Semester Project: Adaptive Multigrid Preconditioned Generalised Conjugate Residual Dirac Solver in LatticeQCD [8]

Author: JingJing Li, Supervisor: Roman Gruber, Prof Marina Marinkovic

Abstract

In Lattice Quantum Chromodynamics simulations, the inversion of the Dirac Operator is a major computation bottleneck, due to the $O(V)$ dimension, where V is the volume of the lattice, the neighbour interaction, the poor condition number under physical conditions, and the many times it is applied in a single simulation. In this semester project, we implemented the Adaptive Multigrid as a preconditioned Generalised Conjugate Residual solver that targets the Dirac Operator. The theory behind multigrid preconditioning and techniques used in my implementation will be explained before the improved convergence is discussed.

1 Introduction

Quantum Chromodynamics is the study of strong forces between quarks which is mediated by gluons. It is a $SU(3)$ Quantum Field Theories. The QCD Dirac Operator has a complex form due to the interplay among its many degrees of freedom. One of the computational approaches developed to make predictions of observables is Lattice Quantum Chromodynamics (LatticeQCD), where the spacetime is discretised into a four-dimensional lattice. The infinite integral is approximated by a Hamiltonian Monte-Carlo sampling of field configuration space because Monte-Carlo error only depends on the number of sample points and does not suffer from high dimensions of the integration unlike many numerical quadrature rules ("curse of dimensionality"). For every Monte-Carlo sample, a large linear system of equations (Dirac Operator) that scales $O(N^4)$ with regard to lattice dimension N, and with potentially bad condition number needs to be solved. The poor scaling is a limiting factor to Lattice QCD simulation reaching the continuum limit, and the poor condition number is a barrier to simulations with physical mass [1].

This semester project is focused on exploring the numerical method of multigrid(MG) preconditioning appied to the Generalised Conjugate Gradient (GCR) Dirac Solver from both theoretical and implementation perspectives. The reason why MG preconditioning enhances the performance of the GCR solver will be explained in theory; the programming techniques used in the C++ implementation and code examples are briefly discussed in implementation; and the improved result on test Dirac Operators are presented in result.

2 Theory

2.1 Wilson-Dirac

The QCD Lagrangian that fundamentally governs the behaviour of strong forces is written as

$$L_{QCD} = \bar{\psi}(i\gamma^\mu D_\mu - m)\psi - \frac{1}{2}Tr(G_{\mu\nu}G^{\mu\nu}), \tag{1}$$

where

$$D_{\mu,ij} = \partial_\mu \delta_{ij} - igT^a_{ij}A^a_\mu \tag{2}$$

$$G^a_{\mu\nu} = \partial_\mu A^a_\nu - \partial_\nu A^a_\mu + gf^{abc}A^b_\mu A^c_\nu \tag{3}$$

ψ is the fermion, A is the gauge boson, T is the generator of $SU(3)$, f is the structure constant, γ is the set of gamma matrices, g is the coupling constant, μ is the Lorentz index, i, j are the spinor indices, a is the colour index.

The Dirac operator, which describes the dynamics of fermions in the theory, comes from the fermionic equation of motion derived from the QCD Lagrangian.

$$D_{dirac} = i\gamma^\mu D_\mu - m \tag{4}$$

The Wilson-Dirac Operator is a modified lattice discretization of the Dirac operator. The fermion doubling problem is remedied in the Wilson approach by a diffusive term, which can also be seen as an offset to the mass term. U is the discretised gauge boson which can be considered to be the field residing on the link between two neighbouring fermions. The Wilson-Dirac operator introduces a lattice spacing and a mass term, allowing for the study of chiral symmetry breaking and other important phenomena.

$$D_{wilson-dirac}(x,y) = -\frac{1}{2}\sum_\mu (1-\gamma^\mu_{ij})U^{\mu,ab}_x \delta_{x+\bar{\mu},y} + (1+\gamma^\mu_{ij})U^{\mu,ab}_x \delta_{x-\bar{\mu},y} \tag{5}$$

The solving of the Dirac equation is chosen as the topic of this project because the Dirac inversion is considered a major limiting factor for the performance of Lattice simulations. In a typical LatticeQCD simulation, especially in the evaluation of observables, the Dirac equation is inverted a large number of times and inverse problems are computationally challenging by nature.

