DEVELOPING A PARALLELISABLE EIGENSOLVER

A thesis submitted in partial fulfilment of the requirements
for the award of the degree of

MASTER OF SCIENCE

by

SREYAM SENGUPTA
13MS121

Under the supervision of
Dr. PUSHAN MAJUMDAR

AND

Dr. ANANDA DASGUPTA

to the

DEPARTMENT OF PHYSICAL SCIENCES

INDIAN INSTITUTE OF SCIENCE EDUCATION AND
RESEARCH KOLKATA

May 6, 2018

Declaration

I hereby declare that this thesis is my own work and to the best of my knowledge, it contains no materials previously published or written by any other person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at IISER Kolkata or any other educational institution, except where due acknowledgement is made in the thesis.

Sreyam Sengupta

[Student's signature]
SREYAM SENGUPTA

April 24, 2018
IISER Kolkata

Certificate

This is to certify that the work entitled “DEVELOPING A PARALLELISABLE EIGEN-SOLVER” submitted by Sreyam Sengupta to the Department of Physical Sciences, IISER Kolkata, for the award of degree of Integrated BS-MS in Physical Sciences is a bonafide record of research work carried out by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

[illegible]

Dr. Pushan Majumdar
Supervisor
Department of Theoretical Physics
IACS

[illegible]

Dr. Ananda Dasgupta
Supervisor
Department of Physical Sciences
IISER Kolkata

3

Acknowledgements

All views expressed in this section are purely my own and do not necessarily reflect those of anybody else.
I begin by acknowledging my guides. I thank Dr. Pushan Majumdar for introducing me to the subject of lattice gauge theory, and for teaching me to appreciate computational physics (and even computer science) as an interesting and rewarding field of study in its own right. I also thank him for always looking out for my best interests (even when I did not), for working hard to ensure that I always had a place to sit and work, for being patient and understanding when helping me work out problems and for providing me access to the Cray XC30.
I thank my other guide, Dr. Ananda Dasgupta, for being one of the greatest teachers I have had the pleasure of knowing (no matter how much he claims otherwise) and for always being there to listen to my problems and for encouraging his students to think deeply for themselves. Without the above two gentlemen, this report would never have been written.
I thank DST-INSPIRE for their funding and scholarship, which enabled me to study at IISER Kolkata for five years, and do a final-year MS project at IACS.
I thank these two institutions as well: IISER Kolkata for being a home for the last five years (undoubtedly the best years of my life so far) and for providing me with a lifetime's worth of memories which I will always cherish, and IACS for welcoming me and giving me the opportunity to work there as one of their own. I thank the non-teaching staff at both these institutions for the tireless work they put in to ensure life is as smooth as possible for the students.
I thank the Indian Lattice Gauge Theory Initiative for letting me use the Cray XC30 parallel supercomputer for my project.
I thank all my teachers, both inside and outside the classroom, who have taught me most of what I know. It is impossible to thank everybody personally, but I would like to name Dr. Siddhartha Lal for being the best (non-academic) mentor one could hope for, and for having the courage to stand up for his beliefs even in the face of an oppressive authority. I also thank Dr. Bhavtosh Bansal for being the coolest professor one could have and a very good friend too - a lot of my ideas about politics and society have been shaped by my conversations with him. The scientific establishment needs more such people to provide a voice of dissent, reason and moderation. Finally, I thank Dr. Narayan Banerjee, for not only being a brilliant teacher but also for teaching his students (through his actions, never his words) of the importance of patience, kindness, compassion, humility and love.
I thank my friends and all those I love, for being accepting and tolerant towards me, for embracing me as one of their own, for teaching me of the importance of creativity and imagination and for helping me get through some difficult times.
I thank my family, for always being there and for making sure I never wanted for anything.
Finally, I thank my parents for everything. Some debts can never be repaid.

Abstract

The goal of this project was to develop a parallelisable algorithm to calculate the low-lying eigenspectrum of a large, sparse, positive-definite Hermitian matrix. The parallelised eigensolver will then be applied to some problems in lattice QCD. This report offers a very brief introduction to QCD as an $\mathbf{SU}(3)$ gauge theory, before motivating and introducing lattice QCD. The introduction is followed by a section describing a method used to resolve the fermion doubling problem on the lattice, and two subsequent sections on eigensolver algorithms. The first is the Lanczos method, the traditional algorithm which nevertheless suffers from a few flaws. The Kalkreuter-Simma (KS) algorithm is then introduced as a more viable alternative. The KS algorithm is alternated with Jacobi diagonalisations to further refine the eigenspectrum estimates. A program that implements the above algorithm is written and parallelised for the GPU using the OpenACC paradigm. Timing studies are performed on the program with two pre-generated lattices of different sizes as input. The results obtained so far are outlined, along with a brief discussion of further prospects.

Contents

List of Figures

List of Tables

Notation

In section (1.1), we have worked in $1+3$ dimensional Minkowski spacetime, with a metric given by $(ds)^2 = (dt)^2 - (dx)^2 - (dy)^2 - (dz)^2$ in the $(+,-,-,-)$ sign convention. From section (1.2) onwards, we switch to Euclidean spacetime, with a metric given by $(ds)^2 = (d\tau)^2 + (dx)^2 + (dy)^2 + (dz)^2$ in 4 dimensions.
Unless specified otherwise, we will be working in natural units, where $c = \hbar = 1$. When referring to spacetime, n dimensional will be understood to mean 1 timelike dimension and $n-1$ spacelike dimensions.

Algorithms in **pseudocode** form are given inside the `algorithmic` environment:

```
1:  procedure procedure name(variables)
2:      statements                                          ▷ comments
3:      if conditional statement then
4:          statements
5:      end if
6:      for loop control statement do
7:          statements
8:      end for
9:      while loop control statement do
10:         statements
11:     end while
12: end procedure
```

Cross-reference links to sections, equations, figures and tables in-document are enclosed inside parantheses, (*cross-reference*). Citation links to items in the bibliography are enclosed inside square brackets, [*citation*].

1 Introduction

1.1 Introduction to QCD

The *strong interaction* is one of the four fundamental interactions found in nature, the others being the weak interaction, electromagnetism (EM) and gravitation. Of these, the strong interaction is by far the strongest. Quantifying the exact strength of the four interactions depends on the particles and energies involved. Nevertheless, roughly speaking, at a length scale of 10^{-15} m, the strong interaction is approximately 137 times as strong as electromagnetism, a million times as strong as the weak interaction, and 10^{38} times as strong as gravitation [1].
The strength of
Quantum chromodynamics (QCD) is a non-abelian gauge theory with gauge group SU(3). It contains both matter and gauge fields. The matter fields describe quarks, spin-$\frac{1}{2}$ fermions that transform according to the fundamental representation of SU(3). The gauge fields describe gluons, spin-1 vector bosons that lie in the adjoint representation of SU(3). A fundamental property of both quarks and gluons is *colour*, a quantum mechanical property somewhat analogous to electric charge. Unlike electric charge, however, colour can be of three types: red, blue and green (and the respective negatives or complements of these, called antired, antiblue and antigreen). QCD is the quantum field theory (QFT) that describes coloured particles and their interactions. This is the strong interaction described above, and it is responsible for the stability of the atomic nucleus, and therefore the stability of all matter.
From [2], the Lie algebra $\mathbf{su}(n)$ associated with a Lie group $\mathbf{SU}(n)$ is a real algebra of dimension n^2-1. The elements of this Lie algebra can be taken to be $n \times n$ traceless Hermitian matrices, and the generators T_a of this algebra satisfy the following property:

$$T_a T_b = \frac{1}{2n}\delta_{ab}\mathbf{1}_n + \frac{1}{2}\sum_{c=1}^{n^2-1}(if_{abc}+d_{abc})T_c \tag{1}$$

where $\mathbf{1}_n$ is the $n \times n$ identity matrix, δ is the Kronecker delta, the f are structure constants and are antisymmetric in any 2 indices, while the d are symmetric in any 2 indices. The elements of the algebra are obtained by linear combinations of the generators of the algebra, while the elements of the identity-connected part of the corresponding group are obtained by exponentiating elements of the algebra.
QCD matter fields transform under the set of local transformations described by

$$\begin{aligned}\psi(x) &\to U(x)\psi(x)\\ \overline{\psi}(x) &\to \overline{\psi}(x)(U(x))^{\dagger}\end{aligned} \tag{2}$$

where the ψ and $\overline{\psi}$ are a spinor and its adjoint respectively. They represent a fermion and its corresponding antifermion ($\overline{\psi} = \psi^{\dagger}\gamma_0$), while the U matrices are members of the gauge group $\mathbf{SU}(3)$ and in general depend on the spacetime coordinates. With spacetime dependent transformations, one needs to replace the partial derivative by the covariant derivative as follows:

$$\partial_\mu \to D_\mu \equiv (\partial_\mu - igA_\mu) \tag{3}$$

where A is an element of the $\mathbf{su}(3)$ algebra, and g is the coupling that determines the strength of the interaction.
Under the transformation described in (2), the A transform as:

$$A_\mu \to U(x)A_\mu(U(x))^{\dagger} - \frac{i}{g}(\partial_\mu U(x))(U(x))^{\dagger} \tag{4}$$

which is exactly the transformation needed to ensure the covariant derivative defined in (3) transforms similarly to the matter fields:

$$D_\mu \psi(x) \to U(x) D_\mu \psi(x). \tag{5}$$

The QCD Lagrangian is given by

$$\mathscr{L} = \overline{\psi} i \not{D}(A)\psi - m_f \overline{\psi}\psi - \frac{1}{4g^2}\mathrm{Tr}(F^{\mu\nu}F_{\mu\nu}) \tag{6}$$

where ψ and $\overline{\psi}$ represent the matter part of the Lagrangian (a fermion and its corresponding antifermion), m_f is the mass of the fermion, $\not{D} = \gamma^{\mu} D_\mu$ (γ are the Dirac matrices) and $F_{\mu\nu} = \partial_\mu A_\nu - \partial_\nu A_\mu + i[A_\mu, A_\nu]$ represents the gauge part of the Lagrangian. The interaction between the matter fields and the gauge fields is contained in the covariant derivative $\not{D}$. One can check that this Lagrangian is invariant under the gauge transformations of the matter and gauge fields described above.
In 4 spacetime dimensions, the action S is defined as

$$S = \int d^4x \mathscr{L}. \tag{7}$$

Given an action, one may define a partition function as follows:

$$Z = \int \mathscr{D}A \mathscr{D}\overline{\psi} \mathscr{D}\psi \, \mathrm{e}^{iS}. \tag{8}$$

One can use the partition function to calculate the expectation value of a physical observable $\mathscr{O}$ using the following relation:

$$\langle \mathscr{O} \rangle = \frac{1}{Z} \int \mathscr{D}A \mathscr{D}\overline{\psi} \mathscr{D}\psi \mathscr{O} \mathrm{e}^{iS} \tag{9}$$

From (6) and (7) it is evident that the action S is quartic in A. It is not known how to solve such functional integrals analytically. One therefore has two alternative ways to proceed. The first is to perturbatively expand the quantity e^{iS} as a series in g. This option is limited by the fact that the series only makes sense for small values of g, so this perturbative technique cannot deal with the strong coupling regime.
The second alternative is to estimate functional integrals like (9) numerically. This technique is fully non-perturbative, but requires one to discretise spacetime. A formulation of QCD in discrete spacetime is known as lattice QCD, which will be introduced briefly in the following chapter.

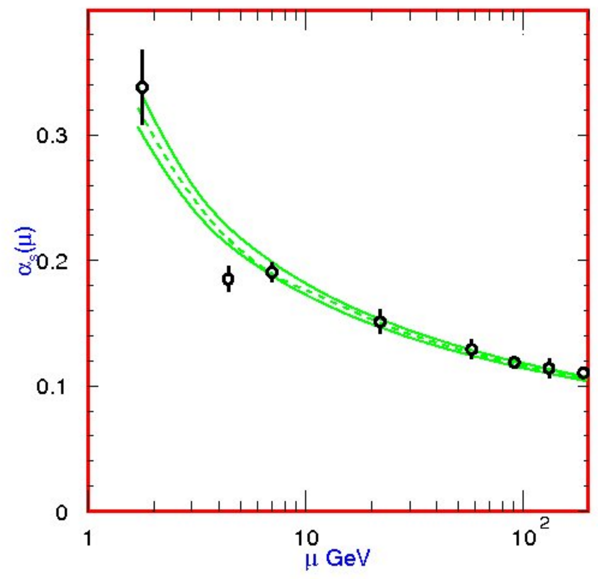

Figure 1: $\alpha_s(\mu)$ vs μ. Lines: central value $\pm 1\sigma$ limits. Data points (in increasing order of μ): τ width, γ decays, deep inelastic scattering, e^+e^- event shapes at 22 GeV from the JADE data, shapes at TRISTAN at 58 GeV, Z width, and e^+e^- event shapes at 135 and 189 GeV.

The value of g depends on the energy at which one probes the system. In figure (1), a physical observable $\alpha_s \equiv g^2/4\pi$ is plotted against μ, the energy scale at which the system is being probed. It is clear that $\alpha_s(\mu)$ decreases with increasing μ. This implies that the coupling g is not a constant, but varies with the energy level at which the system is being probed. The

proton mass is approximately around $\mu \sim 1$. From figure (1), it is evident that the strength of the coupling g is not very small at that energy scale, so perturbative techniques are not very useful for calculating properties of particles such as the proton.
From experiment, it is known that any theory of the strong interaction should have the following properties:

1. It must have a *mass gap*, i.e. the theory should have a constant $\Delta > 0$ so that every excitation of the vacuum has energy at least Δ. This property explains why the nuclear force is strong but short-ranged.

2. It must have *quark confinement* - even though the theory is described in terms of elementary quark fields that transform non-trivially under $\mathbf{SU}(3)$ transformations, the physically observable particle states (such as the neutron, proton and pion) are invariant under $\mathbf{SU}(3)$. Alternatively, one may say that coloured particle states are not physically observable. This property explains the fact that individual quarks (or any coloured combination thereof) are not physically observable.

3. The theory must have *chiral symmetry breaking*, which means that the vacuum is potentially invariant (in the limit where the bare quark masses vanish) only under a certain subgroup of the full symmetry group that acts on the quark fields. This property is needed to account for the low mass of the pion and the absence of parity partners in the low-lying spectrum of hadrons.

