + omega_a(8)/sqrt(3.0d0)
        H_xd%Cl(1,2) = cmplx(omega_a(1), -omega_a(2),dc)
        H_xd%Cl(1,3) = cmplx(omega_a(4), -omega_a(5),dc)

        H_xd%Cl(2,1) = cmplx(omega_a(1),  omega_a(2),dc)
        H_xd%Cl(2,2) =       -omega_a(3) + omega_a(8)/sqrt(3.0d0)
        H_xd%Cl(2,3) = cmplx(omega_a(6), -omega_a(7),dc)

        H_xd%Cl(3,1) = cmplx(omega_a(4),  omega_a(5),dc)
        H_xd%Cl(3,2) = cmplx(omega_a(6),  omega_a(7),dc)
        H_xd%Cl(3,3) = -2.0d0*omega_a(8)/sqrt(3.0d0)

        !Generate and SU3 colour matrix field based on H_xd
        call DiagonaliseMat(H_xd, lambda_xd, V_xd)

        theta_xd%Cl = exp((two_pi*I*0.5d0)*lambda_xd%Cl)

        !Undiagonalise back to H_xd basis, by setting U_xd = V_xd*D[theta_xd]*V_xd^dag.
        !where V_xd is the matrix of eigenvectors.

        !Note: [M*D]_ij = M_ij*theta_j
        call MultiplyMatDiagMat(H_xd,V_xd,theta_xd)

        call MultiplyMatMatDag(U_xd(ix,iy,iz,it,mu),H_xd,V_xd)

     end do; end do; end do; end do
  end do

  call FixSU3(U_xd)

end subroutine RandomSU3Field

end module HybridMonteCarlo
```

HMC: The Hybrid Monte Carlo main program.

```
program HMC

  use MPIRandomSeed
  use HybridMonteCarlo
```

```
use Timer
use FLICFermions
implicit none

integer :: icfg, icfg0, ncfg, ic, starttype, is, n_mdgen, t_rate, sendrecv_req(2)
integer :: sendrecv_status(nmpi_status,2)
integer :: ix,iy,iz,it
real(dp) :: S, t_in, t_out
character(len=3) :: config
character(len=128) :: fileprefix, filename

call InitialiseMPI

call InitShadowGaugeField
call InitShadowFermionField

call InitTimer

alpha_smear = 0.7d0
ape_sweeps = 4

allocate(Un_xd(nxs,nys,nzs,nts,nd,ape_sweeps))

mpiprint *, "HMC"

mpiprint *, "Configuration prefix:"
read (*,'(A128)') fileprefix
mpiprint *, "Quenched?:"
read (*,*) quenched
mpiprint *, "Gauge action:"
read (*,*) actiontype
mpiprint *, "Beta:"
read (*,*) beta
mpiprint *, "Kappa:"
read (*,*) kappa
mpiprint *, "Inital cfg #:"
read (*,*) icfg0
mpiprint *, "Final  cfg #:"
read (*,*) ncfg
mpiprint *, "Hot(1), Equilibrium(0), Other cfg(-1) start:"
read (*,*) starttype
mpiprint *, "Time step:"
read (*,*) dt
mpiprint *, "# Outer MD trajectories:"
read (*,*) n_md
mpiprint *, "# Inner MD Trajectories:"
read (*,*) n_md2
mpiprint *, "#HMC Trajectories per cfg:"
read (*,*) ngen_hmc

if (quenched) then
   m_f = 0.0d0
else
   m_f = 4.0d0 - 0.5d0/kappa
end if

if ( dt <= 0.0d0 ) then
   dt = 1.0d0/n_md
end if

mpiprint *, "HMC Simulation Parameters"
mpiprint *, "Quenched? ", quenched
mpiprint *, "Gauge action: ", actiontype
mpiprint *, "Beta: ", beta
mpiprint *, "m_f: ", m_f, " kappa: ", kappa
mpiprint *, "icfg: ", icfg0, " ncfg: ", ncfg
mpiprint *, "dt: ", dt, " n_md: ", n_md, " n_hmc: ", ngen_hmc

call mpi_random_seed(31459+icfg0)

select case(starttype)
case(-1)
   mpiprint *, "quenched start"
   mpiprint *, "Enter starting cfg"
   read (*,'(A128)') filename
   call ReadGaugeField(filename,U_xd)
   u0_bar = uzero
   mpiprint *, "read uzero= ", uzero, " u0_bar = ", u0_bar
   mpiprint *, beta,lastPlaq,plaqbarAvg,uzero
case(0)
   mpiprint *, "Starting from an equilibrium configuration"
   mpiprint *, "Enter starting cfg"
   read (*,'(A128)') filename
   call ReadGaugeField(filename,U_xd)
   u0_bar = uzero
   u0fl_bar = plaqbarAvg
   mpiprint *, "read uzero= ", uzero, " u0_bar = ", u0_bar
   n_hmc = 10
```