2.2 Generalised Conjugate Residual (GCR)

Krylov solvers are commonly used to solve large and sparse linear systems of equations. In this project, Generalised Conjugate Residual (GCR), one of the Krylov solvers compatible with the non-Hermitian Dirac operator, is the solver of choice. GCR is an iterative method that enjoys ensured convergence after N steps, where N is the dimension of the matrix [6]. The pseudo-code for solving $Ax = b$ is shown below [10].

Algorithm 1: Generalised Conjugate Residual Method

Input: Matrix A, vector b, initial guess x_0
Output: Approximate solution x

Initialize $r_0 = b - Ax_0$
Initialize $p_0 = r_0$
for $k = 0, 1, 2, \ldots$ *until convergence* **do**
  $\alpha_k = \frac{r_k^T r_k}{(Ap_k)^T Ap_k}$
  $x_{k+1} = x_k + \alpha_k p_k$
  $r_{k+1} = r_k - \alpha_k Ap_k$
  Compute $\beta_{j,k} = \frac{Ar_{k+1}^T Ap_j}{Ap_j^T Ap_j}$ for $j = 0, 1, ..., k$
  $p_{k+1} = r_{k+1} + \sum \beta_{j,k} p_j$
end

In this approach, the most computationally intense step is the sparse matrix-vector products Ar and Ap per iteration. The other operations such as dot product and vector addition have negligible cost in comparison. Therefore the total computational cost is roughly the number of iterations $\times$ cost of the sparse matrix-vector product.

A naive GCR progressively constructs a Krylov subspace by keeping all previous directions of the residuals i.e. Ap_j and p_j, which is not sustainable for large systems like the Dirac Operator. One common trick is to do restarts: the stored directions are regularly scratched, and the GCR is restarted. This approach is memory efficient given the constant memory requirement as opposed to linear growth with regard to iteration for the naive GCR.

2.2.1 Preconditioning

One crucial shortcoming of GCR and other Krylov solvers like Conjugate Gradient is their sensitivity to the condition number of the system they are solving. The condition number of a matrix A, $\kappa(A)$, is defined as the ratio between the magnitudes of its largest eigenvalue and its smallest. In theory, the reduction in residual per step, or convergence rate, is $\frac{\sqrt{\kappa(A)}-1}{\sqrt{\kappa(A)}+1}$. This means that in the limit of large condition numbers, the convergence is approximately $1 - \frac{2}{\sqrt{\kappa(A)}}$, approaching rate 1 which equates to non-convergence. Unfortunately, the Dirac equation for large lattices with physical quark mass falls under the poorly conditioned category, also termed the critical slowing down region.

To circumvent this issue, a numerical technique called preconditioning is used. A second matrix B is applied to both the left and the right-hand side of the equation. And B, the preconditioner, is designed such that the new operator to invert BA (or AB if right-preconditioned) is well-conditioned. In practice, B is an approximate for A^{-1}, in many cases, another iterative solver.
Left preconditioning

$$BAx = By \tag{6}$$

or right preconditioning

$$ABx = By \tag{7}$$

2.3 Multigrid Method

Multigrid is a stationary linear iteration where in each iteration the solutions on a number of predetermined reduced spaces are added up. The most intuitive example of the multigrid method is the geometric multigrid method which involves solving on a sequence of regularly refined coarse grids. Thanks to the many levels of coarsening, multigrid methods are good at resolving solutions across the spectrum. This property made it compatible as a preconditioner for Krylov solvers that tend to undermine low modes.

A multigrid solver has 3 main components, the prolongation and restriction operators, the coarse grid correction, and the smoothing operator. The prolongation operator is essentially the projection matrix that transforms the matrix from a coarse basis to a fine basis and restriction is the opposite. The coarse grid correction is an exact solution on the coarsest level. The smoothing operator is a solver defined on the fine grid that smooths the prolongated coarse grid correction, for example, a Gauss-seidel sweep. The main computation of the MG algorithm is structured as follows.