None of these properties can currently be obtained or explained analytically usng perturbative QCD. So, lattice QCD is the only way to perform the calculations and simulations necessary to obtain and explain these properties of the strong interaction.

1.2 Introduction to lattice QCD

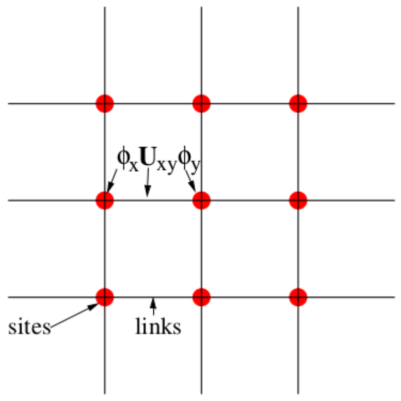

Figure 2: A 2 dimensional lattice. Note that the matter fields are placed on the vertices of the lattice (called *sites*) while the gauge fields are placed on the edges (called *links*).

In this, and all following sections, we shall work in Euclidean spacetime, given by the transformation $t \mapsto \tau \equiv it$.

Lattice QCD is a formulation of QCD that uses a discretised version of spacetime. The objectives of lattice QCD are twofold: to computationally verify that the results obtained in strong interaction experiments can also be obtained starting from the QCD lagrangian, and to develop new non-perturbative techniques to study the physics at strong coupling.

For the purposes of lattice QCD, spacetime is approximated by a 4 dimensional hypercubic lattice of finite lattice spacing and finite extent. The *lattice spacing* is defined as the distance between two nearest neighbour points and is denoted by a. The *lattice extent* in a given direction is the number of lattice points along that direction multiplied by the lattice spacing.

Matter and gauge fields can be defined on the lattice. Matter fields are defined on the lattice vertices, called *sites*. For example, in figure (2) above, two matter fields ϕ_x and ϕ_y can be seen on neighbouring sites on the lattice x and y respectively. Between them is the gauge field U_{xy} defined on the *link* xy. Henceforth gauge fields will be represented by $\mathscr{U}$; the notation U will be reserved for members of the group $\mathbf{SU}(3)$ that represent the gauge transformations.

Quantities defined on the continuum need to be redefined on the lattice. For example, the derivative operator becomes the finite difference operator:

$$\partial_\mu \phi \to \phi(x_\mu + a_\mu) - \phi(x_\mu) \tag{10}$$

where a_μ is the lattice spacing in the μ direction. The infinitesimal parallel transport (or covariant derivative) is replaced by its finite version:

$$D_\mu \phi \to \phi(x_\mu + a_\mu) - \mathscr{U}(x_\mu, x_\mu + a_\mu)\phi(x_\mu) \tag{11}$$

where $\mathscr{U}$ is the gauge field on the link between $\phi(x_\mu + a_\mu)$ and $\phi(x_\mu)$. $\mathscr{U} \sim \exp(iaA)$, where A is an element of the **su**(3) algebra. Finally, the gauge transformations themselves become

$$\begin{aligned} \phi'(x) &= U(x)\phi(x) \\ \mathscr{U}'(x_\mu, x + a_\mu) &= (U(x_\mu))^{-1}\mathscr{U}(x_\mu, x_\mu + a_\mu)U(x_\mu + a_\mu) \end{aligned} \tag{12}$$

where U are the **SU**(3) matrices representing the gauge transformation.
The smallest loops on the lattice are called *plaquettes*. As given in the figure below, a plaquette is a square of side length a (i.e. the lattice spacing) which has 4 sites as its vertices and the gauge links connecting those sites as its edges.

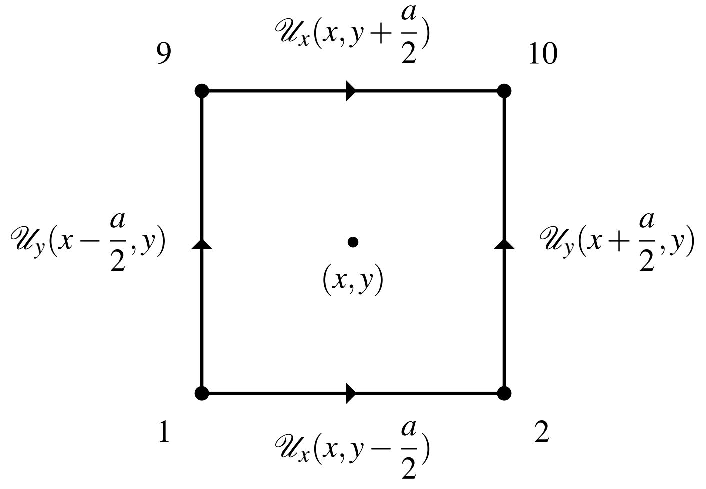
9
$\mathscr{U}_x(x, y + \frac{a}{2})$
10
$\mathscr{U}_y(x - \frac{a}{2}, y)$
(x, y)
$\mathscr{U}_y(x + \frac{a}{2}, y)$
1
$\mathscr{U}_x(x, y - \frac{a}{2})$
2

Figure 3: A plaquette p of a lattice with lattice spacing a and $N_x = 8$. Note that the gauge fields always point from a node with lower site index to one with higher site index.

The *plaquette product* (denoted by $\Box$) is defined as the ordered, cyclic product of the 4 gauge fields that make up the sides of the plaquette. It is an ordered product because one starts from a given edge and takes the product going either clockwise or anti-clockwise around the square, and it is a cyclic product because the gauge fields need to be pointing along the direction that one goes around the square. However, as evident from the figure, in the plaquette, gauge fields always point from a node of lower site index to one of higher site index (by convention). So, one has to conjugate any two of the four gauge fields to obtain the plaquette product. For instance, in the plaquette p shown in the figure above, start from the lower edge, i.e. $\mathscr{U}_x(x, y -$

$a/2)$ and compose the product $\Box(p)$ by going anti-clockwise around the square, conjugating gauge links where their direction opposes the direction of the anti-clockwise cycle. So, the final product is:

$$\Box(p) = \mathscr{U}_x(x,y-\frac{a}{2}) \times \mathscr{U}_y(x+\frac{a}{2},y) \times \mathscr{U}_x^\dagger(x,y+\frac{a}{2}) \times \mathscr{U}_y^\dagger(x-\frac{a}{2},y) \tag{13}$$

As seen above,the gauge field $\mathscr{U}_x(x,y-a/2)$ can be written as

$$\mathscr{U}_x(x,y-\frac{a}{2}) = \exp(iaA_x(x,y-\frac{a}{2})) \tag{14}$$

where all symbols have their usual meaning. On the lattice, one may define the gauge part of the Wilson action as:

$$S_W = \frac{2}{3g^2}\sum_p \Re(\mathrm{Tr}(\mathbf{1}-\Box(p))) \tag{15}$$

where $\Re(z)$ means the real part of z, $\mathbf{1}$ is the relevant identity matrix and the summation is done over all plaquettes. The coupling g will henceforth be suppressed.
It can be shown (and will be briefly demonstrated below) that the Wilson action S_W is equal to the quantity $-\frac{1}{4}\sum_p \mathrm{Tr}(F_{\mu\nu}F^{\mu\nu})$ (i.e. the standard Yang-Mills action in the continuum) to $\mathscr{O}(a^4)$. To show this, one first needs to write the gauge fields as the exponential of the algebra elements, so that the plaquette product becomes

$$\begin{aligned}&\mathscr{U}_x(x,y-\frac{a}{2})\mathscr{U}_y(x+\frac{a}{2},y)\mathscr{U}_x^\dagger(x,y+\frac{a}{2})\mathscr{U}_y^\dagger(x-\frac{a}{2},y)\\ =&\exp(iaA_x(x,y-\frac{a}{2}))\exp(iaA_y(x+\frac{a}{2},y))\exp(-iaA_x(x,y+\frac{a}{2}))\exp(-iaA_y(x-\frac{a}{2},y))\end{aligned} \tag{16}$$

The next step is to expand the exponentials to $\mathscr{O}(a^4)$, so that, for example, $\exp(iaA_x(x,y-\frac{a}{2}))$ is written as $1+iaA_x(x,y-\frac{a}{2})-\frac{a^2}{2!}(A_x(x,y-\frac{a}{2}))^2-\frac{ia^3}{3!}(A_x(x,y-\frac{a}{2}))^3+\frac{a^4}{4!}(A_x(x,y-\frac{a}{2}))^4$. Now, a Taylor series expansion is performed on the A, so in the example above, $A_x(x,y-\frac{a}{2})$ is expanded about the point (x,y) in each of the terms in the exponential so that there is no term in the final expression that is of higher order than $\mathscr{O}(a^4)$. The resulting expression is simplified and the final expression is given below in orders of a:
$\mathscr{O}(1)$: 1 (the term independent of a in each of the exponentials is 1, so their product is 1).
$\mathscr{O}(a)$: 0.
$\mathscr{O}(a^2)$: iF_{xy}, which is a single component of the gauge field tensor, and can therefore be written as $iF_{xy}^aT^a$, where the T^a's are the generators of the Lie algebra $\mathbf{su}(3)$ and are therefore traceless. Hence, when the trace is taken, this quantity vanishes.
$\mathscr{O}(a^3)$: $[(iA_x - iA_y),(\frac{i}{2}\partial_x A_y - \frac{i}{2}\partial_y A_x - \frac{1}{2}A_x^2 - \frac{1}{2}A_y^2 + A_yA_x)]$, which is an exact commutator. Since commutators are traceless, when the trace is taken over the entire expression, this term also vanishes.

$\mathcal{O}(a^4)$: $\frac{1}{2}(\partial_x A_y - \partial_y A_x + iA_xA_y - iA_yA_x)^2 + \frac{1}{4}[\partial_x A_y, \partial_y A_x] + \frac{1}{2}[A_y, A_xA_yA_x]$, of which only the first term survives after the trace is taken. The first term is precisely $\frac{1}{4}F_{\mu\nu}F^{\mu\nu}$ in 2 dimensions.
So it is proved that

$$\Box(p) = \mathbf{1} + \frac{1}{4}F_{\mu\nu}F^{\mu\nu} + \text{commutators} \tag{17}$$

from which it follows that the Wilson action S_W is equivalent to $-\sum_p \frac{1}{4}\operatorname{Tr}(F_{\mu\nu}F^{\mu\nu})$, the continuum Yang-Mills action, to $\mathcal{O}(a^4)$.
Similar to the gauge part of the action, a naive fermion action may be written as [3]:

$$S_F[\psi, \overline{\psi}, \mathscr{U}] = a^4 \sum_{n\in\Lambda} \overline{\psi}(n) \left(\sum_{\mu=1}^{4} \gamma_\mu \frac{\mathscr{U}_\mu(n)\psi(n+\hat{\mu}) - \mathscr{U}_{-\mu}(n)\psi(n-\hat{\mu})}{2a} + m\psi(n) \right) \tag{18}$$

where Λ is the set of all lattice sites, n denotes a particular lattice site, $\hat{\mu}$ is a unit vector in the direction of μ, $\mathscr{U}_\mu(n)$ is the gauge field originating at site n in the positive μ direction, and m is the mass of the particular fermion. The notation used is that of [3]. Now, since the action is bilinear in $\overline{\psi}$ and ψ, it can be written as

$$S_F[\psi, \overline{\psi}, \mathscr{U}] = a^4 \sum_{n,m\in\Lambda} \sum_{a,b,\alpha,\beta} \overline{\psi}(n)_{\alpha,a} D(n|m)_{\alpha\beta,ab} \psi(m)_{\beta,b} \tag{19}$$

where α and β are Dirac indices while a and b are colour indices. All other symbols have their usual meaning. The naive Dirac operator in (19) can be written as:

$$D(n|m)_{\alpha\beta,ab} = \sum_{\mu=1}^{4} (\gamma_\mu)_{\alpha\beta} \frac{\mathscr{U}_\mu(n)_{ab}\delta_{n+\hat{\mu},m} - \mathscr{U}_{-\mu}(n)_{ab}\delta_{n-\hat{\mu},m}}{2a} + m\delta_{\alpha\beta}\delta_{ab}\delta_{n,m}. \tag{20}$$

Assuming the Fourier transform is defined on the lattice, one can compute the Fourier transform of the lattice Dirac operator $D(n|m)$ for the special case of $\mathscr{U}_\mu(n) = \mathbf{1}$, i.e. for free lattice fermions. The Fourier transform is:

$$\begin{aligned} \tilde{D}(p|q) &= \frac{1}{|\Lambda|} \sum_{n,m\in\Lambda} \mathrm{e}^{-ip\cdot na} D(n|m)\, \mathrm{e}^{iq\cdot ma} \\ &= \delta_{p,q}\tilde{D}(p) \end{aligned} \tag{21}$$

where $|\Lambda|$ is the cardinality of the set Λ, i.e. the total number of lattice points, and Fourier transform of the lattice Dirac operator is given by

$$\tilde{D}(p) = m\mathbf{1} + \frac{i}{a} \sum_{\mu=1}^{4} \gamma_\mu \sin(p_\mu a). \tag{22}$$

Using

$$\left(a\mathbf{1} + i\sum_{\mu=1}^{4}\gamma_\mu b_\mu\right)^{-1} = \frac{a\mathbf{1} - i\sum_{\mu=1}^{4}\gamma_\mu b_\mu}{a^2 + \sum_{\mu=1}^{4} b_\mu^2} \tag{23}$$

one obtains:

$$\tilde{D}(p)^{-1} = \frac{m\mathbf{1} - ia^{-1}\sum_\mu \gamma_\mu \sin(p_\mu a)}{m^2 + a^{-2}\sum_\mu \sin(p_\mu a)^2}. \tag{24}$$

This is the inverse of the lattice Dirac operator for free fermions, also called the *quark propagator*.

