UNIVERSITÄT REGENSBURG

Fakultät für Physik
Master of Science Physik

Masterarbeit

zur Erlangung des akademischen Grades eines Master of Science (M.Sc.)

Improving a GMRES-based algorithm for lattice Quantum Chromodynamics

in der Theoretischen Physik.

von

Florian Rappl

(Matrikel-Nr. 136 300 9)

Betreut von:	Dr. Andrea Nobile
Erstgutachter:	Prof. Dr. Tilo Wettig
Zweitgutachter:	Prof. Dr. Gunnar Bali
Einreichung:	11.04.2012

Contents

1. **Introduction** 5

2. **Theory** 7
   - 2.1. Background 8
     - 2.1.1. Inverting matrices 9
     - 2.1.2. Sparse Matrices 10
     - 2.1.3. Hessenberg Matrices 11
     - 2.1.4. Eigenvalue Problems 11
     - 2.1.5. Subspace Iteration 12
     - 2.1.6. Krylov subspace methods 12
   - 2.2. Numerics 14
     - 2.2.1. Conjugate Gradient 14
     - 2.2.2. Arnoldi iteration 16
     - 2.2.3. QR factorization 17
     - 2.2.4. Implicitly Restarted Arnoldi 17
     - 2.2.5. Regular and Harmonic Ritz vectors 19
     - 2.2.6. The Full Orthogonalized Method 20
     - 2.2.7. Preconditioning 22
     - 2.2.8. GMRES 28
   - 2.3. Lattice QCD 36
     - 2.3.1. Continuum QCD 36
     - 2.3.2. Reasons for (lattice) QCD 37
     - 2.3.3. Moving to the Lattice 38
     - 2.3.4. Physical derivation 39
     - 2.3.5. Single flavor case 39
     - 2.3.6. Hybrid Monte Carlo 40

3. **Improving FGMRES-DR** 43
   - 3.1. Development of FFOM-DR 43
     - 3.1.1. The Chroma package 43

3.1.2. Extending Chroma 45
3.1.3. Integrating our inverter 45
3.1.4. High Performance Computing 48
3.2. Major differences to FGMRES-DR 49
3.2.1. General differences 49
3.2.2. Differences for restarting 50
3.2.3. Flexible comparison 51
3.2.4. Savings with FFOM-DR 51
3.2.5. Summary 52
3.3. FFOM-DR 52
3.4. Implementation issues 53
3.5. Further improvements 54
3.5.1. No mixed precision 54
3.5.2. Variation of κ 55
3.5.3. Adjusted deflation 56
3.6. Merging both algorithms to form AOM 61

4. Benchmarks and Tests 65
4.1. Basic evaluation of FFOM-DR 65
4.2. More evaluations of FFOM-DR 69
4.3. Evaluation of each improvement 73
4.4. Evaluation of AOM 76
4.5. Conclusion 78

5. Summary 81

A. Configuration data 85

B. Evaluation data 87

C. Miscellaneous 91

List of Algorithms 93

List of Figures 95

List of Tables 97

Bibliography 99

1. Introduction

Lattice simulations of Quantum Chromo Dynamics have become an important tool to investigate properties of the strong nuclear force. As quantitative first principle approach it influences nearly every aspect of research concerning Quantum Chromo Dynamics. The findings enable scientists to test the standard model with unprecedented precision and to search for new physics. The contributions of Quantum Chromo Dynamics, which appear at high energy particle physics experiments, can be calculated and investigated.

Another reason for the increasing popularity of lattice Quantum Chromo Dynamics is the great computation power that can be used today. Several years ago big clusters or machines with the power to compute even smaller lattices in a reasonable time were either too costly or unavailable. Today we have access to better machines and more computation power for less money.

However, the increasing computation power comes with a price tag and an additional complication. On the one hand we have to stop writing serial programs and twist our mind into the world of writing massively parallel algorithms and programs. On the other hand we have to write more efficient algorithms to fully utilize the computation power of these systems. Modern systems tend to have a very high theoretical peak performance, but it is often very complicated to design applications in a fashion that really fits the machine's concept.

Most simulations tend to have the same problem: A huge linear system has to be solved, e.g. in order to solve the equations of motions. Usually the solver is a quite critical subroutine. This means that in most large-scale scientific and industrial simulations, the majority of the run time is spent in a linear solver. Improving the algorithm for solving the linear system will give great benefits to the overall performance of the simulation. Applications that would benefit from better solvers are being found in all areas from flow models to molecular dynamics to weather forecasting.

Solvers for lattice Quantum Chromo Dynamics have to be very well optimized since the matrices tend to be very large for practical purposes. If we consider a relatively small lattice with 32 sites per spatial direction and 64 sites in the time direction we end up with a matrix that contains 10^{15} entries. Printing out that matrix with a resolution of 1 cm^2 per entry would require 10^5 km^2 of paper. This would be sufficient to cover Iceland as

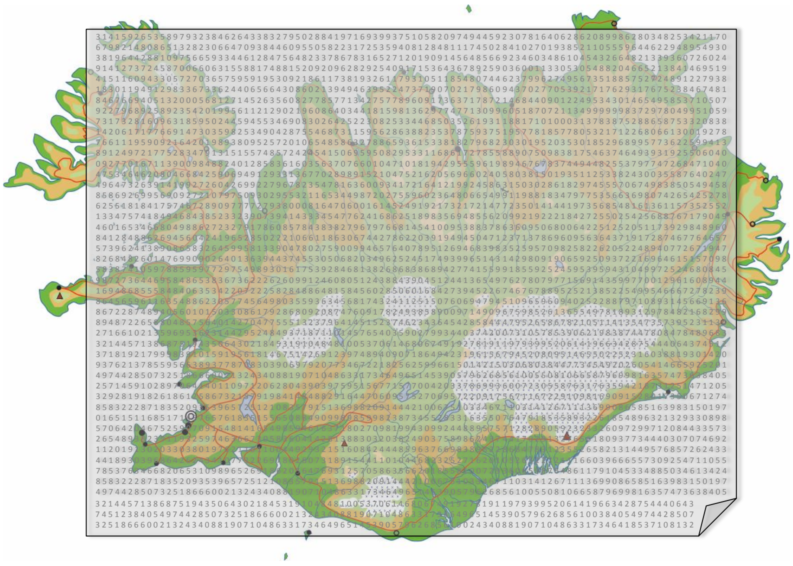


Figure 1.1.: Already small lattices produce matrices which are too big for print.

illustrated in fig. 1.1.

In this thesis we will focus on improving the existing solver algorithm for simulations of Quantum Chromo Dynamics on the lattice with the software package Chroma. We will study various types of solvers. Our initial work will focus on an interesting approach for a new solver that is basically a reduction of the current one. Afterwards we will try to merge the best of both algorithms, i.e. the solver developed by us and the previous implementation, in order to obtain the best performance in every case. We will propose, construct and implement various improvements in the existing algorithm.

Dealing with such huge matrices requires approximate solvers, which work iteratively. The basic concepts and properties of such algorithms will be presented in chapter 2. In chapter 3 we will mainly focus on the modifications to the existing solver called FGMRES-DR. We will discuss the major work as well as minor improvements. One of these improvements is the creation of the Adjusted Deflation algorithm in section 3.5.3. Detailed evaluations can be found in chapter 4, where we investigate the performance of our major algorithm and the amendments due to the minor modifications.