Algorithm 2: Multigrid Algorithm

Input : Matrix A, initial guess x, right-hand side b, level l
Output: Approximate solution x

if $l = L$ **then**
  Coarsest grid Solve $Ax = b$ exactly;
end
else
  Perform pre-smoothing: $x \leftarrow \text{Smoothing}(A, x, b)$;
  Compute residual: $r \leftarrow b - Ax$;
  Restrict residual: $r_c \leftarrow P^H r$;
  Restrict A on the coarse grid: $A_c = P^H AP$ Set initial guess on coarse grid: $x_c \leftarrow 0$;
  Recursively call $A_c, x_c, r_c, l + 1$;
  Prolongate correction: $e \leftarrow Px_c$;
  Update solution: $x \leftarrow x + e$;
  Perform post-smoothing: $x \leftarrow \text{Smoothing}(x, b)$;
end

2.3.1 Adaptive Smoothed Aggregate

Unique to each flavour of MG is the design of its prolongation operator, also called pre-computation since is it done only once and before the main computation. Different prolongation operators lead to different coarse spaces that give different types of coarse corrections.

The Adaptive Multigrid [3] uses the technique called Smoothed Aggregation that groups fine-grid degrees of freedom into domain-decomposed low-mode aggregates, creating a coarse grid problem. It not only utilises geometric coarsening through domain decomposition but also algebraic coarsening by constructing a space with approximate low modes to specifically target GCR's inefficiency for the lower modes in the spectrum. Aggregates are constructed adaptively, taking into account the local behavior of the error in the solution. A full description of the Adaptive Smoothed Aggregation Multigrid method can be found in [4].

In detail, for every level of coarsening, N_e roughly computed lowest eigenvectors ϕ_i are chosen as the basis for the next coarse level. This can be computed using the Arnoldi Algorithm where the inverse operator $M^-_{fine}1$ or an eigenvector solver is applied over and over. Then the domain is geometrically divided into N_{block} tiling sub-blocks (indexed by b) where the global eigenvectors are projected on to produce a set of local eigenvectors $\phi_{i,b}$, $(\phi_{i,b})_j = (\phi_i)_j$ if j lies in the subblock b, $(\phi_{i,b})_j = 0$ otherwise. This decomposition is valid because it is presumed that the sum of $\phi_{i,b}$ over b returns ϕ_i due to a property called local coherence [9] or weak approximation [2][7]. The resultant N_{block} sets of size N_e eigenbasis constitutes the prolongation operator. The global coarse matrix $(N_{block})^2$ of size $(N_e)^2$ local matrix that only acts on a subblock. The coarse matrix is block-sparse which will be discussed later. To construct it, $(m_{coarse})_{b,\hat{b},i,j} = \phi^+_{i,b} M \phi_{j,\hat{b}}$ from the prolongation operator, both self-interaction ($b = \hat{b}$) of the local eigenvectors and neighbour interactions ($b = \hat{b} \pm 1$) between eigenvectors of neighbouring blocks (mirroring the shape of discretised Wilson-Dirac). Once this specific construction of prolongation and coarse matrix is plugged into the multigrid algorithm, the basic adaptive multigrid solver will be functional. For this project, only one level of coarse grid was programmed.

Algorithm 3: Precomputation for two-level MG

Input : Fine matrix M_{fine}

Output: Prolongation Operator P, coarse matrix m_{coarse}

$b \leftarrow$ random vector;

Apply inverse operator to get low mode $\phi_0 \leftarrow M_{fine}^{-1}(b)$;

Remove previous directions **for** $i = 1$ *to* N_e **do**

  $\phi_i = M_{fine}^{-1}(\phi_{i-1})$;

  for $j = 0$ *to* i **do**

    $h_{i,j} \leftarrow \phi_j.dot(\phi_i)$;