In the case of massless fermions ($m = 0$), for fixed p the propagator has the following continuum limit:

$$\tilde{D}(p)^{-1}\Big|_{m=0} = \frac{-ia^{-1}\sum_\mu \gamma_\mu \sin(p_\mu a)}{a^{-2}\sum_\mu \sin(p_\mu a)^2} \xrightarrow{a\to 0} \frac{-i\sum_\mu \gamma_\mu p_\mu}{p^2}. \tag{25}$$

Thus the naive Dirac operator yields the correct propagator in the continuum limit. However, identifying a particle from the pole of a propagator, we see that in the continuum limit ($a \to 0$) the momentum space propagator for massless fermions has a pole at $(0,0,0,0)$. This pole corresponds to the single fermion described by the continuum Dirac operator. On the lattice, however, because the propagator has $\sin(p_\mu a)^2$ in the denominator, whenever all components p_μ are either 0 or $\pi/2$, there is a pole. Since the momentum space contains all momenta p_μ in the range $(-\pi/a, \pi/a]$ with $-\pi/a$ and π/a identified, one cannot exclude the value $p_\mu = \pi/a$. So, in addition to $(0,0,0,0)$, the propagator also has poles at 15 other points. This is the *fermion doubling problem*, and these 15 extra poles are called *doublers*.

2 Staggered operator and eigenvalue algorithms

2.1 Staggered operator

The previous section introduced the fermion doubling problem, where in addition to the momentum pole at $p = (0,0,0,0)$, 15 other unphysical poles were obtained at other points in momentum space as a result of the zeros of the $\sin(p_\mu a)^2$ term in the denominator of the quark propagator on the lattice. There exist several formulations that attempt to overcome this problem, and the one we use will be briefly discussed below. For brevity, Dirac and colour indices have been suppressed in subsequent calculations.

Staggered or *Kogut-Susskind* fermions [4] are a formulation that reduces the 16-fold degeneracy of the naive discretisation outlined above to 4 quarks. The staggered transformation does this by mixing the spinor and spacetime indices, which distributes the quark degrees of freedom on the hypercubes of the lattice.

From (18), setting the gauge fields $\mathscr{U} = \mathbf{1}$, i.e. for free fermions, the action obtained from the naive discretisation is

$$S_F[\psi,\overline{\psi}] = a^4 \sum_{n\in\Lambda} \overline{\psi}(n) \left(\sum_{\mu=1}^{4} \gamma_\mu \frac{\psi(n+\hat{\mu}) - \psi(n-\hat{\mu})}{2a} + m\psi(n) \right) \tag{26}$$

where the notation of [3] has been used. One can then define a *staggered transformation* on the fermionic variables as follows. A lattice site in 4 dimensions may be specified by a 4-tuple of coordinates $n = (n_1, n_2, n_3, n_4)$. Then one effects a transformation on the fermionic fields by defining new variables $\psi(n)'$ and $\overline{\psi}(n)'$ as:

$$\begin{aligned} \psi(n) &= \gamma_1^{n_1}\gamma_2^{n_2}\gamma_3^{n_3}\gamma_4^{n_4}\psi(n)' \\ \overline{\psi}(n) &= \overline{\psi}(n)'\gamma_4^{n_4}\gamma_3^{n_3}\gamma_2^{n_2}\gamma_1^{n_1} \end{aligned} \tag{27}$$

where n_1, n_2, n_3 and n_4 are simply the components of the site n. Since the gamma matrices obey the relation $\gamma_\mu^2 = \mathbf{1}$, it is easy to see from (27) that $\overline{\psi}(n)\psi(n) = \overline{\psi}(n)'\psi(n)'$, so the mass term in (26) remains invariant. The kinetic term, which is the term where $\overline{\psi}(n)$ multiplies with the quantity $(\psi(n+\hat{\mu}) - \psi(n-\hat{\mu}))/2a$, undergoes a change because the γ-matrix coefficients of $\psi(n+\hat{\mu})$ are different from those of $\psi(n-\hat{\mu})$ because they are defined at two different lattice sites, and the transformation (27) is site-dependent through the exponents on the γ-matrices. After some calculation, it is seen that the action defined in (26) has changed to:

$$S_F[\psi',\overline{\psi}'] = a^4 \sum_{n\in\Lambda} \overline{\psi}(n)'\mathbf{1} \left(\sum_{\mu=1}^{4} \eta_\mu(x) \frac{\psi(n+\hat{\mu})' - \psi(n-\hat{\mu})'}{2a} + m\psi(n)' \right) \tag{28}$$

where η are the staggered sign functions, defined as

$$\begin{aligned} \eta_1(n) &= 1 \\ \eta_2(n) &= (-1)^{n_1} \\ \eta_3(n) &= (-1)^{n_1+n_2} \\ \eta_4(n) &= (-1)^{n_1+n_2+n_3} \end{aligned} \tag{29}$$

The η functions (which are scalars) have taken the place of the γ-matrices of (26). Now, the action (28) has the same form for all four Dirac components (the Dirac indices were suppressed in the expressions above) so one may easily discard three and keep one of the identical components. Upon putting back the gauge fields, one finds the form of the action has changed to the *staggered* action, as defined below:

$$S_F[\chi,\overline{\chi}] = a^4 \sum_{n\in\Lambda} \overline{\chi}(n) \left(\sum_{\mu=1}^{4} \eta_\mu(x) \frac{\mathscr{U}_\mu(n)\chi(n+\hat{\mu}) - \mathscr{U}_\mu^\dagger(n-\hat{\mu})\chi(n-\hat{\mu})}{2a} + m\chi(n) \right) \tag{30}$$

where $\bar{\chi}$ and χ are Grassmann fields with colour indices but without Dirac indices.
The chirality matrix $\gamma_5 \equiv \gamma_1\gamma_2\gamma_3\gamma_4$ in Euclidean spacetime is replaced by

$$\eta_5(n) = (-1)^{n_1+n_2+n_3+n_4} \tag{31}$$

according to the relation

$$\overline{\psi}\gamma_5\psi = \eta_5\overline{\psi}'\mathbf{1}\psi'. \tag{32}$$

For the case $m = 0$, the staggered action (30) is invariant under the following transformation, known as *residual chiral symmetry*:

$$\begin{aligned} \chi(n) &\to \mathrm{e}^{i\alpha\eta_5(n)}\chi(n) \\ \bar{\chi}(n) &\to \bar{\chi}(n)\,\mathrm{e}^{i\alpha\eta_5(n)}. \end{aligned} \tag{33}$$

The staggered Dirac operator is defined as

$$D^{st}(n|m) = m\delta_{n,m} + \sum_{\mu=1}^{4} \eta_\mu(n)\frac{\mathscr{U}_\mu(n)\delta_{n+\hat{\mu},m} - \mathscr{U}_\mu^\dagger(n-\hat{\mu})\delta_{n-\hat{\mu},m}}{2a} \tag{34}$$

and for a massless fermion, the above operator is anti-Hermitian and obeys the following relation:

$$(D^{st})^\dagger(n|m) = -D^{st}(n|m) = \eta_5(n)D^{st}(n|m)\eta_5(m). \tag{35}$$

This means the staggered Dirac operator is anti-Hermitian, which in turn means its eigenvalues are purely imaginary. So, the eigenvalues of a quantity such as $D^\dagger D$ are guaranteed to be strictly non-negative. If the quarks are massive, the eigenvalues are strictly positive. In the following sections, algorithms used to find the low-lying eigenspectrum of operators such as $D^\dagger D$ will be briefly discussed.

2.2 Lanczos algorithm

In all mathematical expressions in this subsection and the one after, when referring to computational algorithms, we will denote vectors by lowercase boldface letters such as $\boldsymbol{v}$ and matrices with uppercase boldface letters, e.g. $\boldsymbol{M}$. Scalars will be denoted by lowercase regular font, e.g s. When discussing physical quantities defined either in the continuum or on the lattice, we will use the notation used previously.
As mentioned in the previous subsection, the goal is to find the low-lying eigenspectrum of the operator $D^\dagger D$, where D is the staggered lattice Dirac operator as defined in (34). There are several such algorithms that find a specific set of eigenvalues and corresponding eigenvectors of a given matrix. The method of Lanczos iterations is one such method, and will be briefly

touched upon below.

The goal of the Lanczos method, as given in [5], is to estimate the lowest and highest eigenvalues and corresponding eigenvectors of a real, sparse, symmetric matrix $\boldsymbol{A}$ of size n. A sparse matrix is one in which most of the elements are 0. Let the largest eigenvalue of $\boldsymbol{A}$ be denoted by λ_1 and the smallest eigenvalue by λ_n. It can be shown that, for any $\boldsymbol{y} \in \mathbb{R}^n \ni \boldsymbol{y} \neq 0$,

$$\begin{aligned} \lambda_1 &= \max_{y\neq 0} \frac{\boldsymbol{y}^\top \boldsymbol{A}\boldsymbol{y}}{\boldsymbol{y}^\top \boldsymbol{y}} \\ \lambda_n &= \min_{y\neq 0} \frac{\boldsymbol{y}^\top \boldsymbol{A}\boldsymbol{y}}{\boldsymbol{y}^\top \boldsymbol{y}} \end{aligned} \tag{36}$$

where the quantity $\frac{\boldsymbol{y}^\top \boldsymbol{A}\boldsymbol{y}}{\boldsymbol{y}^\top \boldsymbol{y}}$ is called the Rayleigh quotient of $\boldsymbol{y}$ with respect to $\boldsymbol{A}$ and can be shown [5] to approximate the eigenvalue of $\boldsymbol{A}$ corresponding to $\boldsymbol{y}$ if $\boldsymbol{y}$ were an eigenvector of $\boldsymbol{A}$.

The Lanczos algorithm solves the problem of finding the greatest and least eigenvalue and corresponding eigenvectors of $\boldsymbol{A}$ by constructing a tri-diagonal matrix $\boldsymbol{T}$ from $\boldsymbol{A}$ by the transformation $\boldsymbol{T} = \boldsymbol{Q}^\top \boldsymbol{A}\boldsymbol{Q}$, where $\boldsymbol{Q}$ is an orthogonal matrix. The fact that such a $\boldsymbol{Q}$ always exists is proven by the fact that a real, symmetric matrix $\boldsymbol{A}$ can always be diagonalised by an orthonormal matrix $\boldsymbol{U}$ whose columns are the eigenvectors of $\boldsymbol{A}$ by the transformation $\Lambda = \boldsymbol{U}^\top \boldsymbol{A}\boldsymbol{U}$, where Λ is the diagonal matrix whose elements are the eigenvalues of $\boldsymbol{A}$. Since a diagonal matrix is necessarily tri-diagonal, it follows that there exists a $\boldsymbol{Q}$ that can tri-diagonalise $\boldsymbol{A}$.

The greatest and least eigenvalues of $\boldsymbol{T}$ provide estimates for the greatest and least eigenvalues of $\boldsymbol{A}$ respectively. Also, the Lanczos algorithm converges after n steps, where n is the size of $\boldsymbol{A}$. The algorithm also uses to its advantage the fact that $\boldsymbol{A}$ is sparse. How it does so is breifly discussed below.

One requires a matrix

$$\boldsymbol{T} = \boldsymbol{Q}^\top \boldsymbol{A}\boldsymbol{Q} \tag{37}$$

where $\boldsymbol{T}$ is tri-diagonal:

$$\boldsymbol{T} = \begin{bmatrix} \alpha_1 & \beta_1 & & & & 0 \\ \beta_1 & \alpha_2 & \beta_2 & & & \\ & \beta_2 & \alpha_3 & \beta_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & \beta_{n-2} & \alpha_{n-1} & \beta_{n-1} \\ 0 & & & & \beta_{n-1} & \alpha_n \end{bmatrix} \tag{38}$$

The matrix $\boldsymbol{Q}$ may be expressed as a row $[\boldsymbol{q}_1\boldsymbol{q}_2 \ldots \boldsymbol{q}_n]$ where each $\boldsymbol{q}_i$ is a column. $\boldsymbol{Q}$ may be taken to be orthonormal [5]. In that case, solving the equation (37) is equivalent to solving the equation $\boldsymbol{AQ} = \boldsymbol{QT}$. The k-th column of $\boldsymbol{AQ}$ gives the following recursion relation:

$$\begin{aligned} \boldsymbol{A}\boldsymbol{q}_k &= \boldsymbol{Q}\boldsymbol{T}_k \\ &= \beta_{k-1}\boldsymbol{q}_{k-1} + \alpha_k \boldsymbol{q}_k + \beta_k \boldsymbol{q}_{k+1} \end{aligned} \tag{39}$$

The first step is to set $\beta_0 \boldsymbol{q}_0 = 0$. Since the $\boldsymbol{q}_i$ are orthonormal, multiplying (39) from the left by $\boldsymbol{q}_k^\top$ yields

$$\boldsymbol{q}_k^\top \boldsymbol{A}\boldsymbol{q}_k = \alpha_k ||\boldsymbol{q}_k||^2 = \alpha_k \tag{40}$$

which is the equation for α_k. Given α_k, one may solve for $\beta_k \boldsymbol{q}_{k+1}$:

$$\beta_k \boldsymbol{q}_{k+1} = (\boldsymbol{A} - \alpha_k \mathbf{1})\boldsymbol{q}_k - \beta_{k-1}\boldsymbol{q}_{k-1} \tag{41}$$

The quantity $(\boldsymbol{A} - \alpha_k \mathbf{1})\boldsymbol{q}_k - \beta_{k-1}\boldsymbol{q}_{k-1}$ is called $\boldsymbol{r}_k$. If $\boldsymbol{r}_k \neq 0$,

$$\boldsymbol{q}_{k+1} = \frac{\boldsymbol{r}_k}{\beta_k} \tag{42}$$

and since $\boldsymbol{q}_{k+1}$ is normalised, $\beta_k = ||\boldsymbol{r}_k||$.
The algorithm for Lanczos iterations is given below:

```
1: procedure LANCZOS ITERATION($\boldsymbol{A}$)
2:     $\boldsymbol{r}_0 = \boldsymbol{q}_1$, $\beta_0 = 1$, $\boldsymbol{q}_0 = 0$, $k = 0$        ▷ Initialising the values of $\boldsymbol{r}$, $\beta$, $\boldsymbol{q}$ and $k$.
3:     while $\beta_k \neq 0$ do
4:         $\boldsymbol{q}_{k+1} = \frac{\boldsymbol{r}_k}{\beta_k}$
5:         $k = k + 1$
6:         $\alpha_k = \boldsymbol{q}_k^\top \boldsymbol{A}\boldsymbol{q}_k$
7:         $\boldsymbol{r}_k = (\boldsymbol{A} - \alpha_k \mathbf{1})\boldsymbol{q}_k - \beta_{k-1}\boldsymbol{q}_{k-1}$
8:         $\beta_k = ||\boldsymbol{r}_k||$
9:     end while
10: end procedure
```

Even though the Lanczos algorithm is a powerful and widely used algorithm, it suffers from two shortfalls [6]:

1. Lanczos does not supply the eigenvector estimates, merely the eigenvalue estimates through the extreme eigenvalues of the tri-diagonal matrix $\boldsymbol{T}$.