2. Theory

Dealing with large matrices is a common problem in most physical simulations and not only limited to lattice Quantum Chromo Dynamics (QCD). Improving existing algorithms and optimizing iteration count as well as overall time spent on solving a linear system of equations based on those matrices is necessary despite the fact that computation power is still increasing.

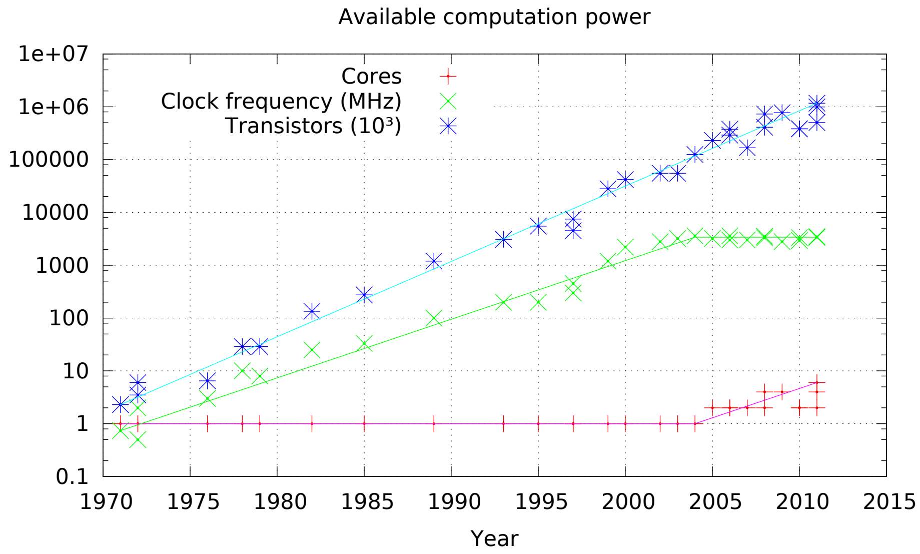


Figure 2.1.: Transistors are nowadays spent on more cores instead of higher frequencies.

However, a shift can be observed lately (Fig. 2.1), since pure computation power on one processor is stagnating due to the fact that the main focus lies on energy efficiency and heat reduction. Due to improvements in the construction process more transistors can be placed on the same space as before. This results in placing more processors on a chip.

We are now at the point where computation cost is less expensive than communication cost, which is a problem we have to deal with while optimizing existing algorithms. So the main focus in optimizing an existing algorithm lies in a more efficient parallel execution of the method as well as less memory consumption.

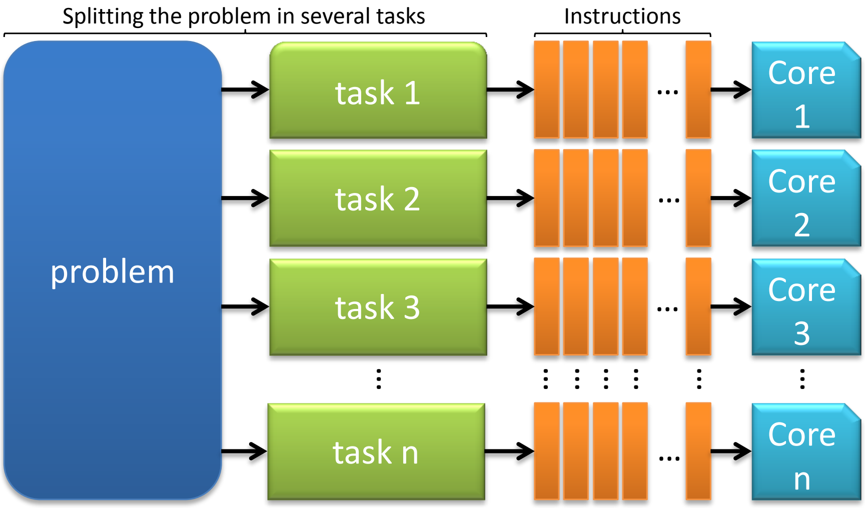


Figure 2.2.: The paradigm of parallel computing is splitting the problem into several tasks.

Fig. 2.2 shows the concept of parallel computing. In parallel solving a problem is split up into parts which can be processed by different CPU cores independently. An efficient parallel performance can therefore be achieved by splitting work into nearly equal parts with little communication required between the cores.

2.1. Background

Solving a system of linear equations is a task that occurs in many fields. Therefore mathematicians have always been interested in solving

$$Ax = b, \tag{2.1}$$

where A is a $n \times n$-dimensional matrix, b being a n-dimensional source vector and x representing the n-dimensional solution vector that has to be found.

One way of solving such a system of linear equations is using the so called Gaussian elimination, where we replace rows by linear combinations of all rows to get an upper triangular matrix of the form

$$A_k = U = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & \cdots & u_{1,n} \\ & u_{2,2} & u_{2,3} & \cdots & u_{2,n} \\ & & \ddots & \ddots & \vdots \\ & & & \ddots & u_{n-1,n} \\ 0 & & & & u_{n,n} \end{bmatrix}_{n\times n}, \tag{2.2}$$

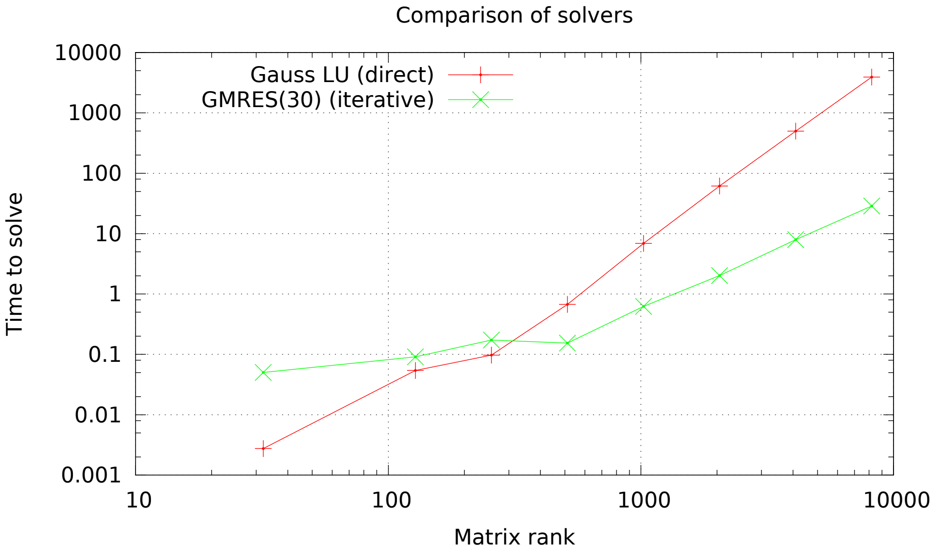


Figure 2.3.: Comparison of a direct (LU) vs an iterative (GMRES with restart) method using MATLAB (for details see appendix C).

where A_k denotes the matrix A after k steps of adding rows. Such *direct methods* will result in an exact result unless the algorithm is unstable. However, the performance of these direct methods scales poorly compared to the second class of solvers. This can be seen in fig. 2.3. Here doubling the matrix rank by a factor of 2 reveals the strong area of this class of solvers, which is very limited. The Gaussian LU decomposition scales with order $\mathcal{O}(n^3)$.