    $\phi_i \leftarrow \phi_i - h_{i,j} * \phi_j$;

  end

  $\phi_i \leftarrow \frac{\phi_i}{\|\phi_i\|}$

end

Restrict to each block b, local prolongation operator $P_b = \phi_{ib}, i = 0, ...N_e - 1$;

for $b = 0$ *to* $b = N_{block} - 1$ **do**

  self interaction/diagonal block $(m_{coarse})_{b,b} = P_b^+ M_{fine} P_b$

  neighbour interaction/off-diagonal block $(m_{coarse})_{b,b\pm1} = P_b^+ M_{fine} P_{b\pm1}$

end

The precomputation step tends to be computationally intensive, as is the case here. However, as it is only done once and not iterated over unlike the main body, the high cost of precomputation is generally not considered problematic.

2.3.2 Chirality preservation

Chirality is a symmetry in the 4 spin components of Dirac operators that is lost in the coarse grid construction procedure outlined thus far. It is observed that enforcing the chirality on the coarse matrix can improve the convergence of the adaptive multigrid. To achieve this, the number of low modes ϕ_i is doubled in a way that preserves chirality, by applying chiral projections P_+ and P_-

$$P_+ = \frac{1}{2}(1 - \Gamma_5); P_- = \frac{1}{2}(1 + \Gamma_5) \tag{8}$$

on eigenvectors ϕ_i.

3 Implementation

The code [8] is written in C++ with the usage of object-oriented programming (OOP) and operator overloading for better readability and user-friendliness. It can take QCD matrices in the Matrix Market format, such as those downloaded from SuiteSparse, as input. The program is self-contained with the parser for pre-processing, and the highly tunable GCR and MG algorithms with many options for precondition, restart, eigenvector resolution, etc. Of course, common features such as norm, dot product, getters and setters are also implemented, giving users the opportunity to explore.

3.1 Data: SuiteSparse QCD matrix [5]

SuiteSparse is a web database supplying example sparse matrices of various origins for scientific purposes. The test QCD matrices used in this project are sourced from the QCD group on SuiteSparse.

3.1.1 Investigation of the matrix

Limited information regarding the QCD matrices was given on the SuiteSparse website, including the periodic boundary condition, odd-even ordering, the Dirac representation of gamma matrices, and the critical k for constructing the full Dirac Operator. Therefore, an investigation of the dimensional arrangement was carried out.

Applying the matrix to a vector that is 0 everywhere except when t, x, y, z, spin, colour indices all equal 0, it is observed that the non-zeros in the resultant vector come in consecutive groups of 3. This shows that the last, or slowest running, index is the colour index with 3 elements. As for the spatial dimensions, observing that for each neighbour in spacetime dimension μ, the discretised Wilson-Dirac stencil applies a different γ_μ factor. Assuming the second last dimension is the fermion index of size 4, I rotated the matrix to the eigenbasis of each of the 4 γ matrices, $S^{-1}MS$ where S is the eigenbasis, and checked which one of the directions is diagonalised in each case. Results consistent with the index sequence $(t, z, y, x, spinor, colour)$ was observed.

Theoretically, the boson field U used to construct the Dirac matrix can also be computed using this approach.

3.1.2 Example: Parse

The functions in Parse.h can be used to parse the downloaded sparse matrices in Matrix Market format provided by SuiteSparse. In particular, parse_data processes the Matrix Market data and creates a Compressed Row Storage (CRS) format intermediate file called "parsed.txt" that can be reused later simply by calling read_data.

The following snippet gives an example of how to use the parse functions.

```
# include "Parse.h"

int main() {
    // parses the downloaded mtx data and creates "parsed.txt"
    parse_data("../../data/sample_matrix/conf5_4-8x8-05.mtx");

    // read "parsed.txt" into an instance of Sparse on the memory
    Sparse A(read_data("parsed.txt"));
}
```

3.2 Code Architecture

3.2.1 OOP

There are 3 main classes, *Mesh*, *Field*, and *Operator*. Mesh deals with the conversion between physical indices and location in the memory, which is for the most part hidden from the user; Field is an object

representing the fermion field configuration with 4 spatial, 1 spinor, and 1 colour indices; Operator is an operation that takes Field as input and yields a Field.