2. Lanczos provides no information about the multiplicities of the eigenvalues. This means it cannot detect degeneracies in the spectrum of $\boldsymbol{A}$. This is problematic in situations where one expects a degeneracy in the eigenvalue spectrum.

The next part deals with the Kalkreuter-Simma (KS) algorithm, which seeks to deal with these limitations and provide an accelerated algorithm based on the method of Conjugate Gradients (CG).

2.3 Kalkreuter-Simma algorithm

The KS algorithm was first proposed in 1995 by Thomas Kalkreuter and Hubert Simma. It takes a sparse, Hermitian, positive-(semi)definite matrix as input and computes a user-specified number of the lowest eigenvalues, starting from the lowest. In this report, the notation of [7] has been used, where a lot of the calculations hinted at in [6] have been worked out explicitly. The notation is briefly described below.
$\boldsymbol{A}$ is the $n \times n$ positive-(semi)definite Hermitian matrix. One requires to find the lowest n_E eigenvalues of $\boldsymbol{A}$ (where $n_E \leq n$ in principle and in practice $n_E << n$) along with their corresponding eigenvectors. Let $\boldsymbol{x}_{(i)}$ denote the i-th eigenvector estimate (here the eigenvalues arranged in ascending order of i), i.e. at the end of one round of KS iterations, $\boldsymbol{x}_{(0)}$ stores the eigenvector estimate corresponding to the lowest eigenvalue, while $\boldsymbol{x}_{(1)}$ stores the eigenvector estimate corresponding to the second-lowest eigenvalue, and so on. One therefore has n_E such vectors, from $\boldsymbol{x}_{(0)}$ to $\boldsymbol{x}_{(n_E-1)}$. Corresponding to each $\boldsymbol{x}_{(i)}$ there is a $\mu_{(i)}$ that stores the i-th eigenvalue estimate. So $\mu_{(0)}$ stores the lowest eigenvalue estimate, $\mu_{(1)}$ stores the second-lowest eigenvalue estimate and so on.
The KS algorithm is described below:

```
1: procedure KS
2:     for $i \in \{0, \dots, n_E - 1\}$ do
3:         $\boldsymbol{x}_{(i)} \leftarrow$ input    ▷ In the initial round of KS, a random input is used. For subsequent rounds the input comes from the result of the immediately previous Jacobi round.
4:         $\boldsymbol{x}_{(i)} \leftarrow \frac{\boldsymbol{x}_{(i)}}{||\boldsymbol{x}_{(i)}||}$    ▷ normalisation
5:         for $j \in \{0, \dots, i-1\}$ do
6:             $\boldsymbol{x}_{(i)} \leftarrow \boldsymbol{x}_{(i)} - \frac{\boldsymbol{x}_{(j)}^\dagger \boldsymbol{x}_{(i)}}{||\boldsymbol{x}_{(j)}||^2} \boldsymbol{x}_{(j)}$    ▷ modified Gram-Schmidt orthogonalisation
7:             $\boldsymbol{x}_{(i)} \leftarrow \frac{\boldsymbol{x}_{(i)}}{||\boldsymbol{x}_{(i)}||}$    ▷ normalisation
```

```
    end for
    $y_{(i)} \leftarrow Ax_{(i)}$    ▷ matrix-vector multiplication - once for every eigenvalue
    $\mu_{(i)} \leftarrow x_{(i)}^{\dagger} y_{(i)}$
    $g_{(i)} \leftarrow y_{(i)} - \mu_{(i)} x_{(i)}$
    for $j \in \{0, \dots, i-1\}$ do
        $g_{(i)} \leftarrow g_{(i)} - \frac{x_{(j)}^{\dagger} g_{(i)}}{||x_{(j)}||^2} x_{(j)}$    ▷ modified Gram-Schmidt orthogonalisation
    end for
    $p_{(i)} \leftarrow g_{(i)}$
    while $||g_{(i)}||^2 \geq tolerance$ and $count \leq max$ do
        if $count \bmod 5 = 0$ then
            for $j \in \{0, \dots, i-1\}$ do
                $x_{(i)} \leftarrow x_{(i)} - \frac{x_{(j)}^{\dagger} x_{(i)}}{||x_{(j)}||^2} x_{(j)}$    ▷ orthogonalising after every 5 iterations
            end for
            $x_{(i)} \leftarrow \frac{x_{(i)}}{||x_{(i)}||}$    ▷ normalising
            for $j \in \{0, \dots, i-1\}$ do
                $p_{(i)} \leftarrow p_{(i)} - \frac{x_{(j)}^{\dagger} p_{(i)}}{||x_{(j)}||^2} x_{(j)}$    ▷ orthogonalising after every 5 iterations
            end for
        end if
        $z \leftarrow Ap_{(i)}$    ▷ matrix-vector multiplication - memory intensive step
        $a \leftarrow \frac{p_{(i)}^{\dagger} z}{||p_{(i)}||^2} - x_{(i)}^{\dagger} y_{(i)}$
        $b \leftarrow 2 \frac{p_{(i)}^{\dagger} y_{(i)}}{||p_{(i)}||}$
```

```
29:             $r \leftarrow \sqrt{a^2+b^2}$
30:             $c \leftarrow \frac{1}{\sqrt{2}}\sqrt{1+\frac{a}{r}}$
31:             $s \leftarrow -\frac{b}{2cr}$
32:             $\boldsymbol{x}_{(i)} \leftarrow c\boldsymbol{x}_{(i)} + s\frac{\boldsymbol{p}_{(i)}}{||\boldsymbol{p}_{(i)}||}$
33:             $\boldsymbol{y}_{(i)} \leftarrow c\boldsymbol{y}_{(i)} + s\frac{\boldsymbol{z}}{||\boldsymbol{p}_{(i)}||}$
34:             $\mu_{(i)} \leftarrow \boldsymbol{x}^{\dagger}_{(i)}\boldsymbol{y}_{(i)}$
35:             $\boldsymbol{z} \leftarrow \boldsymbol{y}_{(i)} - \mu_{(i)}\boldsymbol{x}_{(i)}$
36:             $\boldsymbol{p}_{(i)} \leftarrow \boldsymbol{z} + c\frac{||\boldsymbol{z}||^2}{||\boldsymbol{g}_{(i)}||^2}(\boldsymbol{p}_{(i)} - (\boldsymbol{x}^{\dagger}_{(i)}\boldsymbol{p}_{(i)})\boldsymbol{x}_{(i)})$
37:             $\boldsymbol{g}_{(i)} \leftarrow \boldsymbol{z}$
38:             $count \leftarrow count + 1$
39:         end while
40:         $count \leftarrow 0$
41:     end for
42: end procedure
```

The above pseudocode was for the KS algorithm as a stand-alone. However, one round of KS iterations is alternated with a round of Jacobi iterations [9] to refine the eigenvector estimates. The whole process is briefly described below. The variable *tolerance* denotes the initial tolerance to which the KS iterations are done, while *rounds* specifies the total number of times the process of a set of KS iterations alternated with a Jacobi step is done.

```
 1: procedure KS+JACOBI($n$, $n_E$, $tolerance$, $rounds$)
 2:     $n \leftarrow$ input from user                          ▷ number of dimensions
 3:     $n_E \leftarrow$ input from user                       ▷ number of eigenvalues required
 4:     $tolerance \leftarrow$ input from user                 ▷ initial value of tolerance
 5:     $rounds \leftarrow$ input from user                    ▷ number of rounds of KS + Jacobi
 6:     for $i \in \{0,\dots,n_E-1\}$ do
```

```
7:          $\boldsymbol{x}_{(i)} \leftarrow$ initialised randomly
8:      **end for**
9:      **for** $i \in \{0, \ldots, rounds - 1\}$ **do**        ▷ This loop runs for the total number of rounds
10:         **function** KS($\boldsymbol{A}$, $\boldsymbol{x}$'s)
11:             Function body                                ▷ As described above
12:             **return** $\boldsymbol{x}$'s, $\mu$'s
13:         **end function**
14:         **for** $j \in \{0, \ldots, n_E - 1\}$ **do**
15:             **for** $k \in \{0, \ldots, n_E - 1\}$ **do**
16:                 $\boldsymbol{M}_{jk} = \boldsymbol{x}^{\dagger}_{(j)} \boldsymbol{A} \boldsymbol{x}_{(k)}$        ▷ This matrix is the input for the Jacobi step
17:             **end for**
18:         **end for**
19:         **function** JACOBI($\boldsymbol{M}$)
20:             This function finds the eigenvalues and eigenvectors of the matrix $\boldsymbol{M}$. The
    eigenvalues are just refined estimates of the eigenvalues of $\boldsymbol{A}$ that we have found. The $n_E$
    normalised eigenvectors $\boldsymbol{v}$'s of $\boldsymbol{M}$ are used to refine the eigenvector estimates of $\boldsymbol{A}$.
21:             **return** $\boldsymbol{v}$'s
22:         **end function**
23:         **for** $j \in \{0, \ldots, n_E - 1\}$ **do**
24:             $\boldsymbol{x}_{(j)} \leftarrow \sum_{k=0}^{n_E - 1} \boldsymbol{v}_{(j)k} \boldsymbol{x}_{(k)}$        ▷ Here $\boldsymbol{v}_{(j)k}$ denotes the $k$-th component of $\boldsymbol{v}_{(j)}$, the
    $j$-th normalised eigenvector of $\boldsymbol{M}$. $\boldsymbol{x}_{(j)}$ and $\boldsymbol{x}_{(k)}$ have their usual meanings. These $\boldsymbol{x}_{(j)}$'s
    will be used as the initial vectors in the next round of KS.
25:         **end for**
26:     **end for**                                      ▷ End of all rounds of KS + Jacobi
27: **end procedure**
```

The next section describes the technical implementation of the algorithm on the GPU.

3 GPU considerations and implementation

3.1 GPU considerations

Lattice QCD simulations are done on a lattice of finite lattice spacing a. However, to compare results with experiment, one must extrapolate to the continuum. This is a computationally resource intensive step. For instance, take a lattice whose extent is 4 fm (where 1 fm = 10^{-15} m) along all four directions. Now, say the lattice spacing a is 0.1 fm. That implies there are 40 lattice points along all 4 directions, or $40^4 = 2.56 \times 10^6$ points or sites in all. Since there are 4 links for each lattice site (starting at each site and pointing along each of the 4 positive axes), the number of links is 1.024×10^7. Now, if the lattice spacing a is finer, say 0.01 fm, the number of sites along each direction becomes 400. That implies the total number of sites in the lattice is $400^4 = 2.56 \times 10^{10}$. The number of links then becomes 1.024×10^{11}. While a lattice of 400 sites along each direction is unfeasible for most practical purposes, 40 sites along a direction is a reasonable-sized lattice. Current state of the art simulations can handle upto 80 sites along a direction.

In the example given above, for $a = 0.1$ fm, the matter fields would require a storage space of 1 GB while the gauge fields would require 1.5 GB. For the case of $a = 0.01$ fm, the corresponding figures are 10 TB for the matter fields and 15 TB for the gauge fields. It is clear that such simulations are very memory intensive.

Aside from memory, lattice QCD also seeks to optimise use of processor resources. Fortunately, some of the most processor-intensive steps (such as the action of a matrix on a vector) lend themselves easily to parallelisation. Overall, lattice QCD codes are relatively easy to parallelise.

Parallelisation between different nodes [10] (one node is a stand-alone computer in a network) using MPI like platforms [11] have been around for some time. This project is concerned with parallelisation among large number of cores on the same chip or node. Processors may be either CPUs (central processing units) or GPUs (graphical processing units). Lattice QCD on the GPU has been around for over ten years [12]. GPUs are preferred over CPUs for large-scale parallel programming because, even though an individual GPU core is, on average, less powerful than an individual CPU core, GPU chips contain far more cores than CPU chips. For example, on the Cray XC30 [13] (the parallel supercomputer on which this code runs) there are 476 nodes, each with an Intel Xeon E5-2680 chip and an Nvidia Tesla K20X GPU. The CPU chip contains 8 CPU cores with a base frequency of 2.80 GHz (which can be extended to a maximum of 3.50 GHz). The GPU chip, on the other hand, contains 2688 GPU cores operating at 732 MHz. The relevant GPU specifications are given below in table (1).

Property	**Quantity**
Processor clock	732 MHz
Memory clock	2.6 GHz
Memory size	6 GB
Memory bandwidth	250 GBps
Memory bus width	384-bit GDDR5
Interface type	PCI Express 3.0 x16
Bus type	PCI Express 3.0 x16
CUDA cores	2688

Table 1: Nvidia Tesla K20X GPU specifications

However useful GPUs may be for parallel computing, it is not possible to eliminate the CPU entirely. This is because essential utilities such as input-output and conditional statements cannot be evaluated on the GPU. This presents the problem of CPU-GPU communication, which is relatively slow and is the main bottleneck in most GPU applications. For instance, the speed of data transfer between CPU to GPU or vice-versa is about 5 $\mathrm{GB\,s^{-1}}$ or 8 $\mathrm{GB\,s^{-1}}$ depending on whether the data transfer protocol is PCI 2.0 or PCI 3.0 respectively. The main challenge in GPU programming is getting around such bottlenecks. The fact that CPU-GPU data transfer is slow implies as much of the data as possible should be present on the GPU. For instance, in the GPU implementation of the KS algorithm discussed above, the entire KS step was present on the GPU.

Another challenge is the GPU programming language. Most GPUs are programmed in CUDA [14], which is a C-like language. Since a lot of the existing lattice QCD codes are written in Fortran, it takes a reasonable effort to port them to CUDA. While a version of CUDA does exist for Fortran, it does not rewrite all CUDA functionality in Fortran but merely uses wrappers around the CUDA C functions to interface with the Fortran routines. This makes it inefficient in terms of memory and processor usage. A more efficient and easier alternative to the above is to use a directive-based programming tool such as OpenACC, which will be briefly discussed below.