The methods of the second class are called *iterative methods*. These algorithms have a better performance in general, because they do not solve the system exactly. Instead an approximative solution of a desired precision is achieved by performing a sufficient amount of iterations. That means that this class of solvers does the least amount of work in order to solve the system up to the desired precision.

The GMRES method, which is used for the comparison shown in fig. 2.3, scales with order $\mathcal{O}(nm)$, where m is the number of iterations. Additionally a matrix-vector product is required which scales with either $\mathcal{O}(n^2)$ (dense) or $\mathcal{O}(n)$ for sparse matrices.

2.1.1. Inverting matrices

The problem of inverting a matrix is similar to eq. 2.1 since finding the inverse for a fixed matrix A leads to an answer for all systems using the same matrix and a different source vector b. We already know that in the end our approximate solution vector for eq. 2.1 is as close as desired to the exact solution

$$x = A^{-1}b. \tag{2.3}$$

This style of getting the inverse matrix can also be applied in a more general case, where we do not want to compute $A^{-1}b$, but just A^{-1}. In this case we replace b with the n-dimensional identity matrix I, since A is a $n \times n$ matrix. So the problem can be written as

$$AX = I. \tag{2.4}$$

Instead of one equation we now have n equations with n solution vectors which represent our inverse matrix $X = A^{-1}$. Each equation gives us one solution vector x_i and was obtained by solving

$$Ax_i = e_i = \begin{bmatrix} \delta_{1i} \\ \delta_{2i} \\ \vdots \\ \delta_{ni} \end{bmatrix}_{n\times 1}, \tag{2.5}$$

with the Kronecker symbol δ_{ij}. We see that we probably can save some computation time by using a smarter algorithm with knowledge about the previous solution(s). In this case the new source e_{i+1} is orthogonal to the previous source vector e_i.

2.1.2. Sparse Matrices

A matrix is called sparse when it primarily contains zeros. This concept is useful in many fields and also appears when we are trying to solve partial differential equations with numerical methods. Since we can gain performance by saving memory we have to adapt our algorithm in order to support sparse matrices in an optimized way, where we take advantage of the special structure.

One special case of a sparse matrix is a *banded* matrix. This particular case belongs to a special type of sparse matrices called structured sparse matrices. The other interesting type is therefore called unstructured. The first type is identified with a regular pattern, hence the name structured. According to Saad[Saa96] such patterns are found to be small blocks of nonzero elements (dense submatrices) of the same size as well as along a small number of diagonals with nonzero elements.

Again we can use *direct* or *iterative* methods to solve such systems. The latter provides a better performance due to the same reasons as before. Existing methods like the *Conjugate Gradient* (CG) and the *generalized minimal residue* (GMRES) have been proven to be fast, stable and open for optimizations. Therefore we are mainly seeing optimizations and specializations of those algorithms lately.

2.1.3. Hessenberg Matrices

A matrix is called an upper Hessenberg matrix when it only contains zero entries below the first sub-diagonal. A lower Hessenberg matrix just contains zeros above the first super-diagonal. We will only cope with upper Hessenberg matrices H_m that look like

$$H_m = \begin{bmatrix} h_{1,1} & h_{1,2} & h_{1,3} & \dots & h_{1,m} \\ h_{2,1} & h_{2,2} & h_{2,3} & \dots & h_{2,m} \\ & h_{3,2} & h_{3,3} & \dots & h_{3,m} \\ & & h_{4,3} & \ddots & \vdots \\ 0 & & & h_{m-1,m} & h_{m,m} \end{bmatrix}_{m\times m} . \tag{2.6}$$

One property of an upper Hessenberg matrix is that the product of a Hessenberg matrix with a triangular matrix results in another upper Hessenberg matrix. In our algorithm we will have to deal with a Hessenberg matrix that represents a projection of a matrix in another basis.

2.1.4. Eigenvalue Problems

An eigenvalue problem is always defined by the eigenvalue equation,

$$Au = \lambda u, \tag{2.7}$$

where u is a n-dimensional vector and λ is a scalar. As a condition A must be diagonalizable. In this case we say that λ is an eigenvalue of A and u is an eigenvector of A. We call (λ, u) an eigenpair of A. A $n \times n$-dimensional matrix contains at least one eigenvalue and n eigenvalues at most. This can be expressed by writing

$$AV = VD, \qquad V = [u_1, ..., u_n], \quad D = \mathrm{diag}(\lambda_1, ..., \lambda_n). \tag{2.8}$$

Now we rewrite the problem of eq. 2.7 to obtain

$$(A - \lambda I)u = 0, \tag{2.9}$$

with I being the n-dimensional identity matrix. Since u is generally a non-zero vector we see that $A - \lambda I$ must be zero in order to fulfil this equation. In other words we need to have

$$\det(A - \lambda I) = 0. \tag{2.10}$$

The result is a polynomial equation of degree n and is called the characteristic polynomial. The roots of this polynomial equation are given by the eigenvalues of the matrix. Since there is no algebraic formulae for roots of a general polynomial with degree $d > 4$ we need to find the roots numerically. As opposed to solving a linear system of equations this is a case where advancing it iteratively is the only way of solving the problem. Therefore the eigenvalue computations for a general matrix are always done iteratively.

2.1.5. Subspace Iteration

A subspace iteration performs multiplication with the matrix A in order to converge. The original version of an algorithm based on the subspace iteration was invented by Bauer [Bau57]. It starts with an initial system of m vectors forming an $n \times m$ matrix X_0. m is usually chosen to be $m \leq n$, but this is not mandatory. By powering the matrix A we get

$$X_k = A^k X_0. \tag{2.11}$$

This method is called *Treppeniteration.* For normalized column vectors this method will lead to eigenvectors associated with the dominant eigenvalue. Therefore subspace iteration methods are among the simplest for solving large sparse eigenvalue problems. The Treppeniteration can be viewed as a block generalization of the power method by von Mises [vMPG29].

However, Saad [Saa92] mentioned that the system X_k will progressively loose its linear independence. The idea of Bauer's method is to re-establish linear independence for these vectors by a process such as the **LR** or the **QR** factorization. We will use the QR factorization later on in combination with Arnoldi's method.

2.1.6. Krylov subspace methods

A Krylov subspace method is an algorithm for which the subspace is in a special form. The vector space of a Krylov subspace is

$$\mathcal{K}_m(A, v) \equiv \mathrm{span}\{v, Av, A^2v, ..., A^{m-1}v\}. \tag{2.12}$$

We will write Krylov subspaces as $\mathcal{K}_m$. The dimension of the denoted subspace is determined by the index m. The Krylov subspace methods are based on projection processes using $\mathcal{K}_m$. Those processes are orthogonal and oblique onto $\mathcal{K}_m$, which are spanned by vectors of the form $p(A)v$, where p is a polynomial, i.e.

$$p(A) = \rho_1 + \rho_2 A + \rho_3 A^2 + ... + \rho_m A^{m-1} = \sum_{i=1}^{m} \rho_i A^{i-1}. \tag{2.13}$$

Therefore any vector x can be written as $x = p(A)v$, where the degree of the polynomial is not exceeding $m - 1$.