Polymorphism is also extensively used. For example, class Operator is the parent class of all operations that takes Field as an argument, including Dirac matrix, GCR solver, and MG solver. They all have a common set of methods, such as get_dim() which returns the dimension of the operator, while each child has different implementations.

```
// an object that acts on a field
template <typename num_type>
class Operator {
public:
    virtual Field<num_type> operator()(const Field<num_type> &) = 0;

    [[nodiscard]] num_type get_dim() const {return dim;};
    virtual void initialise(Operator * op) {};
    [[nodiscard]] virtual std::complex<double> val_at(num_type location) const=0;
    [[nodiscard]] virtual std::complex<double> val_at(num_type row, num_type col) const=0;
    virtual ~Operator()= default;

protected:
    num_type dim = 0;
};
```

3.2.2 Operator Overloading

In Field, operations such as $+$, $-$, and $*$ are overloaded to perform vector addition, subtraction, and scalar multiplication. In Operator, () takes the Field said Operator is operating on. This made the code easier to understand and more intuitive to use. Of course, other commonly used features like norm, getters and setters are also implemented so that users have the freedom to explore.

Below is an example of how the $+$ operator is overloaded in Field class. It takes a second Field object B as the argument to the right of $+$ and returns a third Field object C whose field value is the element sum of the B and self A, i.e. $C = A + B$ as how we normally think of field vector addition.

```
template <typename num_type>
Field<num_type> Field<num_type>::operator+(const Field& f) const {
    assertm(field_size() == f.field_size(), "Lengths of two fields do not match!");

    Field output(mesh);
    for (num_type i=0; i<field_size(); i++) {
        output.mod_val_at(i, field[i] + f.val_at(i));
    }
    return output;
}
```

3.2.3 Example: Field Methods

An example script for how to use Field class methods.

```
#include <complex>
#include "Field.h"
int main() {
    // define 6D fermion fields with dimensions ranked 4-spacetime, spinor, colour
    int ndim = 6;
    int dims[6] = {8, 8, 8, 8, 4, 3}
    Field<long> fermion1(dims, ndim);
```

```
    Field<long> fermion2(dims, ndim);

    //random initialisation with different seeds
    fermion1.init_rand(1);
    fermion2.init_rand(2);

    printf("Number of elements in the fermion field = %d\n", fermion1.get_size());
    std::complex<double> dot_product = fermion1.dot(fermion2);
    printf("Dot product between 2 fermion fields = (%.4e, %.4e)\n",
    dot_product.real(), dot_product.imag());
    Field<long> addition = fermion1 + fermion2;
    printf("Norm of sum of 2 fermions = %.4e\n", addition.norm());
}
```

3.2.4 Example: Operator Methods

An example script for how to make use of the Operator class polymorphism and the overloaded functor.

```
#include "Operator.h"
int main() {
    // read matrix (assumed parsed data exists)
    Operator *Op; // or Sparse *Op;
    Op = new Sparse(read_data("parsed.txt"));

    // define a fermion field
    int dims[6] = {8, 8, 8, 8, 4, 3}
    Field<long> fermion(dims, ndim);
    fermion.init_rand();

    // apply operator to field
    Field result = (*Op)(fermion);
}
```