OpenACC [15] is a programming standard for parallel computing developed by Cray, CAPS, Nvidia and PGI. The objective of OpenACC is to simplify parallel programming of systems that contain both CPU and GPU chips (and as a result will contain CPU-GPU data transfer). The OpenACC Application Program Interface (API) lists a collection of compiler directives (put inside the program in the form of comments or annotations) to specify loops and regions of code in standard C, C++ and Fortran programs, called *data regions*, which are to be processed on the GPU accelerator. A *compiler directive* is a direct instruction to the compiler on

how to process input. The OpenACC interface is flexible and can be used in various operating systems, host CPUs and accelerators. It is possible for programmers to write high-level programs involving both CPU host and GPU accelerator without needing to explicitly initialize the accelerator, manage data or program transfers between the host and accelerator, or initiate accelerator startup and shutdown [16]. The OpenACC API takes care of all the above functionalities.
While the compiler handles the actual data transfer, instructions on when to move data and what data to move must be specified by the programmer. This is an important task because, as discussed above, the main bottleneck in fast GPU programming is CPU-GPU data transfer. For this reason it is advantageous to have as much of the program inside the OpenACC data region as possible.
In the next part, the portion of the actual code which is implemented on the GPU is given.

3.2 GPU implementation of KS algorithm

The KS algorithm was implemented as the subroutine `eigen` of the Fortran file `ferm.f`. The entire code is split over multiple files, not all of which are used at every stage of operation. In the current stage of operation, the task is to act with the Dirac operator upon some pre-generated lattice configurations and find out the low-lying eigenspectrum.
The entire contents of the subroutine `eigen` are given in Appendix (A). The next section addresses the results obtained.

4 Results

4.1 Initial testing for small matrices

Initially the KS algorithm was implemented in C. A pseudo-random Hermitian matrix of size n was generated using the `srand()` function to which the calendar time was passed as seed. The matrix was then squared to obtain a Hermitian matrix whose eigenvalues are non-negative. The code was initially implemented without the Jacobi algorithm, i.e. only one round of KS iterations were completed. The results were verified using Mathematica [17].

4.2 Improvement of KS algorithm with Jacobi iterations

The code was modified to include Jacobi iterations. As mentioned in section (2.3), the Jacobi diagonalisations serve to refine the eigenvalue and eigenvector estimates of the large, sparse matrix $\boldsymbol{A}$ of size n. Suppose one wishes to find the lowest n_E eigenvalues and eigenvectors of

$\boldsymbol{A}$. After one round of KS iterations, one has n_E eigenvector estimates. Let $\boldsymbol{x}_{(j)}$ denote the eigenvector estimate corresponding to the j-th smallest eigenvalue. An $n_E \times n_E$ matrix $\boldsymbol{M}$ is constructed such that $\boldsymbol{M}_{jk} = \boldsymbol{x}^{\dagger}_{(j)} \boldsymbol{A} \boldsymbol{x}_{(k)}$. This matrix is the input for the Jacobi step.
Inside the Jacobi step, one diagonalises $\boldsymbol{M}$ and finds its eigenvalues and eigenvectors. Now, $\boldsymbol{M}$ is almost diagonal, but not quite. It has small, nonzero off-diagonal elements because the $\boldsymbol{x}_{(j)}$ are not exact eigenvectors of $\boldsymbol{A}$, but merely estimates upto some user-defined tolerance. They are refined as follows. One calculates the eigenvalues and corresponding eigenvectors of $\boldsymbol{M}$. Since $\boldsymbol{M}$ is a square matrix of size n_E, this is not very difficult. The eigenvalues of $\boldsymbol{M}$ serve as improved estimates of the lowest n_E eigenvalues of $\boldsymbol{A}$. The eigenvectors of $\boldsymbol{M}$ are used to refine the $\boldsymbol{x}_{(j)}$. Let $\boldsymbol{v}_{(j)}$ denote the j-th normalised eigenvector of $\boldsymbol{M}$ (i.e. corresponding to the j-th smallest eigenvalue), and $\boldsymbol{v}_{(j)k}$ denote its k-th component. One obtains improved eigenvector estimates of $\boldsymbol{A}$, denoted by $\boldsymbol{x}'$ by the following relation:

$$\boldsymbol{x}'_{(j)} = \sum_{k=1}^{n_E} \boldsymbol{v}_{(j)k} \boldsymbol{x}_{(k)}. \tag{43}$$

These $\boldsymbol{x}'$ serve as the initial search directions in the next round of KS.
Now, if $\boldsymbol{M}$ were exactly diagonal, $\boldsymbol{v}_{(j)}$ would be equal to $\hat{\boldsymbol{e}}_j$, where $\hat{\boldsymbol{e}}_j$ is the j-th column of the $n_E \times n_E$ identity matrix. However, since $\boldsymbol{M}$ has small off-diagonal components, each $\boldsymbol{v}_{(j)}$ in general differs slightly from the corresponding $\hat{\boldsymbol{e}}_j$. We quantify this difference by defining ρ, the *mixing*, such that

$$\rho(tolerance, round) = \ln\left(||\boldsymbol{v}_{(j)} - \hat{\boldsymbol{e}}_j||\right) \tag{44}$$

where *tolerance* is the initial tolerance of the KS step, and *round* denotes which round of KS+Jacobi one is considering. Intuitively one may expect ρ to decrease with finer tolerance and with increasing rounds of KS+Jacobi. This is exactly what happens. We measured ρ for $n = 20$, $n_E = 5$ and *tolerance* $\in \{10^{-3}, 10^{-4}, 10^{-5}\}$. For each *tolerance* we implemented 4 rounds of KS+Jacobi. The results are plotted below.

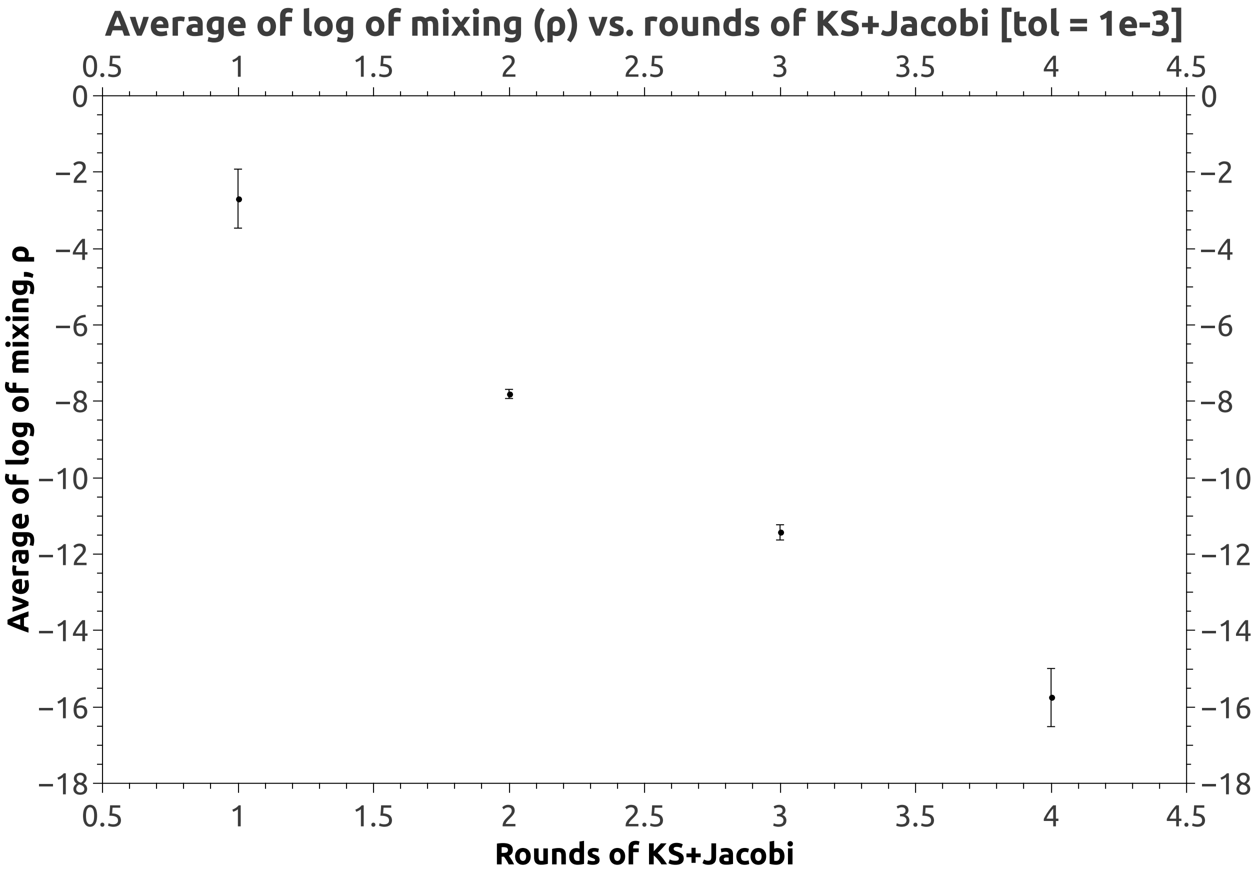

Figure 4: Average of the mixing, $\langle\rho\rangle$ vs. rounds of KS+Jacobi. $n = 20$; $n_E = 5$; initial tolerance $tol = 10^{-3}$ (tol updated as $tol \leftarrow tol/100$). Error bars denote standard deviation of ρ.

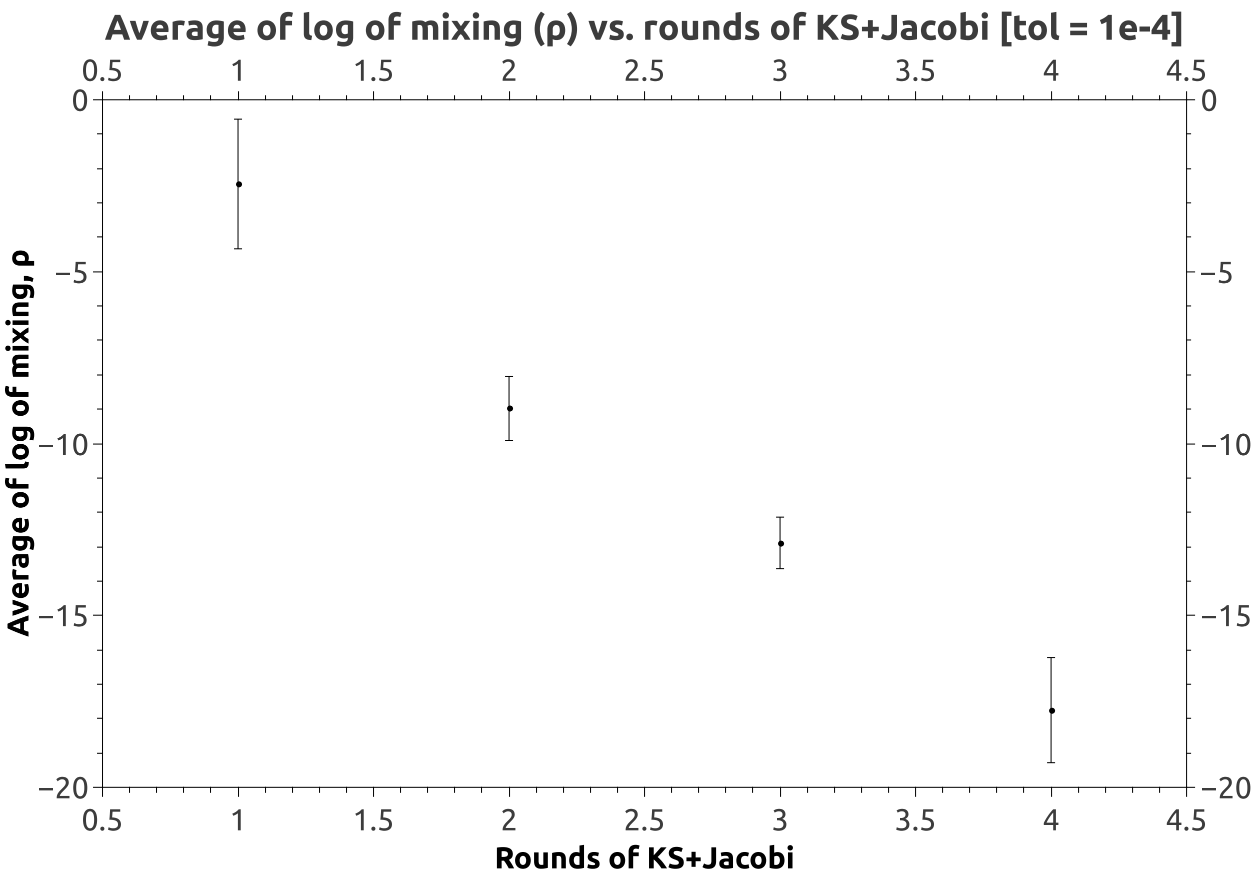

Figure 5: Average of the mixing, $\langle \rho \rangle$ vs. rounds of KS+Jacobi. $n = 20$; $n_E = 5$; initial tolerance $tol = 10^{-4}$ (tol updated as $tol \leftarrow tol/100$). Error bars denote standard deviation of ρ.