```
  case(1)
     mpiprint *, "hot start"
     call RandomSU3Field(U_xd)
     call ShadowGaugeField(U_xd,1)
     u0_bar = 1.0d0
     alpha_smear = 0.5d0
  end select

  if ( starttype /= 0 ) then

     thermalising = .true.

     ntherm_hmc = 12*ngen_hmc
     nfixuzero = 6*ngen_hmc
     n_hmc = ntherm_hmc

     call HMCTrajectory(U_xd)

     filename = fileprefix(1:len_trim(fileprefix))//"000"
     mpiprint *, "writing ", filename

     icfg = 0
     call WriteGaugeField(filename,U_xd,icfg)

     thermalising = .false.
     nupdates = 0
     ntraj = 0
     nfixuzero = 0

  end if

  n_hmc = ngen_hmc

  do icfg=icfg0,ncfg

     call HMCTrajectory(U_xd)

     write (config,'(I3.3)') icfg
     filename = fileprefix(1:len_trim(fileprefix))//config
     mpiprint '(A,A128)', "writing ", filename

     call WriteGaugeField(filename,U_xd,icfg)

  end do

  deallocate(Un_xd)

  call FinaliseMPI

end program HMC
```

Bibliography

[1] M. E. Peskin and D. V. Schroeder, "An introduction to quantum field theory," Reading, USA: Addison-Wesley (1995) 842 p.

[2] K. G. Wilson, "Confinement of quarks," *Phys. Rev.*, vol. D10, pp. 2445–2459, 1974.

[3] H. Neuberger, "Bounds on the Wilson Dirac operator," *Phys. Rev.*, vol. D61, p. 085015, 2000, hep-lat/9911004.

[4] I. Montvay and G. Münster, "Quantum fields on a lattice," Cambridge, UK: Univ. Pr. (1994) 491 p. (Cambridge monographs on mathematical physics).

[5] H. J. Rothe, "Lattice gauge theories: An introduction," *World Sci. Lect. Notes Phys.*, vol. 59, pp. 1–512, 1997.

[6] G. P. Lepage and P. B. Mackenzie, "On the viability of lattice perturbation theory," *Phys. Rev.*, vol. D48, pp. 2250–2264, 1993, hep-lat/9209022.

[7] M. Luscher and P. Weisz, "On-shell improved lattice gauge theories," *Commun. Math. Phys.*, vol. 97, p. 59, 1985.

[8] P. de Forcrand *et al.*, "Renormalization group flow of SU(3) lattice gauge theory: Numerical studies in a two coupling space," *Nucl. Phys.*, vol. B577, pp. 263–278, 2000, hep-lat/9911033.

[9] A. Borici and R. Rosenfelder, "Scaling in SU(3) theory with a MCRG improved lattice action," *Nucl. Phys. Proc. Suppl.*, vol. 63, pp. 925–927, 1998, hep-lat/9711035.

[10] P. de Forcrand *et al.*, "Search for effective lattice action of pure QCD," *Nucl. Phys. Proc. Suppl.*, vol. 53, pp. 938–941, 1997, hep-lat/9608094.

[11] T. Takaishi, "Heavy quark potential and effective actions on blocked configurations," *Phys. Rev.*, vol. D54, pp. 1050–1053, 1996.

[12] B. Sheikholeslami and R. Wohlert, "Improved continuum limit lattice action for QCD with Wilson fermions," *Nucl. Phys.*, vol. B259, p. 572, 1985.

[13] M. Luscher, "Advanced lattice QCD," 1998, hep-lat/9802029.

[14] D. B. Leinweber *et al.*, "FLIC fermions and hadron phenomenology," 2002, nucl-th/0211014.

[15] S. O. Bilson-Thompson, D. B. Leinweber, and A. G. Williams, "Highly-improved lattice field-strength tensor," *Ann. Phys.*, vol. 304, pp. 1–21, 2003, hep-lat/0203008.

[16] B. Berg, "Dislocations and topological background in the lattice O(3) sigma model," *Phys. Lett.*, vol. B104, p. 475, 1981.

[17] M. Teper, "Instantons in the quantized SU(2) vacuum: A lattice Monte Carlo investigation," *Phys. Lett.*, vol. B162, p. 357, 1985.