3.2.5 Example: Solvers and Solver Parameters

An example script for how to use GCR solvers set solver parameters including preconditioning.

```
# include "GCR.h"
# include "MG.h"
# include "Field.h"

int main() {
    // read matrix (assumed parsed data exists)
    Operator *Op; // or Sparse *Op;
    Op = new Sparse(read_data("parsed.txt"));

    // define a fermion field rhs
    int dims[6] = {8, 8, 8, 8, 4, 3}
    Field<long> rhs(dims, ndim);
    rhs.init_rand();

    /* example 1: GCR solver */
    // define GCR parameter: no truncation, 5 steps per restart, max iteration 4000,
    // relative tolerance 1e-13, verbose, no left or right preconditioner
    GCR_Param<long> gcr_param(0, 5, 4000, 1e-13, true, nullptr, nullptr);

    // solve
    GCR gcr(Op, &gcr_param);// gcr is also an Operator approximately Op^-1
```

```
    Field resu1 = gcr(rhs);

    /* example 2: MG preconditioned GCR solver */
    // define MG parameter
    GCR_Param<long> eigen(0, 10, 10, 1e-8, false, nullptr, nullptr);
    GCR_Param<long> coarse(0, 10, 50, 1e-2, false, nullptr, nullptr);
    GCR_Param<long> smooth(0, 10, 0, 1e-8, false, nullptr, nullptr);
    auto solver_c = new GCR(&coarse);
    auto solver_s = new GCR(&smooth);
    // MG setting: mesh to decompose, block size 4^4, 10 eigenvectors, Arnoldi GCR solver,
    // coarse GCR solver, smooth GCR solver, 1 coarse level, no left/right preconditioner
    MG_Param<long> mg_param(mesh, 4, 10, &eigen, solver_c, solver_s, 1, nullptr, nullptr);
    auto mg = new MG(&mg_param);

    // right preconditioned GCR solver
    GCR_Param<long> gcr_param_precond(0, 2, 2000, 1e-13, true, nullptr, mg);
    GCR grc_precond(Op, &gcr_param_precond);
    Field resu2 = grc_precond(rhs);
}
```

3.3 Preservation of Sparsity and Computational Cost

To preserve as much sparsity as possible in the computation, data is stored in the Compressed Row Storage (CRS) format as much as possible. The CRS format contains 3 vectors: the array of the non-zero values in row-major (size = number of non-zero values), the array of column indices of each corresponding value (size = number of non-zero values), and the location in the non-zero value when a new row begins (size = number of rows). For a sparse matrix where the number of non-zero values scales as $O(N)$, the memory required is also only $O(N)$. Additionally, the CRS format is particularly efficient for matrix-vector multiplication thanks to memory locality, which is the most important kernel in GCR.

For this project, both the Dirac Operator and the coarse operator in Multigrid are stored in CRS format, one realised as the class Sparse, the other HierarchicalSparse. Sparse class is simply a CRS wrapper as outlined before with complex number as the value datatype. The coarse matrix accounts for interactions between N_{block} blocks each block represented by a reduced space of dimension of N_e. Since each block only interacts with itself and its eight neighbours, for a large lattice with block number much larger than nine the coarse matrix is still sparse, which is always true in production lattice simulation. To take advantage of this sparsity the HierarchicalSparse class was designed. It represents a $N_{block} \times N_{block}$ sparse collection of $N_e \times N_e$ dense matrix that acts on a local spacetime subblock. Note that due to the periodic boundary contidion, there could be overcounting in self or neighbour interactions if the number of blocks in a certain direction is only one or two. But once that is taken care of, the implemented m_coarse passes the $p^+ m_{coarse} P = M_{fine} P^+ P$ test in machine precision across all blocking schemes, per theory.

The cost of the original GCR solver is $O(V)$ (V being the lattice volume) per iteration due to the sparse matrix-vector multiplication being $O(V)$. There is some loss of sparsity going from the fine Dirac operator to the coarse Dirac operator, since the colour and spinor dimensions of freedom (12) are approximated by $2 \times N_e$ eigenvectors which is usually bigger than 12. However, it can be argued that as long as the number of non-zero values in the coarse matrix, $9 \times N_{block} \times {N_e}^2$, is much smaller than that in the fine matrix $8 \times V \times 12$ (one less neighbour interaction in the t direction), the cost of the extra MG per GCR iteration is negligible.

4 Result

4.1 Tuning the condition number of Dirac Matrices

The sample matrices from the QCD group on SuiteSparse [5] are off-diagonal matrix D that can be used to reconstruct the full Dirac matrix $1 - kD$. D is inherently poorly conditioned, therefore the full Dirac's condition number is related to k the inverse quark mass. Generally, the larger the k, the more ill-conditioned the resultant Dirac.