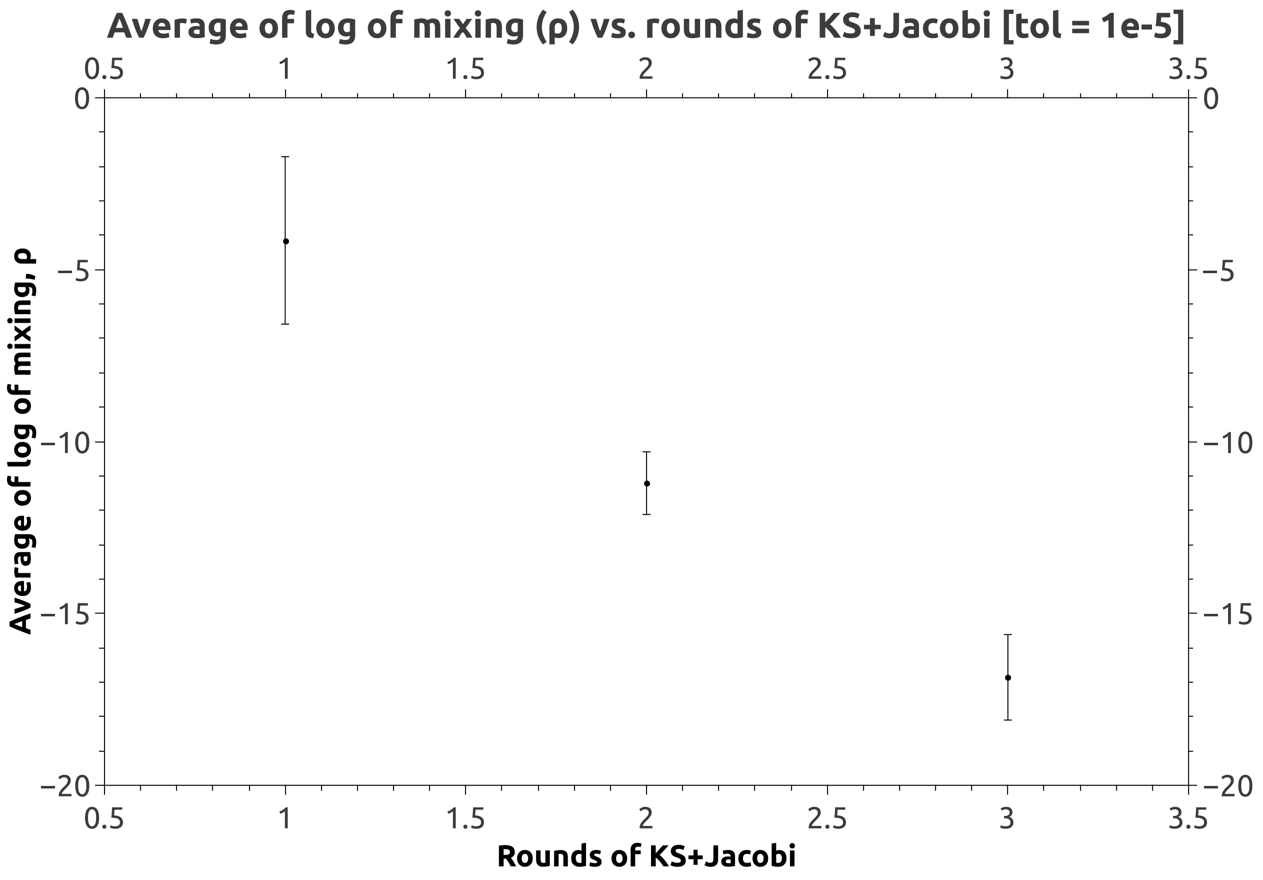

Figure 6: Average of the mixing ρ, $\langle\rho\rangle$ vs. rounds of KS+Jacobi. $n = 20$; $n_E = 5$; initial tolerance $tol = 10^{-5}$ (tol updated as $tol \leftarrow tol/100$). Error bars denote standard deviation of ρ. This figure only has 3 data-points because for the 4th round, $||\boldsymbol{v}_{(j)} - \hat{\boldsymbol{e}}_j||$ was equal to 0 upto machine precision.

From the figures above, we see that the $\boldsymbol{v}_{(j)}$ get closer to $\hat{\boldsymbol{e}}_j$ with each round of KS+Jacobi, and with decreasing *tolerance*. However, it is to be noted that the above study was performed with the C code for very small matrices ($n = 20$). For the full Fortran code, the initial results, while not conclusive, seem to suggest that the quantity $||\boldsymbol{v}_{(j)} - \hat{\boldsymbol{e}}_j||$ is almost 0 after the 1st round itself, i.e. there is not much mixing.

4.3 Improvement on the GPU

The KS algorithm with alternated Jacobi diagonalisations was implemented in Fortran as the subroutine `eigen` in the Fortran file `ferm.F`. It was run on the Cray XC30 parallel supercomputer whose specifications are given in section (3.1). Before being implemented on the GPU, however, it was implemented on the CPU of the Cray. A few timing studies were done. The code was optimised for the GPU using the OpenACC paradigm as discussed above. Initial

timing studies suggest that the performance on the GPU is roughly 10 to 15 times faster than on the CPU.
Some examples of how OpenACC works are provided below.
The OpenACC data region starts as follows:

```
!$ACC data
!$ACC+ copy(X0,x,y)
!$ACC+ copyin(u,cu,iup,idn,q,p,fm,X1)
!$ACC+ create(v,ap,atap,z,pp)
!$ACC+ copyout(pnorm,mu,error,niter)
```

Note that Fortran comments start with `!`. However, the combination `!$` is a compiler directive in Fortran (a direct instruction to the compiler), similar to `pragma` in C. The various commands in the code snippet above are:

- `$ACC data` starts the data region.
- `copy` copies data from CPU to GPU and then back to CPU after the data region ends.
- `copyin` copies from CPU to GPU.
- `copyout` creates data in GPU and copies to CPU at the end.
- `create` creates data local to the GPU.

Some common operations are reductions (where one or more arrays is given as input and a scalar obtained as output) and conditional statements. On the GPU they look like the following.

- A reduction operation:

```
!$ACC parallel loop present(y1,x) reduction(+: xnu)
             do lcv1=1,(nc*mvd2)
             xnu=xnu+(conjg(y1(lcv1))*x(lcv1))
             enddo
```

- A conditional statement:

```
!$ACC update host(error)
        if (error.le.tol) go to 5
        ! Statements
        5       continue
        endif
```

The code was tested on two pre-defined lattice configurations of size 24^4 and 32^4 respectively, and the results are shown in the next section.

4.4 Some timing studies on the GPU

Timing studies were performed on two lattices of volume 24^4 and 32^4 respectively. In both cases, the fermion mass was $m_f = 0.01$. The initial tolerance is always 5×10^{-5}, with *tolerance* updated as $tolerance \leftarrow tolerance/100$ after a round of Jacobi diagonalisations. There are 4 such rounds of KS+Jacobi, so the tolerance for the final round is 5×10^{-11}. For the 24^4 lattice, the total CPU time was measured for 16, 32, 64 and 128 eigenvalues (4 runs in each case). The timings were then averaged over the 4 runs and plotted as a function of the number of eigenvalues.

For the 32^4 lattice, 4 runs were performed for 16 eigenvalues, and one run each was performed for 32 and 64 eigenvalues. In each case the total CPU time was measured and plotted as a function of the number of eigenvalues. The results are given below.

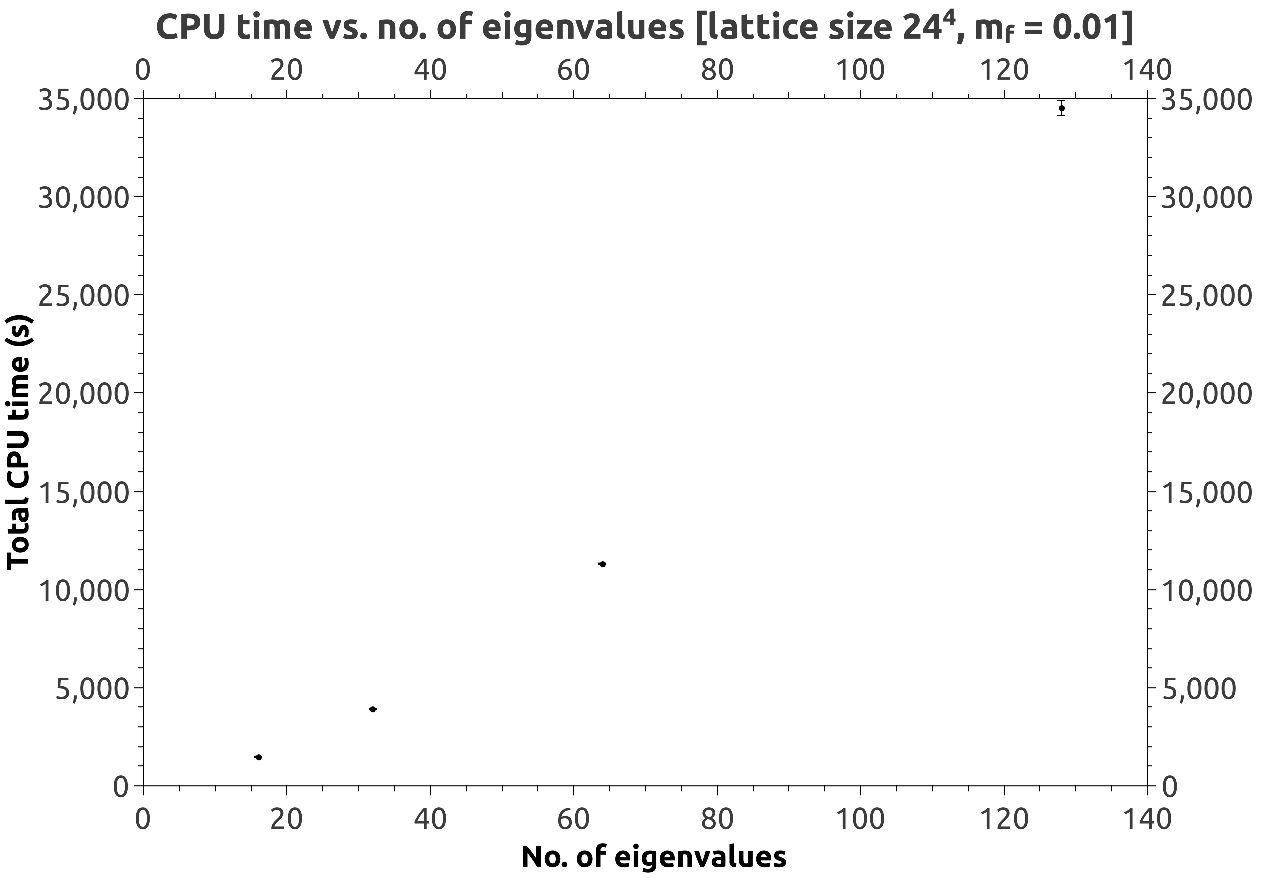

Figure 7: Average CPU time (over 4 runs) vs. no. of eigenvalues. Lattice size 24^4; fermion mass $m_f = 0.01$; initial tolerance $tol = 5 \times 10^{-5}$ (tol updated as $tol \leftarrow tol/100$); 4 rounds of KS+Jacobi.

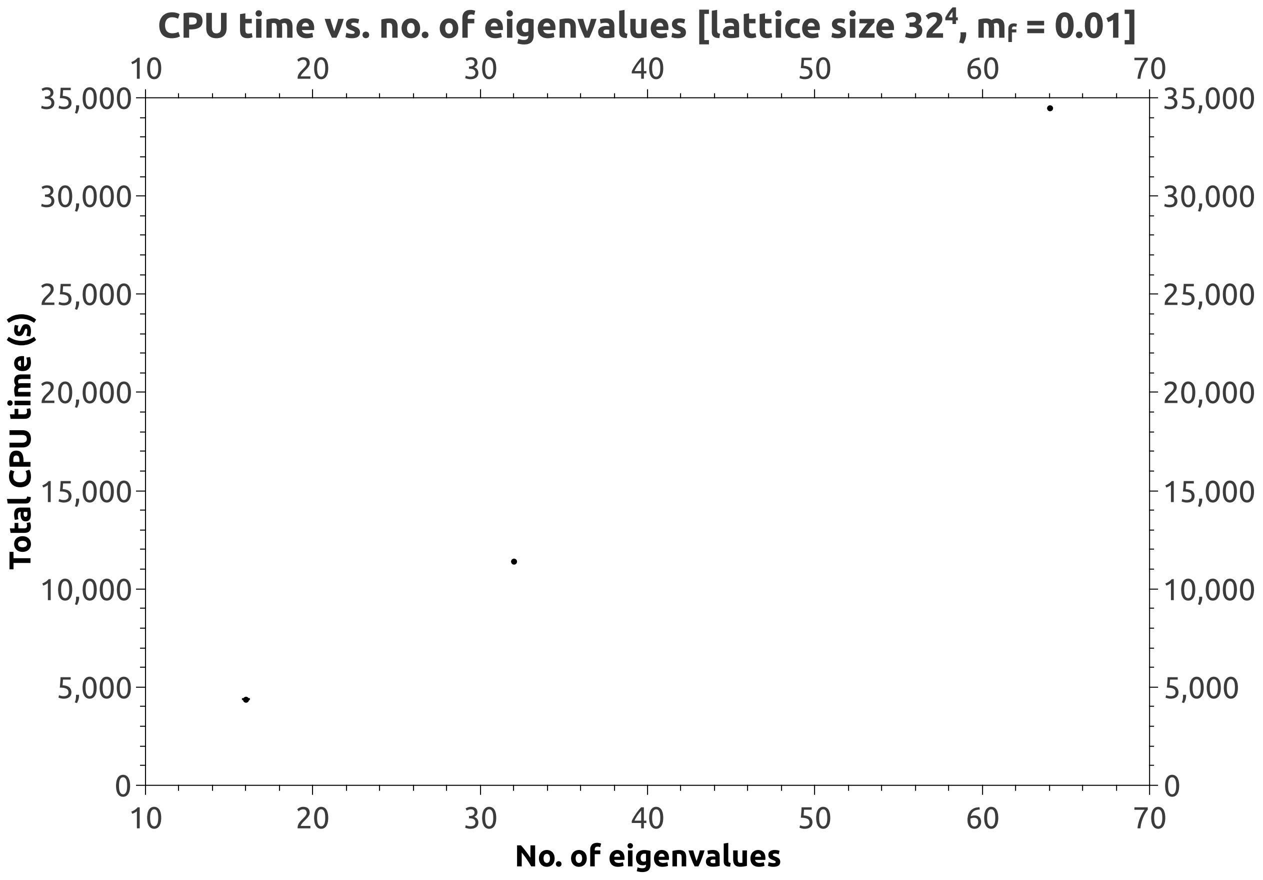

Figure 8: CPU time (averaged over 4 runs for 16 eigenvalues, one run each for 32 and 64-eigenvalue cases) vs. no. of eigenvalues. Lattice size 24^4; fermion mass $m_f = 0.01$; initial tolerance $tol = 5 \times 10^{-5}$ (tol updated as $tol \leftarrow tol/100$); 4 rounds of KS+Jacobi.