[18] M. Teper, "Axial anomaly suppression (and axial U(1) symmetry restoration) at high temperatures: A lattice Monte Carlo study," *Phys. Lett.*, vol. B171, p. 81, 1986.

[19] M. Teper, "The topological susceptibility in SU(2) lattice gauge theory: An exploratory study," *Phys. Lett.*, vol. B171, p. 86, 1986.

[20] E. M. Ilgenfritz, M. L. Laursen, G. Schierholz, M. Muller-Preussker, and H. Schiller, "First evidence for the existence of instantons in the quantized SU(2) lattice vacuum," *Nucl. Phys.*, vol. B268, p. 693, 1986.

[21] M. Falcioni, M. L. Paciello, G. Parisi, and B. Taglienti, "Again on SU(3) glueball mass," *Nucl. Phys.*, vol. B251, pp. 624–632, 1985.

[22] M. Albanese *et al.*, "Glueball masses and string tension in lattice QCD," *Phys. Lett.*, vol. B192, p. 163, 1987.

[23] F. D. R. Bonnet, D. B. Leinweber, A. G. Williams, and J. M. Zanotti, "Improved smoothing algorithms for lattice gauge theory," 2001, hep-lat/0106023.

[24] M. C. Chu, J. M. Grandy, S. Huang, and J. W. Negele, "Evidence for the role of instantons in hadron structure from lattice QCD," *Phys. Rev.*, vol. D49, pp. 6039–6050, 1994, hep-lat/9312071.

[25] F. D. R. Bonnet, P. Fitzhenry, D. B. Leinweber, M. R. Stanford, and A. G. Williams, "Calibration of smearing and cooling algorithms in SU(3)- color gauge theory," *Phys. Rev.*, vol. D62, p. 094509, 2000, hep-lat/0001018.

[26] M. C. Chu, J. M. Grandy, S. Huang, and J. W. Negele, "Evidence for the role of instantons in hadron structure from lattice QCD," *Phys. Rev.*, vol. D49, pp. 6039–6050, 1994, hep-lat/9312071.

[27] T. DeGrand, A. Hasenfratz, and T. G. Kovacs, "Instantons and exceptional configurations with the clover action," *Nucl. Phys.*, vol. B547, pp. 259–280, 1999, hep-lat/9810061.

[28] J. M. Zanotti *et al.*, "Hadron masses from novel fat-link fermion actions," 2001, hep-lat/0110216.

[29] H. B. Nielsen and M. Ninomiya, "Absence of neutrinos on a lattice. 1. proof by homotopy theory," *Nucl. Phys.*, vol. B185, p. 20, 1981.

[30] H. B. Nielsen and M. Ninomiya, "Absence of neutrinos on a lattice. 2. intuitive topological proof," *Nucl. Phys.*, vol. B193, p. 173, 1981.

[31] H. B. Nielsen and M. Ninomiya, "No go theorem for regularizing chiral fermions," *Phys. Lett.*, vol. B105, p. 219, 1981.

[32] P. H. Ginsparg and K. G. Wilson, "A remnant of chiral symmetry on the lattice," *Phys. Rev.*, vol. D25, p. 2649, 1982.

[33] D. B. Kaplan, "A method for simulating chiral fermions on the lattice," *Phys. Lett.*, vol. B288, pp. 342–347, 1992, hep-lat/9206013.

[34] J. Callan, Curtis G. and J. A. Harvey, "Anomalies and fermion zero modes on strings and domain walls," *Nucl. Phys.*, vol. B250, p. 427, 1985.

[35] S. A. Frolov and A. A. Slavnov, "An invariant regularization of the standard model," *Phys. Lett.*, vol. B309, pp. 344–350, 1993.

[36] R. Narayanan and H. Neuberger, "Infinitely many regulator fields for chiral fermions," *Phys. Lett.*, vol. B302, pp. 62–69, 1993, hep-lat/9212019.

[37] R. Narayanan and H. Neuberger, "Chiral determinant as an overlap of two vacua," *Nucl. Phys.*, vol. B412, pp. 574–606, 1994, hep-lat/9307006.

[38] R. Narayanan and H. Neuberger, "Chiral fermions on the lattice," *Phys. Rev. Lett.*, vol. 71, pp. 3251–3254, 1993, hep-lat/9308011.

[39] R. Narayanan and H. Neuberger, "A construction of lattice chiral gauge theories," *Nucl. Phys.*, vol. B443, pp. 305–385, 1995, hep-th/9411108.