4.2 Multigrid Result

In this experiment, I solve an $8 \times 8 \times 8 \times 8$ lattice Wilson-Dirac system, i.e. dimension 49152×49152 using pure GCR solver and MG preconditioned GCR solver under the setting $4 \times 4 \times 4 \times 4$ block size, 10 eigenvectors, i.e. coarse matrix dimension 320×320, 2 steps per restart,$1e-8$ or 10 steps of GCR for eigenvector solver, $1e-2$ tolerance or 50 steps of GCR for coarse solver, and smooth solver turned off. Eight k values leading up to the critical $k_c = 0.17865$ are used to test the solver's convergence property. The y direction shows the number of iterations required to converge (residual $< 1e-13$).

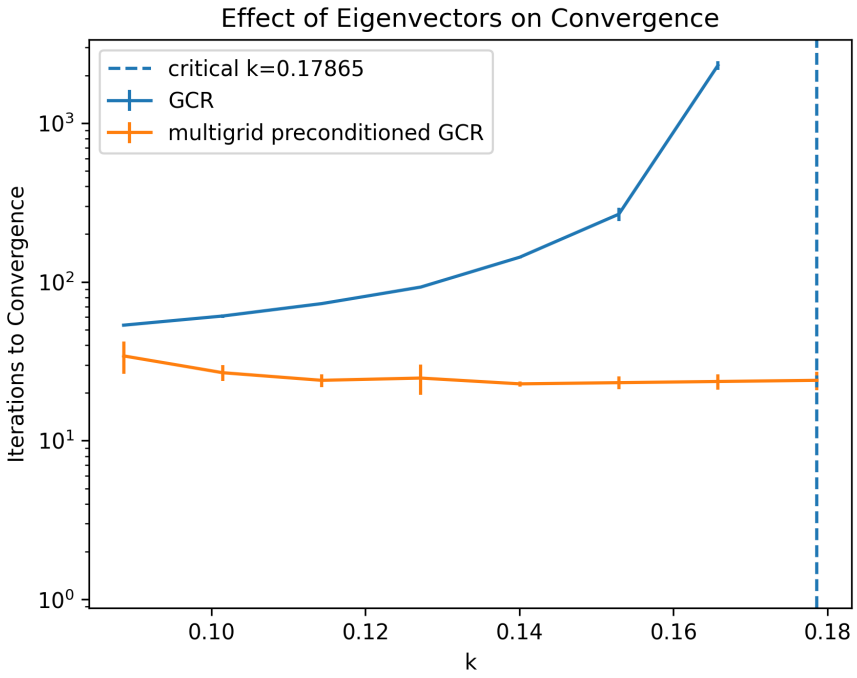

Figure 1: Convergence leading up to critical k

From 4.2, it is apparent that the pure GCR is highly sensitive to the condition number. In fact, the data point at the critical k_c is missing because the GCR solver failed to converge. This is possibly because the convergence rate is so minimally smaller than 1, that numerical errors overrode any convergent behaviour. On the other hand, the convergence of MG preconditioned GCR is consistent regardless of condition number. In this test case, the constant steps to convergence seems to suggest that the solver performance has been decoupled with the condition number.

The convergence analysis in 2 is a lin-log plot showing the residual reduction over iteration at $k = 0.15292$, the second last data point in the previous figure. The GCR line appears linear which is expected as GCR is a linear method that converges exponentially. The MG preconditioned GCR also displays exponential convergence but with a much larger negative gradient. This plot showcases the effect of preconditioning as predicted by theory: it improves the algorithmic condition number and

increases the convergence rate.