The $n \times m$ matrix that contains all vectors which span the Krylov subspace is called the Krylov matrix K_m. This matrix does have some interesting properties. We know that an arbitrary vector $b \in \mathbb{C}^n$ can be rewritten in the basis of eigenvectors $u_1, ..., u_n$ of A to form

$$b = \sum_{i=1}^{n} c_i u_i, \qquad c_i \in \mathbb{C}. \tag{2.14}$$

Therefore the Krylov matrix K_m could be separated into two parts. On the left side we have a matrix that represents the different vectors of the sum of eq. 2.14 and on the right side we have a matrix containing the n eigenvalues of A with different powers ranging from 0 to $m-1$, i.e.

$$K_m = \begin{bmatrix} c_1 u_1 & \dots & c_n u_n \end{bmatrix}_{n \times n} \times \begin{bmatrix} 1 & \lambda_1 & \dots & \lambda_1^{m-1} \\ 1 & \lambda_2 & \dots & \lambda_2^{m-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \lambda_n & \dots & \lambda_n^{m-1} \end{bmatrix}_{n \times m} . \tag{2.15}$$

By multiplying those two matrices and using eq. 2.7, i.e.

$$\sum_i c_i \underbrace{x_i \lambda_i^{m-1}}_{A x_i \lambda_i^{m-2}} = A^{m-1} \sum_i c_i x_i = A^{m-1} b. \tag{2.16}$$

as well as eq. 2.14 we get the Krylov matrix again. This can be used to get information about eigenvalues. Some other properties of the Krylov subspace (according to Saad [Saa96]) are:

- $\mathcal{K}_m$ is the subspace of all vectors in $\mathbb{C}^n$, i.e. $\mathcal{K}_m \subseteq \mathbb{C}^n$.
- A consequence of the Cayley-Hamilton theorem is that the degree of the minimal polynomial of v does not exceed n. The degree of the minimal polynomial of v is often called the grade of v.
- Let μ be the grade of v. Then $\mathcal{K}_\mu$ is invariant under A and $\mathcal{K}_m = \mathcal{K}_\mu$ for all $m \geq \mu$.
- The Krylov subspace $\mathcal{K}_m$ is of dimension m if and only if the grade μ of v with respect to A is not less than m, i.e.

  $$\dim(\mathcal{K}_m) = m \qquad \leftrightarrow \qquad \text{grade}(v) \geq m. \tag{2.17}$$

- Therefore the dimension of $\mathcal{K}_m$ is the minimum of m and the grade of v.
- Let Q_m be any projector onto $\mathcal{K}_m$ and let A_m be the section of A to $\mathcal{K}_m$ that is, $A_m = Q_m A$.

There are some well-known Krylov subspace methods. One well-known Krylov subspace method we will use extensively is *Arnoldi's method*. Another one is the so called *(hermitian) Lanczos algorithm*. An algorithm using this one is the Conjugate Gradient method. We will discuss this method in chapter 2.2.1.

2.2. Numerics

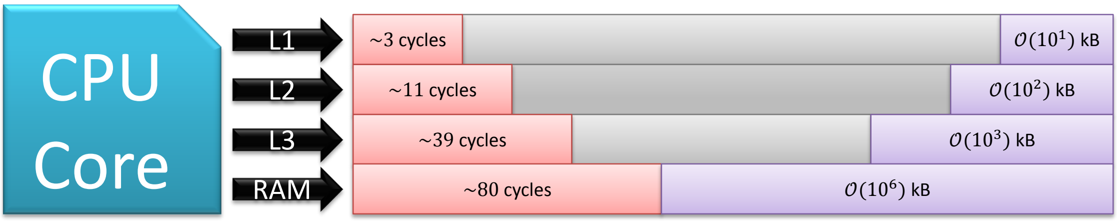


Figure 2.4.: As memory size (purple) increases the access time (red) increases as well.

In order to solve the problem approximately we use a lot of different techniques. However, as we will see our purpose is not only to solve the problem with some defined accuracy in a short time, but also to solve the problem in such a manner that our implementation is *scalable*.

This means that the time required to solve the problem should not grow exponentially with the size of the problem. We want the algorithm to work with a limited amount of memory to avoid long memory access times as displayed in fig. 2.4. As last criterion we want to use algorithms that can easily be split up in (approximately) equal parts to build an efficient parallel code.

2.2.1. Conjugate Gradient

The Conjugate Gradient method (short **CG**) is specialized to solve a system of linear equations for matrices, which have the property of being symmetric and positive-definite. This property is also known as *SPD*. Therefore this method works only for matrices which satisfy

$$A = A^\dagger. \tag{2.18}$$

Additionally to satisfy the positive definite part the matrix A must have only positive eigenvalues. Otherwise $z^\dagger Az > 0$ could not be fulfilled by an arbitrary non-zero vector z. The main idea of the CG method is that solving eq. 2.1 is equivalent to minimizing the quadratic function

$$E(x) = \frac{1}{2}\langle x, Ax\rangle - \langle x, b\rangle, \tag{2.19}$$

with the euclidean scalar product $\langle\cdot,\cdot\rangle$. After picking an initial vector x_0 we have to calculate the start residue r_0. It is easy to see that minimizing the residue is the same as minimizing eq. 2.19, since the gradient at x_k is

$$\nabla\, E(x)\big|_{x=x_k} = Ax_k - b \equiv -r_k. \tag{2.20}$$

Algorithm 1 The Conjugate Gradient Method

```
Input $A \in \mathbb{C}^{n\times n}$, $b \in \mathbb{C}^n$, $x_0 \in \mathbb{C}^n$, $T \in \mathbb{R}_+$
Output $x_k \in \mathbb{C}^n$
  $r_0 := Ax_0 - b$
  $d_0 := r_0$
  $k := 0$
  repeat
    $\alpha_k := \langle d_k, r_k\rangle / \langle d_k, Ad_k\rangle$
    $x_{k+1} := x_k + \alpha_k d_k$
    $r_{k+1} := r_k - \alpha_k A d_k$
    $\beta_k := (\|r_{k+1}\|_2 / \|r_k\|_2)^2$
    $d_{k+1} := r_{k+1} + \beta_k d_k$
    $k := k + 1$
  until $|r_{k+1}| < T$
```

Now we just have to iterate until $\|r_{k+1}\|_2$ is smaller than a set tolerance or until we have reached a defined maximum number of iterations. The complete algorithm is shown in alg. 1.

With some small extensions the CG can be modified to support non-symmetric matrices as well. This algorithm is then called bi-Conjugate Gradient (**BiCG**) and is also based on the *Lanczos algorithm*. The BiCG is rarely used in practice since it is prone to rounding errors and not very stable in general.

One approach to improve the stability of BiCG is the **BiCGstab** method. However even this improved version could not get rid of stability issues. Therefore we will not use this algorithm for our optimizations. CG based lattice QCD codes tend to use the standard CG algorithm with a matrix A that is the product

$$A = M^\dagger M, \qquad (M^\dagger M)x = M^\dagger b = \tilde{b}, \tag{2.21}$$

which is equivalent to solving $Mx = b$. The downside of solving $Ax = \tilde{b}$ is that all eigenvalues of M are squared absolute values in A. This means that the ratio of the largest eigenvalue to the smallest eigenvalue will also be squared, which increases the condition of the problem.

From previous investigations [RD10] we also know that CG based algorithms are likely to have problems with *deflated restarts*. Deflated restarts will be a necessary technique in order to save memory and speed up the computation. We will go into details later on.