5 Conclusion

The initial testing on C for small matrices was done on a laptop. The purpose of this set of tests was to determine whether the algorithm worked, and results were verified with Mathematica. The variable parameters were the size of the matrix n, the initial tolerance tol and the number of rounds of KS with Jacobi diagonalisations. Testing was done for matrices of size $n = 50$ to $n = 200$. No clear trend was observed other than the obvious one, i.e. the program took a longer time to find $n_E = 20$ eigenvalues for larger n, all other parameters being held constant.
The Fortran code was tested on the Cray, initially on the CPU, using the OpenMP paradigm. It was then ported to the GPU using the OpenACC paradigm, resulting in a speedup of almost 15x over the same code implemented on the CPU. This is an encouraging result.
For the small matrix tests, the Jacobi diagonalisations were necessary, as shown by the decrease in the amount of mixing versus the number of rounds of KS+Jacobi (4.2). This proves

the Jacobi diagonalisations are necessary, in this case at least. However, whether the Jacobi diagonalisations have any effect in the Fortran case is still an open question - initial results seem to suggest that the amount of mixing is very low, which implies the Jacobi diagonalisations do not make much difference. Further testing is needed in this regard.

Overall, the KS algorithm proves to work well for large, sparse matrices. It is found to be relatively easy to parallelise as well.

The OpenACC paradigm is also found to be easy to use. It is powerful as well and provides a lot of functionalities for testing and debugging.

6 Afterword

I would like to begin by saying that, unlike all previous sections in this report, this section is a purely subjective and personal opinion of my experience in working on this project. Any views provided here are solely my own and do not necessarily reflect those of my guides or mentors, or anyone else in general.

The OpenACC paradigm is well-suited for GPU computation. It is easy to learn - even a complete novice can pick up the basic commands in a relatively short period of time.

Lattice QCD is by now a very well-established formalism and is extensively used for both theoretical calculations and numerical simulations on the computer. I did not get to learn much about lattice QCD itself, aside from a very brief introduction to some common techniques and methods used in it. My project was focused more on the KS algorithm and how to parallelise it for the GPU. Nevertheless, I appreciate the utility of the lattice in providing a computational framework to solve problems where analytical methods do not work very well.

On a broader level, I learned about the deeper underlying relations between quantum field theories, statistical field theories, and mathematical subjects such as topology and differential geometry. I realised a certain knowledge of topology and differential geometry is essential if one wishes to work in field theory, whether it is quantum field theory or statistical field theory. In the case of lattice gauge theories, a little knowledge of graph or network theory also helps, as the underlying lattice is discrete and may be thought of as a regular graph.

However, one limitation of lattice gauge theory that I noticed is that it is a well-defined theory only for a Euclidean lattice, i.e. a lattice of zero intrinsic curvature. This means that using lattice gauge theory, one can only study quantum fields in flat spacetime. This has given me an idea of the general direction in which I want to pursue research in the future. I know that computational techniques to simulate curved spaces such as H^3 and $H^2 \times E$ exist ([18] and [19]). Perhaps matter and gauge fields may be put on such curved lattices. For now the next step would be to do an extensive literature survey of all attempts to do QFT on curved spacetime lattices. Possible applications of QFT in curved spacetime could be to study Hawking radiation,

which is a high-energy process in the immediate vicinity of a black hole and therefore requires techniques from both QFT and gravitational physics.
On another level, this project gave me a deeper appreciation of computational techniques. I had been under the assumption that computers were merely a tool (albeit very powerful) by which one could perform calculations and simulate real-world physics. However, this project (specifically instances such as the fermion doubling problem) showed me that new, interesting physics emerges when one discretises the problem for computation. While it may be argued that such problems have no physical existence, I argue that they are interesting problems in their own right and thinking deeply about them might reveal new insights into our physical world as well as the workings of computers, which is an interesting subject in itself. I also developed an interest in theoretical computer science, independent of computer applications in physics. Clearly these are vast fields, and perhaps it will not be possible to do original research in every subject that I am interested in, but I think a basic working knowledge of other disciplines enriches research in one's own chosen field and encourages collaboration and cross-fertilisation.
In short, this project was an enormous learning experience, and I am grateful I had the opportunity to work on it. Ideally I would like to work in any discipline that requires knowledge and techniques from QFT, gravitational physics, statistical field theory (specifically the theory of phase transitions), the physics of scale change and scale transformations, the mathematical subjects of topology, differential geometry, and graph theory, as well as computer science. The study of quantum fields in discrete spacetime appears to be such a discipline, and so, for the moment at least, I would like to continue in this rich and diverse field.

References

[1] Address: https://profmattstrassler.com/articles-and-posts/particle-physics-basics/the-known-forces-of-nature/the-strength-of-the-known-forces/

[2] Brian C. Hall. Lie Groups, Lie Algebras, and Representations: An Elementary Introduction. Springer, 2015. DOI: http://dx.doi.org/10.1007/978-3-319-13467-3

[3] C. Gattringer, C.B. Lang. Quantum Chromodynamics on the Lattice: An Introductory Presentation. Springer, 2010. DOI: http://dx.doi.org/10.1007/978-3-642-01850-3

[4] J. Kogut, L. Susskind. Hamiltonian formulation of Wilson's lattice gauge theories. Physical Review D. 15 January 1975. DOI: https://doi.org/10.1103/PhysRevD.11.395

[5] C. Caramanis, S. Sanghavi. EE381V: Large Scale Learning. Department of Electrical and Computer Engineering at University of Texas, Austin. Address: http://users.ece.utexas.edu/~sanghavi/courses/scribed_notes/Lecture_13_and_14_Scribe_Notes.pdf

[6] 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. July 1995. DOI: https://doi.org/10.1016/0010-4655(95)00126-3

[7] V. Dick. Lattice Investigation of the DIRAC-Spectrum in the Vicinity of the Chiral Transition. Master thesis. Fakultät für Physik, Universität Bielefeld. 30 July 2012. Address: http://www2.physik.uni-bielefeld.de/fileadmin/user_upload/theory_e6/Master_Theses/Masterarbeit_ViktorDick.pdf

[8] V. Dick. The Chiral Anomaly of Quantum Chromodynamics at High Temperatures: Lattice Investigation of the Overlap Dirac Spectrum. Dissertation. Fakultät für Physik, Universität Bielefeld. April 2016. Address: https://pub.uni-bielefeld.de/publication/2903475

[9] C.G.J. Jacobi. Über ein leichtes Verfahren, die in der Theorie der Säkularstörungen vorkommenden Gleichungen numerisch aufzulösen. *Crelle's Journal*. 1846. Address: http://gdz.sub.uni-goettingen.de/dms/load/img/?PID=GDZPPN002144522

[10] Blaise Barney. Introduction to Parallel Computing. Lawrence Livermore National Laboratory. Address: https://computing.llnl.gov/tutorials/parallel_comp/

[11] Blaise Barney. OpenMP. Lawrence Livermore National Laboratory. Address: https://computing.llnl.gov/tutorials/openMP/

[12] G.I. Egri, Z. Fodor, C. Hoelbling, S.D. Katz, D. Nogradi, K.K. Szabo. Lattice QCD as a video game. June 2007. Address: arXiv:hep-lat/0611022v2. DOI: https://doi.org/10.1016/j.cpc.2007.06.005

[13] Indian Lattice Gauge Theory Initiative: The Cray XC30. Address: http://www.ilgti.tifr.res.in/machines/crayxc30.php

[14] Address: https://developer.nvidia.com/cuda-zone

[15] Address: https://www.openacc.org/

[16] Pushan Majumdar. Lattice Simulations using OpenACC compilers. November 2013. Address: arXiv:1311.2719v1

[17] Mathematica 8.0. Wolfram Research Inc. 2010. Address: http://www.wolfram.com

[18] Vi Hart, Andrea Hawksley, Elisabetta Matsumoto and Henry Segerman. Non-euclidean Virtual Reality I: Explorations of H^3. Proceedings of Bridges 2017: Mathematics, Art, Music, Architecture, Education, Culture. 2017. Address: http://archive.bridgesmathart.org/2017/bridges2017-33.pdf

[19] Vi Hart, Andrea Hawksley, Elisabetta Matsumoto and Henry Segerman. Non-euclidean Virtual Reality II: Explorations of $H^2 \times E$. Proceedings of Bridges 2017: Mathematics, Art, Music, Architecture, Education, Culture. 2017. Address: http://archive.bridgesmathart.org/2017/bridges2017-41.pdf

[20] John B. Kogut. An introduction to lattice gauge theory and spin systems. Rev. Mod. Phys. 51, 659. October 1979. DOI: https://doi.org/10.1103/RevModPhys.51.659.

[21] John B. Kogut. The lattice gauge theory approach to quantum chromodynamics. Rev. Mod. Phys. 55, 775. July 1983. DOI: https://doi.org/10.1103/RevModPhys.55.775

[22] DeGrand, T. and DeTar, C. Lattice Methods for Quantum Chromodynamics. World Scientific. 2006. Address: https://books.google.co.in/books?id=V48ddclvbioC

[23] Ramond, P. Field Theory: A Modern Primer. Avalon Publishing. 1997. Address: https://books.google.co.in/books?id=Ctr9K61fY4kC

[24] Matthias Blau. Notes on (Semi-)Advanced Quantum Mechanics: The Path Integral Approach to Quantum Mechanics. 2014. Address: http://www.blau.itp.unibe.ch/Lecturenotes.html

[25] Rothe, H.J. Lattice Gauge Theories: An Introduction. World Scientific. 2005. Address: https://books.google.co.in/books?id=U1hBLG-_WxAC

[26] J.M. Drouffe and C. Itzykson. Lattice Gauge Fields. Facts and Prospects of Gauge Theories, pp. 203-204. Springer. 1978. DOI: https://doi.org/10.1007/978-3-7091-8538-4_5

[27] Lepage, G Peter. Lattice QCD for novices. Strong Interactions at Low and Intermediate Energies, pp. 49-90. World Scientific. DOI: https://doi.org/10.1142/9789812793225

[28] Pokorski, Stefan. Gauge Field Theories (Cambridge Monographs on Mathematical Physics). Cambridge: Cambridge University Press. 2000. DOI: https://doi.org/10.1017/CBO9780511612343

[29] Schwartz, M.D. Quantum Field Theory and the Standard Model (Quantum Field Theory and the Standard Model). Cambridge University Press. 2014. Address: https://books.google.co.in/books?id=HbdEAgAAQBAJ

[30] Ryder, L.H. Quantum Field Theory. Cambridge University Press. 1996. Address: https://books.google.co.in/books?id=nnuW_kVJ500C