[40] H. Neuberger, "Exactly massless quarks on the lattice," *Phys. Lett.*, vol. B417, pp. 141–144, 1998, hep-lat/9707022.

[41] Y. Kikukawa and H. Neuberger, "Overlap in odd dimensions," *Nucl. Phys.*, vol. B513, pp. 735–757, 1998, hep-lat/9707016.

[42] H. Neuberger, "Exact chiral symmetry on the lattice," *Ann. Rev. Nucl. Part. Sci.*, vol. 51, pp. 23–52, 2001, hep-lat/0101006.

[43] H. Neuberger, "More about exactly massless quarks on the lattice," *Phys. Lett.*, vol. B427, pp. 353–355, 1998, hep-lat/9801031.

[44] D.-J. Kusterer, J. Hedditch, W. Kamleh, D. B. Leinweber, and A. G. Williams, "Low-lying eigenmodes of the Wilson-Dirac operator and correlations with topological objects," *Nucl. Phys.*, vol. B628, pp. 253–269, 2002, hep-lat/0111029.

[45] P. Hasenfratz, V. Laliena, and F. Niedermayer, "The index theorem in QCD with a finite cut-off," *Phys. Lett.*, vol. B427, pp. 125–131, 1998, hep-lat/9801021.

[46] R. G. Edwards, U. M. Heller, and R. Narayanan, "A study of practical implementations of the Overlap-Dirac operator in four dimensions," *Nucl. Phys.*, vol. B540, pp. 457–471, 1999, hep-lat/9807017.

[47] A. Bode, U. M. Heller, R. G. Edwards, and R. Narayanan, "First experiences with HMC for dynamical overlap fermions," 1999, hep-lat/9912043.

[48] H. Neuberger, "Vector like gauge theories with almost massless fermions on the lattice," *Phys. Rev.*, vol. D57, pp. 5417–5433, 1998, hep-lat/9710089.

[49] H. Neuberger, "The overlap Dirac operator," 1999, hep-lat/9910040.

[50] P. Hernandez, K. Jansen, and M. Luscher, "Locality properties of Neuberger's lattice Dirac operator," *Nucl. Phys.*, vol. B552, pp. 363–378, 1999, hep-lat/9808010.

[51] H. Neuberger, "A practical implementation of the Overlap-Dirac operator," *Phys. Rev. Lett.*, vol. 81, pp. 4060–4062, 1998, hep-lat/9806025.

[52] T.-W. Chiu, T.-H. Hsieh, C.-H. Huang, and T.-R. Huang, "A note on the Zolotarev optimal rational approximation for the overlap Dirac operator," *Phys. Rev.*, vol. D66, p. 114502, 2002, hep-lat/0206007.

[53] B. Jegerlehner, "Krylov space solvers for shifted linear systems," 1996, hep-lat/9612014.

[54] R. G. Edwards, U. M. Heller, and R. Narayanan, "Chiral fermions on the lattice," 1999, hep-lat/9905028.

[55] K. Fujikawa, "A continuum limit of the chiral Jacobian in lattice gauge theory," *Nucl. Phys.*, vol. B546, pp. 480–494, 1999, hep-th/9811235.

[56] H. Suzuki, "Simple evaluation of chiral Jacobian with the overlap Dirac operator," *Prog. Theor. Phys.*, vol. 102, pp. 141–147, 1999, hep-th/9812019.

[57] D. H. Adams, "Axial anomaly and topological charge in lattice gauge theory with overlap-Dirac operator," 1998, hep-lat/9812003.

[58] Y. Kikukawa and A. Yamada, "Weak coupling expansion of massless QCD with a Ginsparg- Wilson fermion and axial U(1) anomaly," *Phys. Lett.*, vol. B448, pp. 265–274, 1999, hep-lat/9806013.

[59] D. H. Adams, "On the continuum limit of fermionic topological charge in lattice gauge theory," *J. Math. Phys.*, vol. 42, pp. 5522–5533, 2001, hep-lat/0009026.

[60] W. Bietenholz, I. Hip, and K. Schilling, "Fast evaluation and locality of overlap fermions," *Nucl. Phys. Proc. Suppl.*, vol. 106, pp. 829–831, 2002, hep-lat/0111027.