Algorithm 2 Arnoldi's method

```
Input $v_1 \in \mathbb{C}^n$ with $\|v_1\|_2 = 1$, $m \in \mathbb{N}$
Output $H_m \in \mathbb{C}^{m \times m}$, $V_m \in \mathbb{C}^{m \times m}$
  for $j := 1 \to m$ do
    $s := 0$
    for $i := 1 \to j$ do
      $h_{ij} := \langle A v_j, v_i \rangle$
      $s := s + h_{ij} v_i$
    end for
    $w_j := A v_j - s$
    $h_{j+1,j} := \|w_j\|_2$
    $v_{j+1} := w_j / h_{j+1,j}$
  end for
```

2.2.2. Arnoldi iteration

Arnoldi [Arn51] proposed an orthogonal projection method onto the Krylov subspace $\mathcal{K}_m$ for general non-hermitian matrices. By using his method we obtain a matrix with the shape of the Hessenberg matrix in eq. 2.6.

This method is very effective and can be used to solve a lot of different problems, e.g. combined with a preconditioner it can also be used to solve non-linear equations [Jas04].

According to Saad [Saa96] Arnoldi also hinted that the eigenvalues of the resulting Hessenberg matrix could provide accurate approximations to some eigenvalues of the original matrix. This is already obvious if we compare the original Arnoldi relation

$$\begin{aligned} AV_m &= V_{m+1} \tilde{H}_m \\ &= V_m H_m + w_m e_m^T \end{aligned} \tag{2.22}$$

to the eigenvalue equation eq. 2.8. Obviously it was later discovered that this strategy leads to an efficient technique for approximating eigenvalues of large sparse matrices.

The basic algorithm is shown in alg. 2. This algorithm leads to the Arnoldi relation of eq. 2.22, which can be rewritten to

$$V_m^\dagger A V_m = H_m. \tag{2.23}$$

Arnoldi's method is still a topic of active research. Saad [SSW98] also uses some improved methods based on the Arnoldi iteration. We will now go into details of one specific optimization called *Implicitly Restarted Arnoldi* (IRA), which was proposed by Sorenson [LS96].

2.2.3. QR factorization

Another important technique in order to use the GMRES method is the so called QR factorization. The method has been introduced by Francis [Fra61, Fra62]. There are several ways to factorize a matrix A into the product QR, i.e. $A = QR$, with $Q^\dagger = Q^{-1}$. For our purposes we will use the so called *Givens rotation* by Givens [Giv58].

The Givens rotation multiplies the original matrix with a rotation matrix, which looks like

$$G_i = \begin{bmatrix} 1 & & & & & 0 \\ & \ddots & & & & \\ & & c_i^* & s_i^* & & \\ & & -s_i & c_i & & \\ & & & & 1 & \\ & & & & & \ddots & \\ 0 & & & & & & 1 \end{bmatrix}. \tag{2.24}$$

In this matrix the complex conjugated cosine and sine occur in the i-th row and the normal cosine and sine are placed in the $i+1$-th row.

After $n-1$ rotations we have computed an upper tridiagonal representative R of our original matrix A. Therefore we have applied the transformation

$$Q^\dagger = G_n G_{n-1} \cdots G_1, \qquad Q^\dagger A = R, \tag{2.25}$$

where $Q^\dagger$ is the product of the Givens matrices G_i. In order to obtain the values for c_i and s_i (and also c_i^* and s_i^* respectively) we only need to compute

$$\beta = \sqrt{|A_{ii}|^2 + |A_{i+1,i}|^2}, \qquad s_i = A_{i+1,i}/\beta, \quad c_i = A_{jj}/\beta. \tag{2.26}$$

For using this technique in the GMRES method we can tweak the algorithm a bit, knowing what certain operations will do. Since we have to execute a QR factorization after an Arnoldi iteration we have to apply all former Givens rotations on the new entries of H. After this required step we are allowed to determine the next rotation and perform this one.

The algorithm for the $j-th$ rotation is shown in alg. 3. The first element γ_0 must equal to r_0. We will see that the resulting element γ_{j+1} can be used in order to determine the residue r_j of $Ax_j - b$ without doing any operation.

2.2.4. Implicitly Restarted Arnoldi

Methods to find eigenvalues are important. They can optimize the convergence of systems and boost the solver methods a lot in performance and efficiency. Saad [Saa92] explains that the boost is gained through a technique called deflated restart.

Algorithm 3 j-th Givens rotation

```
Input $H \in \mathbb{C}^{n \times n}$, $j \in \mathbb{N}_{\not{0}}$, $\gamma_j \in \mathbb{C}$
Output $\gamma_{j+1} \in \mathbb{C}$
  for $i := 1 \to j-1$ do
    $v_1 := h_{ij}$
    $v_2 := h_{i+1,j}$
    $h_{ij} := c_i^* v_1 + s_i^* v_2$
    $h_{i+1,j} := c_i^* v_2 - s_i^* v_1$
  end for
  $\beta := \sqrt{|h_{jj}|^2 + |h_{j+1,j}|^2}$
  $s_j := h_{j+1,j} / \beta$
  $c_j := h_{jj} / \beta$
  $h_{j,j} := \beta$
  $\gamma_{j+1} := -s_j \gamma_j$
  $\gamma_j := c_j^* \gamma_j$
```

The goal of Implicitly Restarted Arnoldi (**IRA**) is to develop a procedure which is equivalent to applying a polynomial filter to the initial vector that is used in the Arnoldi process.

So the idea of IRA contains three main components:

- Polynomial filtering,
- Arnoldi procedure and
- the QR algorithm for computing eigenvalues.

We consider a polynomial filter which is factored like

$$p_q(t) = (t - \theta_1)(t - \theta_2) \dots (t - \theta_q). \tag{2.27}$$

The basic IRA is quite simple in structure and is very closely related to the implicitly shifted QR-Algorithm for dense problems since it uses the q-step shifted QR algorithm as shown in alg. 4.

Generalized eigenvalue problems arise naturally in applications that contain partial differential equations. They have a number of subtleties with respect to numerically stable implementation of spectral transformations. Spectral transformations are also a topic of current research in order to improve the performance of Krylov subspace methods.

This new technique turns out to be a truncated form of the implicitly shifted QR algorithm. Hence implementation issues and final behaviour are closely tied to that well

Algorithm 4 q-step shifted QR

```
Input $H \in \mathbb{C}^{m \times m}$, $q \in \mathbb{N}$, $\theta_i \in \mathbb{C}$ with $i = 1, ..., q$
Output $H \in \mathbb{C}^{m \times m}$, $Q \in \mathbb{C}^{m \times m}$, $R \in \mathbb{C}^{m \times m}$
  $I := \mathbb{R}^{m \times m}$ with $I_{ij} = \delta_{ij}$ for $i, j = 1, ..., m$
  for $j := 1 \to q$ do
    calculate $Q$, $R$ of $(H - \theta_j I)$
    $H := RQ + \theta_j I$
  end for
```

understood method. Due to its reduced storage and computational requirements, the technique is suitable for large scale eigenvalue problems.

Implicit restarting provides a means to approximate a few eigenvalues with user specified properties. In alg. 5 k is the number of eigenvalues sought. We will always obtain the extreme eigenvalues sitting on the surface of the ellipsoid presenting the spectrum of eigenvalues in the Gaussian plane.