Appendices

A GPU implementation of KS algorithm in Fortran

The subroutine eigen is as follows:

```
      subroutine eigen(nEigen)
!This is the Kalkreuter-Simma routine.
      use size
      use param1
      use param2, only : fm
      use config
      use donfig
      use iupidn
      use ranlxd_generator
      use ksjParameters
      complex, dimension (mvd2,nc) :: v, ap, atap
      complex :: px1, px2, px3, px4, px5, px6, v1, v2, v3
      complex :: mu,xnu,x00,temp1,temp2 ! temp1 and temp2 will store values in the ma
      complex, dimension(nEigen,nc*mvd2) :: X0,X1
      complex, dimension(nc*mvd2) :: w,x,y,y1,z,q,p,pp
      complex, dimension(nEigen) :: Eigval
      real, dimension(2*nEigen,2*nEigen) :: JB,Y0
      real, dimension(nc*mvd2) :: w1,w2
      real, dimension(60) :: timer !timer contains total time taken for each parallel
      real :: xnorm,pnorm,a,b,r,c,s,err0,error,tol,jb1,jb2,temp3
      integer, dimension(nEigen) :: eval,ksiter
      integer :: lcv1, lcv2 ! Loop control variables exclusively for matrix-vector op
      call makhc
      nitcg=1
      tol=tolInit
!$OMP parallel do collapse(3) default(private) shared(cu,u,idn)
        do nn=1,mb
         do ic1=1,nc
         do ic2=1,nc
            imu=int((nn-1)/mv)
            nd=idn(nn)+imu*mv
            cu(nn,ic1,ic2)=conjg(u(nd,ic1,ic2))
         enddo
         enddo
        enddo
!$OMP parallel do collapse(2) default(shared) private(lcv1,lcv2)
        do lcv2=1,(nc*mvd2)
           do lcv1=1,nEigen
              X0(lcv1,lcv2)=(0.0,0.0)
           enddo
        enddo
!$OMP end parallel do
        do i=1,nEigen
          call ranlxd(w1)
          call ranlxd(w2)
!$OMP parallel do default(shared) private(j)
          do j=1,nc*mvd2
            x(j)=cmplx(w1(j),w2(j))
          enddo
!$OMP end parallel do
        temp1 = cmplx(0.0,0.0)
!$OMP parallel do default(shared) reduction(+: temp1) private(lcv1)
        do lcv1=1,(nc*mvd2)
            temp1=temp1+(conjg(x(lcv1))*x(lcv1))
        enddo
!$OMP end parallel do
        xnorm=sqrt(temp1)
!$OMP parallel do default(shared) private(lcv1)
        do lcv1=1,(nc*mvd2)
            x(lcv1)=(x(lcv1))/xnorm
        enddo
!$OMP end parallel do
!$OMP parallel do default(shared) private(lcv1)
        do lcv1=1,(nc*mvd2)
         X1(i,lcv1)=x(lcv1)
        enddo
!$OMP end parallel do
        enddo
        do kk=1,rounds  ! loop over # of Jacobi diagonalizations
        write(*,*) "Jacobi iteration:",kk,"; tolerance=",tol,"."
        do j=1,nEigen ! loop over #of eigenvalues
!$ACC data
!$ACC+ copy(X0,x,y)
!$ACC+ copyin(u,cu,iup,idn,q,p,fm,X1)
!$ACC+ create(v,ap,atap,z,pp)
!$ACC+ create(x00,y1,xnu,xnorm)
!$ACC+ copyout(pnorm,mu,error,niter)
!$ACC parallel loop present(x,X1)
        do lcv1=1,(nc*mvd2)
         x(lcv1)=X1(j,lcv1)
        enddo
        do i=1,nEigen
!$ACC parallel
        x00=X0(i,1)
!$ACC end parallel
!$ACC update host(x00)
          if (x00.NE.0) then
!$ACC parallel loop present(y1,X0)
          do lcv1=1,(nc*mvd2)
           y1(lcv1)=X0(i,lcv1)
          enddo
!$ACC parallel
             xnu=cmplx(0.0,0.0)
!$ACC end parallel
!$ACC parallel loop present(y1,x) reduction(+: xnu)
             do lcv1=1,(nc*mvd2)
                 xnu=xnu+(conjg(y1(lcv1))*x(lcv1))
             enddo
!$ACC parallel loop present(x,y1)
             do lcv1=1,(nc*mvd2)
                 x(lcv1)=x(lcv1)-(xnu*y1(lcv1))
             enddo
          endif
        enddo
!$ACC parallel
        temp1 = cmplx(0.0,0.0)
```

```
!$ACC end parallel
!$ACC parallel loop present(x) reduction(+: temp1)
        do lcv1=1,(nc*mvd2)
            temp1=temp1+(conjg(x(lcv1))*x(lcv1))
        enddo
!$ACC parallel
        xnorm=sqrt(temp1)
!$ACC end parallel
!$ACC parallel loop present(x,xnorm)
        do lcv1=1,(nc*mvd2)
            x(lcv1)=(x(lcv1))/xnorm
        enddo
!$ACC parallel loop present(v,x)
         do i=1,mvd2*3,3
           k=(i+2)/3
           v(k,1)=x(i)
           v(k,2)=x(i+1)
           v(k,3)=x(i+2)
         enddo
!$ACC parallel loop collapse(2) present(ap)
         do lcv1=1,mvd2
          do lcv2=1,nc
           ap(lcv1,lcv2)=cmplx(0.0,0.0)
          enddo
         enddo
         call fmv_acc(v,ap)
!$ACC parallel loop collapse(2) present(fm,atap,v)
         do lcv1=1,mvd2
          do lcv2=1,nc
           atap(lcv1,lcv2)=4*fm*fm*v(lcv1,lcv2)
          enddo
         enddo
         call fmtv_acc(ap,atap)
!$ACC parallel loop present(atap,y)
         do i=1,mvd2*3,3
           k=(i+2)/3
           y(i)=atap(k,1)
           y(i+1)=atap(k,2)
           y(i+2)=atap(k,3)
         enddo
!$ACC parallel
        mu= cmplx(0.0,0.0)
!$ACC end parallel
!$ACC parallel loop present(x,y) reduction(+: mu)
        do lcv1=1,(nc*mvd2)
            mu=mu+(conjg(x(lcv1))*y(lcv1))
        enddo
!$ACC parallel loop present(q,y,mu,x)
        do lcv1=1,(nc*mvd2)
            q(lcv1) = y(lcv1) - mu*x(lcv1)
            p(lcv1)=q(lcv1)
        enddo
        do i=1,nEigen
!$ACC parallel
        x00=X0(i,1)
!$ACC end parallel
!$ACC update host(x00)
          if (x00.NE.0) then
!$ACC parallel loop present(y1,X0)
             do lcv1=1,(nc*mvd2)
              y1(lcv1)=X0(i,lcv1)
             enddo
!$ACC parallel
             xnu=cmplx(0.0,0.0)
!$ACC end parallel
!$ACC parallel loop present(y1,p) reduction(+: xnu)
             do lcv1=1,(nc*mvd2)
              xnu=xnu+(conjg(y1(lcv1))*p(lcv1))
             enddo
!$ACC parallel loop present(p,y1)
             do lcv1=1,(nc*mvd2)
              p(lcv1)=p(lcv1)-(xnu*y1(lcv1))
             enddo
          endif
        enddo
!KALKREUTER-SIMMA (OUTER LOOPS) BEGINS.
        do nx=1,2000 ! outer loop over iterations
        do ny=1,10 ! After every 10 iterations, a Gram-Schmidt orthogonalisation.
!$ACC parallel loop present(v,p)
         do i=1,mvd2*3,3
           k=(i+2)/3
           v(k,1)=p(i)
           v(k,2)=p(i+1)
           v(k,3)=p(i+2)
         enddo
!$ACC parallel loop collapse(2) present(ap)
         do lcv1=1,mvd2
          do lcv2=1,nc
           ap(lcv1,lcv2)=cmplx(0.0,0.0)
          enddo
         enddo
         call fmv_acc(v,ap)
!$ACC parallel loop collapse(2) present(atap,v,fm)
         do lcv1=1,mvd2
          do lcv2=1,nc
           atap(lcv1,lcv2)=4*fm*fm*v(lcv1,lcv2)
          enddo
         enddo
         call fmtv_acc(ap,atap)
!$ACC parallel loop present(z,atap)
         do i=1,mvd2*3,3
           k=(i+2)/3
           z(i)=atap(k,1)
           z(i+1)=atap(k,2)
           z(i+2)=atap(k,3)
         enddo
!       The actual KS algorithm.
!$ACC parallel
        temp1=cmplx(0.0,0.0)
!$ACC end parallel
!$ACC parallel loop present(p) reduction(+: temp1)
```

```
        do lcv1=1,(nc*mvd2)
         temp1=temp1+(conjg(p(lcv1))*p(lcv1))
        enddo
!$ACC parallel
        pnorm=temp1
!$ACC end parallel
!$ACC parallel
        temp1=cmplx(0.0,0.0)
        temp2=cmplx(0.0,0.0)
!$ACC end parallel
!$ACC parallel loop present(p,z,x,y)
!$ACC+ reduction(+: temp1,temp2)
        do lcv1=1,(nc*mvd2)
         temp1=temp1+(conjg(p(lcv1))*z(lcv1))
         temp2=temp2+(conjg(x(lcv1))*y(lcv1))
        enddo
!$ACC parallel
        temp1=temp1/pnorm
        a=temp1-temp2
!$ACC end parallel
!$ACC parallel
        temp1=cmplx(0.0,0.0)
!$ACC end parallel
!$ACC parallel loop present(p,y) reduction(+: temp1)
        do lcv1=1,(nc*mvd2)
         temp1=temp1+(conjg(p(lcv1))*y(lcv1))
        enddo
!$ACC parallel
        b=2*temp1/sqrt(pnorm)
        r=sqrt(a*a + b*b)
        c=sqrt(1+a/r)/sqrt(2.0)
        s=-b/(2*c*r)
!$ACC end parallel
!$ACC parallel loop present(x,p,y,z)
        do lcv1=1,(nc*mvd2)
         x(lcv1)=c*x(lcv1)+s*p(lcv1)/sqrt(pnorm) !
         y(lcv1)=c*y(lcv1)+s*z(lcv1)/sqrt(pnorm) !
        enddo
!$ACC parallel
        mu=cmplx(0.0,0.0)
!$ACC end parallel
!$ACC parallel loop present(x,y) reduction(+: mu)
        do lcv1=1,(nc*mvd2)
         mu=mu+(conjg(x(lcv1))*y(lcv1))
        enddo
!$ACC parallel
        err0=cmplx(0.0,0.0)
!$ACC end parallel
!$ACC parallel loop present(q) reduction(+: err0)
        do lcv1=1,(nc*mvd2)
         err0=err0+(conjg(q(lcv1))*q(lcv1))
        enddo
!$ACC parallel
        temp1=cmplx(0.0,0.0)
!$ACC end parallel
!$ACC parallel loop present(x,p) reduction(+: temp1)
        do lcv1=1,(nc*mvd2)
         temp1=temp1+(conjg(x(lcv1))*p(lcv1))
        enddo
!$ACC wait
!$ACC parallel loop present(pp,p,x)
        do lcv1=1,(nc*mvd2)
         pp(lcv1)=p(lcv1)-(temp1*x(lcv1))
        enddo
!$ACC parallel loop present(q,y,x,z)
        do lcv1=1,(nc*mvd2)
         q(lcv1) = y(lcv1) - mu*x(lcv1) !
         z(lcv1)=q(lcv1) !
        enddo
!$ACC parallel
        error=cmplx(0.0,0.0)
!$ACC end parallel
!$ACC parallel loop present(z) reduction(+: error)
        do lcv1=1,(nc*mvd2)
```

```
         error=error+(conjg(z(lcv1))*z(lcv1))
        enddo
!$ACC parallel loop present(p,z,pp)
        do lcv1=1,(nc*mvd2)
         p(lcv1)=z(lcv1) + c*error*pp(lcv1)/err0 !
        enddo
!$ACC parallel
        niter=(nx-1)*10+ny
!$ACC end parallel
!$ACC update host(error)
        if (error.le.tol) go to 5
        enddo  ! inner loop over iterations ends (after this ends, one round of Gram-
3       continue
        do i=1,nEigen
!$ACC parallel
          x00=X0(i,1)
!$ACC end parallel
!$ACC update host(x00)
          if (x00.NE.0) then
!$ACC parallel loop present(y1,X0)
             do lcv1=1,(nc*mvd2)
              y1(lcv1) = X0(i,lcv1) !
             enddo
!$ACC parallel
             xnu=cmplx(0.0,0.0)
!$ACC end parallel
!$ACC parallel loop present(y1,p) reduction(+: xnu)
             do lcv1=1,(nc*mvd2)
              xnu=xnu+(conjg(y1(lcv1))*p(lcv1))
             enddo
!$ACC parallel loop present(p,y1)
             do lcv1=1,(nc*mvd2)
              p(lcv1)=p(lcv1)-xnu*y1(lcv1) !
             enddo
!$ACC parallel
             xnu=cmplx(0.0,0.0)
!$ACC end parallel
```

```
!$ACC parallel loop present(y1,x) reduction(+: xnu)
             do lcv1=1,(nc*mvd2)
              xnu=xnu+(conjg(y1(lcv1))*x(lcv1))
             enddo
!$ACC parallel loop present(x,y1)
             do lcv1=1,(nc*mvd2)
              x(lcv1)=x(lcv1)-xnu*y1(lcv1) !
             enddo
!$ACC parallel
             xnorm=cmplx(0.0,0.0)
!$ACC end parallel
!$ACC parallel loop present(x) reduction(+: xnorm)
             do lcv1=1,(nc*mvd2)
              xnorm=xnorm+(conjg(x(lcv1))*x(lcv1))
             enddo
!$ACC parallel
             xnorm=sqrt(xnorm)
!$ACC end parallel
!$ACC parallel loop present(x)
             do lcv1=1,(nc*mvd2)
              x(lcv1)=(x(lcv1))/xnorm
             enddo
          endif
        enddo
        enddo   ! outer loop over iterations ends (one complete round of KS iteration
5       continue
!$ACC parallel loop present(X0,x)
        do lcv1=1,(nc*mvd2)
         X0(j,lcv1)=x(lcv1)
        enddo
!$ACC end data
        write(*,*) j, "Lowest eigenvalue=",mu,"; Iterations=",niter
        enddo ! loop over number of eigenvalues ends
!$OMP parallel do collapse(2) default(shared) private(lcv1,lcv2)
        do lcv1=1,nEigen
         do lcv2=1,nEigen
          JB(lcv1,lcv2)=0.0
         enddo
        enddo
!$OMP end parallel do
        do j1=1,nEigen
        do j2=1,nEigen
           if (X0(j1,1).NE.0) then
           if (X0(j2,1).NE.0) then
!$OMP parallel do default(shared) private(lcv1)
           do lcv1=1,(nc*mvd2)
              q(lcv1)=X0(j1,lcv1)
              p(lcv1)=X0(j2,lcv1)
           enddo
!$OMP end parallel do
!$OMP parallel do shared(v,p) private(i,k)
         do i=1,mvd2*3,3
           k=(i+2)/3
           v(k,1)=p(i)
           v(k,2)=p(i+1)
           v(k,3)=p(i+2)
         enddo
!$OMP parallel do collapse(2) default(shared) private(lcv1,lcv2)
         do lcv1=1,mvd2
          do lcv2=1,nc
           ap(lcv1,lcv2)=cmplx(0.0,0.0)
          enddo
         enddo
!$OMP end parallel do
         call fmv_omp(v,ap)
!$OMP parallel do collapse(2) default(shared) private(lcv1,lcv2)
         do lcv1=1,mvd2
          do lcv2=1,nc
           atap(lcv1,lcv2)=4*fm*fm*v(lcv1,lcv2)
          enddo
         enddo
!$OMP end parallel do
         call fmtv_omp(ap,atap)
!$OMP parallel do shared(z,atap) private(i,k)
```

```
          do i=1,mvd2*3,3
            k=(i+2)/3
            z(i)=atap(k,1)
            z(i+1)=atap(k,2)
            z(i+2)=atap(k,3)
          enddo
               temp1=cmplx(0.0,0.0)
!$OMP parallel do default(shared) reduction(+: temp1) private(lcv1)
               do lcv1=1,(nc*mvd2)
                temp1=temp1+(conjg(q(lcv1))*z(lcv1))
               enddo
!$OMP end parallel do
               jb1=real(temp1)
               jb2=aimag(temp1)
               JB(j1,j2)=jb1
               JB(j1+nEigen,j2+nEigen)=jb1
               JB(j1,j2+nEigen)=-jb2
               JB(j1+nEigen,j2)=jb2
            endif
            endif
         enddo
         enddo
         call Jacobi(JB,Y0,tol,2*nEigen)
!$OMP parallel do collapse(2) default(shared) private(lcv1,lcv2)
         do lcv2=1,(nc*mvd2)
            do lcv1=1,nEigen
               X1(lcv1,lcv2)=cmplx(0.0,0.0)
            enddo
         enddo
!$OMP end parallel do
!$OMP parallel do collapse(3) default(shared) reduction(+: temp1)
!$OMP+ private(lcv1,lcv2)
        do lcv1=1,(nc*mvd2)
         do i=1,nEigen
          do j=1,nEigen
          X1(i,lcv1)=X1(i,lcv1)+cmplx(Y0(i,j),Y0(i+nEigen,j))*X0(j,lcv1) !
          enddo
         enddo
        enddo
!$OMP end parallel do
!$OMP parallel do collapse(2) default(shared) private(lcv1,lcv2)
         do lcv2=1,(nc*mvd2)
            do lcv1=1,nEigen
               X0(lcv1,lcv2)=(0.0,0.0)
            enddo
         enddo
!$OMP end parallel do
         tol=tol/100.0
         enddo ! loop over number of Jacobi iterations ends
         stop
         end
```