[61] W. Bietenholz, "Solutions of the Ginsparg-Wilson relation and improved domain wall fermions," *Eur. Phys. J.*, vol. C6, pp. 537–547, 1999, hep-lat/9803023.

[62] P. Hasenfratz, S. Hauswirth, K. Holland, T. Jorg, and F. Niedermayer, "First results from a parametrized fixed-point QCD action," *Nucl. Phys. Proc. Suppl.*, vol. 106, pp. 799–801, 2002, hep-lat/0109004.

[63] P. Hasenfratz *et al.*, "The construction of generalized Dirac operators on the lattice," *Int. J. Mod. Phys.*, vol. C12, pp. 691–708, 2001, hep-lat/0003013.

[64] T. DeGrand, "A variant approach to the overlap action," *Phys. Rev.*, vol. D63, p. 034503, 2001, hep-lat/0007046.

[65] R. G. Edwards, U. M. Heller, and R. Narayanan, "Spectral flow, chiral condensate and topology in lattice QCD," *Nucl. Phys.*, vol. B535, pp. 403–422, 1998, hep-lat/9802016.

[66] M. G. Alford, T. R. Klassen, and G. P. Lepage, "Improving lattice quark actions," *Nucl. Phys.*, vol. B496, pp. 377–407, 1997, hep-lat/9611010.

[67] T. Kalkreuter and H. Simma, "An accelerated conjugate gradient algorithm to compute low lying eigenvalues: A study for the Dirac operator in SU(2) lattice QCD," *Comput. Phys. Commun.*, vol. 93, pp. 33–47, 1996, hep-lat/9507023.

[68] R. G. Edwards and U. M. Heller, "Are topological charge fluctuations in QCD instanton dominated?," *Phys. Rev.*, vol. D65, p. 014505, 2002, hep-lat/0105004.

[69] S. J. Dong *et al.*, "Chiral properties of pseudoscalar mesons on a quenched 20**4 lattice with overlap fermions," *Phys. Rev.*, vol. D65, p. 054507, 2002, hep-lat/0108020.

[70] S. Duane, A. D. Kennedy, B. J. Pendleton, and D. Roweth, "Hybrid Monte Carlo," *Phys. Lett.*, vol. B195, pp. 216–222, 1987.

[71] T. Takaishi and P. de Forcrand, "Odd-flavor Hybrid Monte Carlo algorithm for lattice QCD," *Int. J. Mod. Phys.*, vol. C13, pp. 343–366, 2002, hep-lat/0108012.

[72] M. Luscher, "A new approach to the problem of dynamical quarks in numerical simulations of lattice QCD," *Nucl. Phys.*, vol. B418, pp. 637–648, 1994, hep-lat/9311007.

[73] Y. Liang, K. F. Liu, B. A. Li, S. J. Dong, and K. Ishikawa, "Lattice calculation of glueball matrix elements," *Phys. Lett.*, vol. B307, pp. 375–382, 1993, hep-lat/9304011.

[74] A. Ali Khan *et al.*, "Light hadron spectroscopy with two flavors of dynamical quarks on the lattice," *Phys. Rev.*, vol. D65, p. 054505, 2002, hep-lat/0105015.

[75] A. Alexandru and A. Hasenfratz, "Partial-global stochastic Metropolis update for dynamical smeared link fermions," *Phys. Rev.*, vol. D66, p. 094502, 2002, hep-lat/0207014.

[76] W. Kamleh, D. H. Adams, D. B. Leinweber, and A. G. Williams, "Accelerated overlap fermions," *Phys. Rev.*, vol. D66, p. 014501, 2002, hep-lat/0112041.

[77] T. DeGrand, "A variant approach to the overlap action," *Phys. Rev.*, vol. D63, p. 034503, 2001, hep-lat/0007046.

[78] W. Bietenholz, "Convergence rate and locality of improved overlap fermions," *Nucl. Phys.*, vol. B644, pp. 223–247, 2002, hep-lat/0204016.

[79] T. G. Kovacs, "Locality and topology with fat link overlap actions," *Phys. Rev.*, vol. D67, p. 094501, 2003, hep-lat/0209125.

[80] M. Stephenson, C. DeTar, T. DeGrand, and A. Hasenfratz, "Scaling and eigenmode tests of the improved fat clover action," *Phys. Rev.*, vol. D63, p. 034501, 2001, hep-lat/9910023.