2.2.5. Regular and Harmonic Ritz vectors

A concept that is very helpful in finding the eigenpairs with a subspace based algorithm as Arnoldi's method are regular and harmonic Ritz vectors. We will just use the definitions by Sleijpen [SV96]:

Definition If $\mathcal{V}_k$ is a linear subspace of $\mathbb{C}^{n \times n}$ then θ_k is a *Ritz value* of A with respect to $\mathcal{V}_k$ with *Ritz vector* u_k if

$$u_k \in \mathcal{V}_k, u_k \neq 0, \qquad Au_k - \theta_k u_k \perp \mathcal{V}_k. \tag{2.28}$$

Definition A value $\tilde{\theta}_k \in \mathbb{C}$ is a *harmonic Ritz value* of A with respect to some linear subspace $\mathcal{W}_k$ if $\tilde{\theta}_k^{-1}$ is a Ritz value of A^{-1} with respect to $\mathcal{W}_k$.

It can be shown that the definition of the *harmonic Ritz value* is equivalent to

$$\tilde{u}_k \in \mathcal{V}_k, \tilde{u}_k \neq 0, \qquad A\tilde{u}_k - \tilde{\theta}_k \tilde{u}_k \perp A\mathcal{V}_k, \tag{2.29}$$

which looks similar to equation eq 2.28. The proof of eq. 2.29 can be found in the paper of Paige [PPV95]. In general we can say that a Ritz value is an eigenvalue of H_m, which is the resulting upper Hessenberg matrix of Arnoldi's method.

Computing the Ritz values is not as expensive as computing the real eigenvalues since H_m is a matrix of modest size. One property of the Ritz values is that they are usually converging to the largest eigenvalues of A, because H_m is just an orthogonal projection of A into a Krylov subspace that is mainly spanned by powers of A.

Algorithm 5 Implicitly Restarted Arnoldi (IRA)

```
Input $A \in \mathbb{C}^{n\times n}$, $k \in \mathbb{N}$
Output $H_m \in \mathbb{C}^{m\times m}$, $V_m \in \mathbb{C}^{m\times m}$
  for $j := 1 \to m$ do
    Perform step $j$ of Arnoldi
  end for
  We have now obtained $AV_m = V_m H_m + v_{m+1} e_m^T$
  $q := m - k$
  Select the shifts $\theta_1, ..., \theta_q$ from the eigenvalues of $H_m$
  Perform $q$-step shifted QR on $H_m$ and obtain $H_m, Q$
  $H_k := H_m(1:k, 1:k)$
  $V_k := V_k Q$
  $\eta_k := Q_{m,k}$
  $v_{k+1} := v_{k+1} + \eta_k v_{m+1}$
  Set the Arnoldi factorization to $AV_k = V_k H_k + v_{k+1} e_k^T$
  for $j := k \to m-1$ do
    Perform step $j$ of Arnoldi
  end for
```

2.2.6. The Full Orthogonalized Method

Given an initial guess x_0 to the system of linear equations from eq. 2.1, we now consider taking Arnoldi's method in order to build a Krylov subspace in form of eq. 2.12 to obtain

$$\mathcal{K}_m(A, r_0) \equiv \operatorname{span}\{r_0, Ar_0, A^2 r_0, ..., A^{m-1} r_0\}, \tag{2.30}$$

in which $r_0 = Ax_0 - b$. This method seeks an approximate solution x_m from the affine subspace $x_0 + \mathcal{K}_m$ of dimension m. So we set $v_1 = r_0/\|r_0\|_2$ in Arnoldi's method and compute $\beta = \|r_0\|_2$ in order to get the known Arnoldi relation eq. 2.23. As a result the approximate solution vector x_m is given by

$$x_m = x_0 + V_m y_m, \qquad y_m = H_m^{-1}(\beta e_1). \tag{2.31}$$

The residual vector of the approximate solution x_m computed by the Full Orthogonalized Method (**FOM**) is now given by

$$\begin{aligned} b - Ax_m &\overset{(2.31)}{=} b - A(x_0 + V_m y_m) && (2.32)\\ &= r_0 - AV_m y_m && (2.33)\\ &\overset{(2.22)}{=} \beta v_1 - V_m H_m y_m - h_{m+1,m} e_m^T y_m v_{m+1}. && (2.34)\end{aligned}$$

Algorithm 6 Full Orthogonalized Method (FOM)

```
Input $A \in \mathbb{C}^{n \times n}$, $b \in \mathbb{C}^n$, $x_0 \in \mathbb{C}^n$, $T \in \mathbb{R}_+$, $m \in \mathbb{N}$
Output $x \in \mathbb{C}^n$
  $x := x_0$    $r_0 := b - Ax_0$    $h := \mathbb{C}^{m \times m}$
  $v_1 := r_0/\beta$
  $\beta := \|r_0\|_2$
  repeat
    Perform step $j$ of Arnoldi
    if $h_{j+1,j} = 0$ then
      Stop iterating (Singularity detected)
    end if
    Perform $j$-th QR rotation to minimize $\|h_m y - \gamma\|_2$
    if $|h_{j+1,j} \cdot \gamma_j / h_{j,j}| < T$ then
      Stop iterating (Convergence occurred)
    end if
    $j := j + 1$
  until $j + 1 > m$
  for $i := j, j-1, j-2, ..., 1$ do
    $y_i = (\gamma_i - \sum_{k=i+1}^{j} h_{i,k} y_k)/h_{i,i}$
  end for
  for $i := 1 \to j$ do
    $x := x + y_i v_i$
  end for
```

This can be simplified by using $H_m y_m = \gamma$ with $\gamma = \beta e_1$. We can easily see that $\beta v_1 - V_m H_m y_m = 0$. Therefore we finally obtain

$$\|b - Ax_m\|_2 = h_{m+1,m}|e_m^T y_m| = \frac{h_{m+1,m}\gamma_m}{h_{m,m}}, \tag{2.35}$$

where γ_m is the m-th element of γ. The residue is determined by the source of a least squares problem, which is also target of Givens rotations.

The algorithm shown in alg. 6 shows two breaking conditions. First we want to stop iterating if we encounter a singularity in Arnoldi's method. This is an obvious choice because $H_{j+1,j} = 0$ results in a direct solution and unstable algorithm. The second criterion is stronger, because the first one is already a subgroup of the second one. Here we calculate the residue according to eq. 2.35 and compare it to the set tolerance.

We see that the first condition is actually a subgroup since eq. 2.35 would be zero for $H_{j+1,j} = 0$, which is always smaller than a set tolerance $T > 0$. The reason for explicitly asking to look for both cases lies in the Givens rotation between the two conditions. We can save computation power by omitting this computation in case that $H_{j+1,j}$ is already zero.

The basic FOM works as expected, but does have some drawbacks. One technique to accelerate the convergence is to use preconditioning. This is also used in order to prevent stagnation and other decelerating properties of some matrices.

2.2.7. Preconditioning

Preconditioners can significantly accelerate convergence. Therefore an analysis of the code is usually recommended in order to determine if a preconditioner should be included. Another important reason to include a preconditioner is that a proper method can also increase stability.

2.2.7.1. General

With a preconditioner we mean a matrix M, which transforms our given problem into a form that is more suitable for the algorithm. Also such a matrix has to reduce the condition number of the matrix that will be used to solve the problem. Since the condition number is optimized with the help of a preconditioner we can expect to obtain the solution in less iterations.