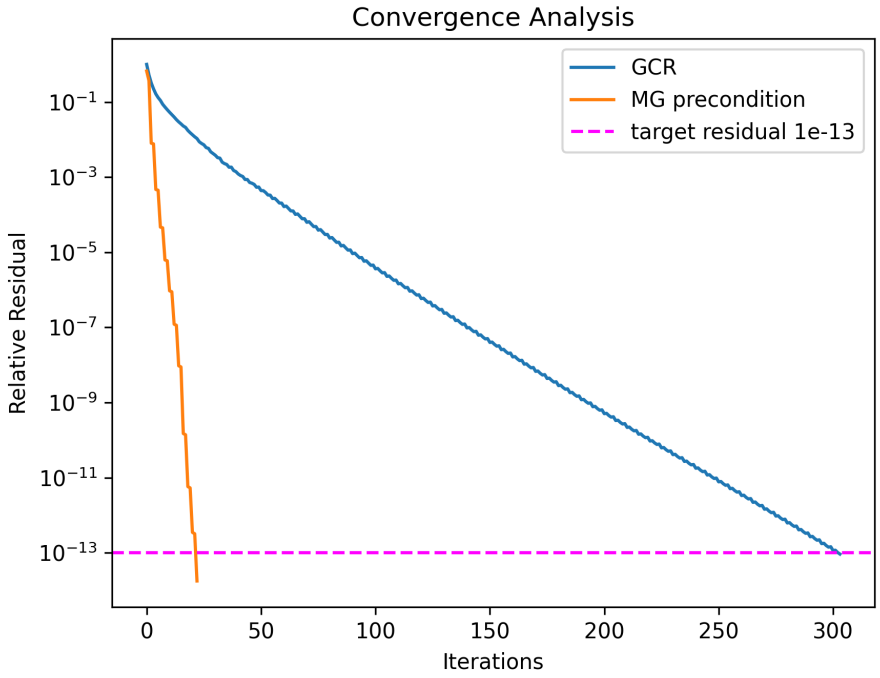

Figure 2: Residual over iteration

5 Conclusion

As seen in the experiment, multigrid preconditioning greatly improves the solver's performance in the critical region. This is scientifically significant because it enables the exploration of physical lattices in simulations. It is a viable strategy to push lattice simulations closer to real experimental setups. The reduction in steps, and in turn compute time, could translate to savings in resources like electricity and manpower at high performance computing centres.

For the next step, alternative Dirac matrices larger than the $8\times8\times8\times8$ ones provided by SuiteSparse should be tested. For larger matrices, it could become beneficial to have more levels of coarsening than the current 1 level. This upgrade can help generalise the MG solver, allowing it to be applied to an even larger range of science and engineering problems. Furthermore, the MG preconditioning could be paired with other popular Krylov solvers for lattice simulations such as GMRES, CG, etc.

References

[1] Computational strategies in lattice qcd, 2009.

[2] R. Babich, J. Brannick, R. C. Brower, M. A. Clark, T. A. Manteuffel, S. F. McCormick, J. C. Osborn, and C. Rebbi. Adaptive multigrid algorithm for the lattice wilson-dirac operator. *Phys. Rev. Lett.*, 105(20):201602, November 2010.

[3] J. Brannick, R. C. Brower, M. A. Clark, J. C. Osborn, and C. Rebbi. Adaptive multigrid algorithm for lattice qcd. *Phys. Rev. Lett.*, 100(4):041601, January 2008.

[4] M. Brezina, R. Falgout, S. MacLachlan, T. Manteuffel, S. McCormick, and J. Ruge. Adaptive smoothed aggregation (αsa) multigrid. *SIAM Journal on Scientific Computing*, 27(3):937–959, 2005.

[5] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection, 2011.

[6] Magnus R. Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6):409, December 1952.

[7] Jun Ping Wang James H. Bramble, Joseph E. Pasciak and Jinchao Xu. Convergence estimates for multigrid algorithms without regularity assumptions. *Math. Comp.*, 57:23–45, 1991.

[8] JingJing Li. Multigrid preconditioned general conjugate residual solver, 2024. https://github.com/jing2li/MGPreconditionedGCR.git.

[9] Martin Lüscher. Solution of the dirac equation in lattice qcd using a domain decomposition method. *Comput.Phys.Commun.*, 156:209–220, 2003.

[10] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. the Society for Industrial and Applied Mathematics, 2003.