[81] A. Hasenfratz and F. Knechtli, "Flavor symmetry and the static potential with hypercubic blocking," *Phys. Rev.*, vol. D64, p. 034504, 2001, hep-lat/0103029.

[82] R. Frezzotti and K. Jansen, "The PHMC algorithm for simulations of dynamical fermions. i: Description and properties," *Nucl. Phys.*, vol. B555, pp. 395–431, 1999, hep-lat/9808011.

[83] A. D. Kennedy, I. Horvath, and S. Sint, "A new exact method for dynamical fermion computations with non-local actions," *Nucl. Phys. Proc. Suppl.*, vol. 73, pp. 834–836, 1999, hep-lat/9809092.

[84] S. Aoki *et al.*, "Non-perturbative determination of quark masses in quenched lattice QCD with the Kogut-Susskind fermion action," *Phys. Rev. Lett.*, vol. 82, pp. 4392–4395, 1999, hep-lat/9901019.

[85] D. Becirevic, V. Gimenez, V. Lubicz, and G. Martinelli, "Light quark masses from lattice quark propagators at large momenta," *Phys. Rev.*, vol. D61, p. 114507, 2000, hep-lat/9909082.

[86] J. I. Skullerud and A. G. Williams, "Quark propagator in Landau gauge," *Phys. Rev.*, vol. D63, p. 054508, 2001, hep-lat/0007028.

[87] J. Skullerud, D. B. Leinweber, and A. G. Williams, "Nonperturbative improvement and tree-level correction of the quark propagator," *Phys. Rev.*, vol. D64, p. 074508, 2001, hep-lat/0102013.

[88] P. O. Bowman, U. M. Heller, and A. G. Williams, "Lattice quark propagator in Landau and Laplacian gauges," *Nucl. Phys. Proc. Suppl.*, vol. 106, pp. 820–822, 2002, hep-lat/0110081.

[89] T. Blum *et al.*, "Non-perturbative renormalisation of domain wall fermions: Quark bilinears," *Phys. Rev.*, vol. D66, p. 014504, 2002, hep-lat/0102005.

[90] F. D. R. Bonnet, P. O. Bowman, D. B. Leinweber, A. G. Williams, and J.-b. Zhang, "Overlap quark propagator in Landau gauge," *Phys. Rev.*, vol. D65, p. 114503, 2002, hep-lat/0202003.

[91] J. B. Zhang, F. D. R. Bonnet, P. O. Bowman, D. B. Leinweber, and A. G. Williams, "Towards the continuum limit of the overlap quark propagator in Landau gauge," 2002, hep-lat/0208037.

[92] R. G. Edwards, U. M. Heller, and R. Narayanan, "A study of chiral symmetry in quenched QCD using the overlap-Dirac operator," *Phys. Rev.*, vol. D59, p. 094510, 1999, hep-lat/9811030.

[93] F. D. R. Bonnet, P. O. Bowman, D. B. Leinweber, A. G. Williams, and D. G. Richards, "Discretisation errors in Landau gauge on the lattice," *Austral. J. Phys.*, vol. 52, pp. 939–948, 1999, hep-lat/9905006.

[94] A. Cucchieri and T. Mendes, "A multigrid implementation of the fourier acceleration method for Landau gauge fixing," *Phys. Rev.*, vol. D57, pp. 3822–3826, 1998, hep-lat/9711047.

[95] A. Frommer, B. Nockel, S. Gusken, T. Lippert, and K. Schilling, "Many masses on one stroke: Economic computation of quark propagators," *Int. J. Mod. Phys.*, vol. C6, pp. 627–638, 1995, hep-lat/9504020.

[96] F. D. R. Bonnet, P. O. Bowman, D. B. Leinweber, A. G. Williams, and J.-b. Zhang, "Overlap quark propagator in Landau gauge," *Phys. Rev.*, vol. D65, p. 114503, 2002, hep-lat/0202003.

[97] S. Aoki, "New phase structure for lattice QCD with Wilson fermions," *Phys. Rev.*, vol. D30, p. 2653, 1984.

[98] M. Golterman and Y. Shamir, "Localization in lattice QCD," *Phys. Rev.*, vol. D68, p. 074501, 2003, hep-lat/0306002.

[99] M. Golterman and Y. Shamir, "Localization in lattice QCD (with emphasis on practical implications)," 2003, hep-lat/0309027.