Instead of solving the original linear system of eq. 2.1 we do now solve the (optimized) system

$$AMy = b. \tag{2.36}$$

Such a system is called *right preconditioned* system. In the end the solution x can be obtained by using

$$M^{-1}x = y, \qquad x = My. \tag{2.37}$$

As long as M is non-singular we will get the same solution as with the original equation. A bit more common than this method is the so called *left preconditioned* system. It is almost the same except that we multiply A with M from the left side in contrast to the right side as in eq. 2.36. The new equation reads

$$MAx = Mb, \qquad \tilde{A}x = \tilde{b}. \tag{2.38}$$

In this case our solution x of the modified equation eq. 2.38 is also the solution of the original equation eq. 2.1.

Typically there is a trade-off in the choice of M. Since the preconditioner must be applied at each step of the iterative linear solver, it should have a small cost of applying the operation to determine MA or AM. The cheapest preconditioner would therefore be $M = I$.

However, this would not change the computation cost. The most expensive preconditioner would be $M = A^{-1}$. Here we have the problem that the preconditioner is as expensive as the solution.

As a result we have to find a good compromise that saves us iterations on one side and does consume less time than the saved iterations would have required on the other side. The preconditioner $M = A^{-1}$ has the (optimal) *condition number* of 1, requiring a single iteration for convergence. This means that a good preconditioning matrix is also a good approximation to A^{-1}.

There are many possible preconditioning techniques. Among the more known preconditioners are the following methods:

- **Jacobi**, which is one of the simplest forms. Here we set

  $$M^{-1} = \mathrm{diag}(A). \tag{2.39}$$

  Assuming that $A_{ii} \neq 0 \; \forall i$ we obtain

  $$M_{ij} = \frac{\delta_{ij}}{A_{ii}}. \tag{2.40}$$

- Successive Over-Relaxation (short **SOR**) is a method that decomposes a matrix A into its diagonal D and its lower triangular part L. It can be parametrized using a real coefficient ω to determine

  $$M^{-1} = \frac{1}{\omega} D + L, \tag{2.41}$$

  which closely related to the Jacobi preconditioner in eq. 2.39.

- Symmetric SOR or **SSOR** is a technique that has also been used for lattice QCD previously [FFG+96]. It works for hermitian matrices A that can be decomposed as

  $$A = \mathrm{diag}(A) + A_{i<j;j=1,\ldots,n} + A_{i>j;j=1,\ldots,n} = D + L + L^{\dagger}. \tag{2.42}$$

  After having split up A into its diagonal, lower and upper triangular part, the SSOR matrix is defined as

  $$M^{-1} = (D + L) D^{-1} (D + L)^{\dagger}. \tag{2.43}$$

  It can be shown that this is a much better approximation to the inverse. Eq. 2.43 can also be parametrized as eq. 2.41 using some parameter ω to regulate the dominance of D. Additionally we need a factor $(2 - \omega)^{-1}$.

- The **Approximate Inverse** method uses a banded approximation to the inverse of A. The solution is obtained by truncating the matrix to some bandwidth.
- **ILU** (Incomplete LU factorization) , is a sparse approximation of the so-called LU factorization.
- The **Schwarz Alternating procedure** is quite common in lattice QCD and is also the preconditioner of our choice. It can also be used as an appropriate solver [Lüs04b]. Details will be explained in section 2.2.7.3.

We will now focus on a special type of preconditioners called Domain decomposition methods.

2.2.7.2. Domain decomposition methods

Definition Given a function $f : X \to Y$, the set X is the domain of f. The set Y is the co-domain of f. In the expression $f(x)$, x is the argument and $f(x)$ is the value.

A typical problem occurs when we want to find a good parallel algorithm for solving an elliptical partial differential equation (**PDE**). Such problems always deal with finding a good way of splitting a specific domain as shown in fig. 2.5 into smaller parts. Also the emergence of multi-cores and their theoretical potential has led to an enforced research in domain decomposition methods [SBG04].

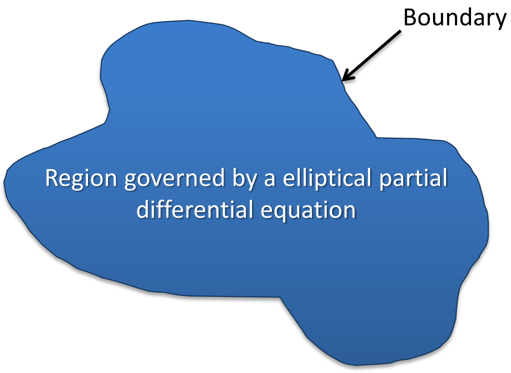

Boundary
Region governed by a elliptical partial differential equation

Figure 2.5.: Solving a PDE on a certain region with a boundary, a so called domain.

Domain decomposition methods (**DDM**) solve a boundary value problem by splitting a domain into smaller boundary value problems on so called sub-domains as shown in fig. 2.6. In order to solve the original problem iterations are required. Those iterations coordinate the solution between the created adjacent sub-domains.

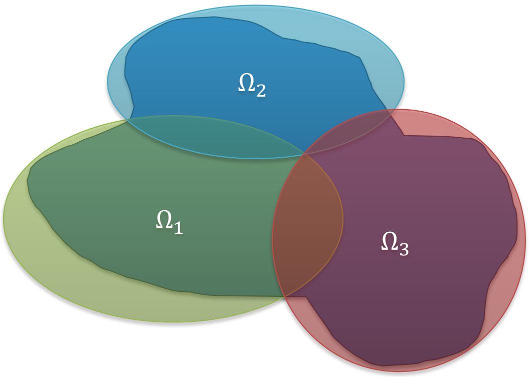


Figure 2.6.: Finding a proper domain decomposition into equal Ω_i is a difficult task.

A problem with some unknowns per sub-domain is used to further coordinate the solution between the sub-domains globally. The problems on the sub-domains are treated as independent. This means that no further communication between the sub-domains is required, which makes DDM suitable for parallel computing. Using such methods as preconditioners for our problem gives us the advantage of an easy to use parallel algorithm while increasing the condition number of our original problem.

Also DDM seems to be very suitable as a preconditioner for lattice QCD as experiments suggest [MS10]. Usually two approaches of DDM are differentiated: the non-overlapping and the overlapping methods. We are only interested in the last one. In overlapping DDM, the sub-domains overlap by more than the interface. Those methods include the Schwarz Alternating Procedure, which will be discussed in the next subsection. The following definition might be helpful when considering domain decomposition for linear systems:

Definition A sub-domain of X is any domain Ω where $\Omega \subset X$. The sub-domain of a matrix is any sub-matrix.

2.2.7.3. Schwarz Alternating Procedure

Schwarz [Sch70] proposed an iterative method for solving a boundary value problem as described in the previous section. We are considering a bounded, open domain Ω that is smooth and connected and can be decomposed in two sub-domains Ω_1 and Ω_2 such that

$$\Omega = \Omega_1 \cup \Omega_2. \tag{2.44}$$

The Schwarz Alternating Procedure (**SAP**) now tries to find the solution of a PDE on Ω by solving the equation on Ω_1 and Ω_2 separately. The boundary condition is always set by the latest values of the approximate solution.

There are two practical implementations of the SAP:

- The additive Schwarz method adds the results of the sub-domains. Therefore it can be viewed as an overlapping Jacobi preconditioner.
- The multiplicative Schwarz method also uses the product as a third term. This prevents an embarrassingly parallel algorithm. However, it also features a better approximation.

Therefore the multiplicative Schwarz Procedure does solve

$$(A_1 + A_2 - A_2A_1)x = b, \tag{2.45}$$

where we want to find x for some right hand side b with matrices A_i containing the action of A on the domain Ω_i. The additive version does not contain such a mixed term, which makes it more suitable for parallel computing. Here the corresponding equation looks like

$$(A_1 + A_2)x = b. \tag{2.46}$$

Using the multiplicative Schwarz method for lattice QCD has been popular for some time due to various efficient implementations [Lüs03]. Sometimes SAP is taken as a starting point and modified to form new methods. One popular implementation of such modifications are known as Multigrid [OBB⁺10] methods. The basic principle of these implementations is to compute some regions independently, thus saving a lot of communication and simplifying the problem.

However, since we use the multiplicative version we also have a mixed term which has to be calculated from the independent solutions. Even though this requires more communication, it is necessary in order to provide a better accuracy and therefore in a better conditioned matrix. The additional communication overhead is still acceptable since communication and computation of the solver is even more expensive.

Following the paper of Martin Lüscher [Lüs04a] the SAP starts by a factorization of the quark determinant, i.e. we rewrite the (massive) Wilson–Dirac operator $D \equiv D_w + m_0$ in the block form

$$D = \begin{bmatrix} D_\Omega & D_{\partial\Omega} \\ D_{\partial\Omega^*} & D_{\Omega^*} \end{bmatrix}. \tag{2.47}$$

Now the determinant can be rewritten as

$$\det D = \det D_\Omega \det D_{\Omega^*} \det\left(1 - D_\Omega^{-1} D_{\partial\Omega} D_{\Omega^*}^{-1} D_{\partial\Omega^*}\right). \tag{2.48}$$

The argument of the last determinant is referred to as the preconditioned Dirac operator. The form of eq. 2.47 is the Schur complement of the Dirac operator with respect to the block decomposition. The next section will explain the Schur decomposition in greater detail.

The blocks have to obey some restrictions. One restriction is that the lattice can be split up in domains Λ which satisfy

$$D_{\Omega} + D_{\Omega^*} = \sum_{\Lambda} D_{\Lambda}. \tag{2.49}$$

Furthermore we can build up a checkerboard out of all subdomains. Here all black domains sum up to D_{Ω}, while all white domains sum up to D_{Ω^*}. This leads to the interesting identity

$$\det D_{\Omega} \det D_{\Omega^*} = \prod_{\Lambda} \det \tilde{D}_{\Lambda}. \tag{2.50}$$

$\tilde{D}_{\Lambda}$ is the Dirac operator with even–odd preconditioning on the block Λ with Dirichlet boundary conditions. The classical Schwarz procedure obtains the solution of the Wilson–Dirac equation in an iterative process, where all black and all white blocks are visited alternately. The equation is then solved with Dirichlet boundary values given by the current approximation to the solution. Since the lattice Dirac operator involves only nearest-neighbor hopping terms, the equations on the black and white blocks are completely decoupled from each other and can be solved in parallel.

2.2.7.4. Schur decomposition

If there is an even number of such Schwarz blocks in all dimensions we are able to apply even-odd preconditioning in order to accelerate the convergence even more. Such an even-odd preconditioning will decrease communication time as well as computation time since A can be split up in sub-matrices like

$$A \rightarrow \begin{bmatrix} A_{oo} & A_{oe} \\ A_{eo} & A_{ee} \end{bmatrix} \equiv \begin{bmatrix} T & B \\ C & X \end{bmatrix}. \tag{2.51}$$

We are now able to rewrite this decomposition to a really handy form, which can help us in our task of solving a linear system. Basically eq. 2.51 now reads

$$A = \begin{bmatrix} 1 & BX^{-1} \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} T - BX^{-1}C & 0 \\ 0 & X \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 \\ X^{-1}C & 1 \end{bmatrix}. \tag{2.52}$$

We can calculate the inverse of the matrix A in dependence of the sub-matrices now using the known equation for inverting a 2 × 2-matrix resulting in

$$A^{-1} = \begin{bmatrix} 1 & 0 \\ -X^{-1}C & 1 \end{bmatrix} \cdot \begin{bmatrix} (T - BX^{-1}C)^{-1} & 0 \\ 0 & X^{-1} \end{bmatrix} \cdot \begin{bmatrix} 1 & -BX^{-1} \\ 0 & 1 \end{bmatrix}. \tag{2.53}$$

If the sub-matrix X is cheap to invert we will get some benefit from using this method. Usually we want to capture the low eigenmodes in X. If this can be realized the computed sub-matrix $A - BX^{-1}C$ will be better conditioned and as a result the used solver will normally work better. The implementation of a Schur decomposition in lattice QCD is quite common and gives performance boosts for the most configurations [RD10].

2.2.8. GMRES

The Generalized Minimum Residual Method (GMRES) is a projection method introduced by Saad [SS86]. The algorithm is based on taking the m-th Krylov subspace $\mathcal{K}_m$ and applying the matrix A on it. In order to orthogonalize the different Krylov subspaces against each other we need a proper method. The GMRES algorithm uses the well-known Arnoldi iteration to orthogonalize the subspaces.

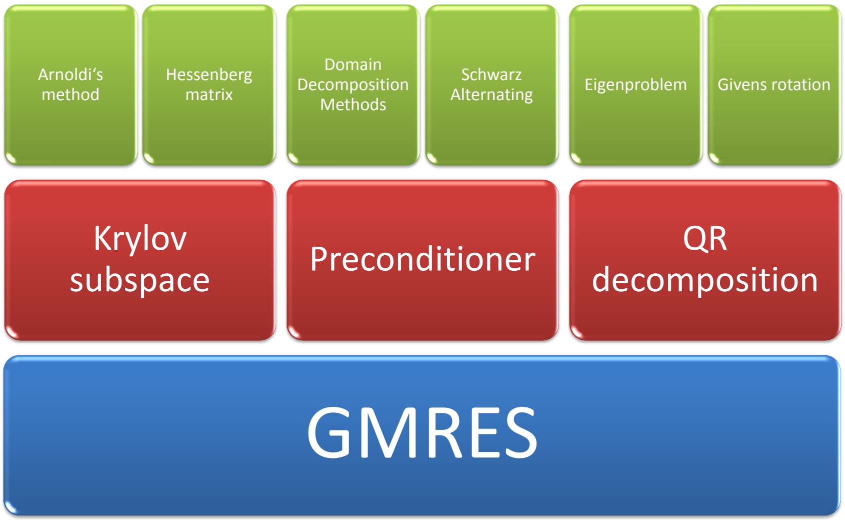


Figure 2.7.: The GMRES algorithm uses a lot of different techniques.

As we see in fig. 2.7 GMRES touches a lot of different fields and algorithms in order to solve a system like eq. 2.1 in an efficient way. Each sub-field is topic of current research and contains plenty of possible specializations and optimizations. We will focus in optimizing FOM. Therefore we have to look at optimized variations of a quite similar algorithm, which is in this case GMRES. The used algorithms like SAP, QR and others will be taken for granted and will not be part of the investigation.

2.2.8.1. Comparison between FOM and GMRES

The matrix H_m from eq. 2.6 could additionally contain another row forming an $m+1 \times m$-dimensional matrix instead of an $m \times m$ block- matrix. The extended matrix is called $\tilde{H}_m$ and is responsible for all differences and equalities between FOM and GMRES.

The residue is much easier to calculate using GMRES. In FOM we had to use eq. 2.35 to determine the residue for the current number of iterations. In GMRES we just have to use the last entry of the γ-vector, i.e. γ_{m+1} since our least squares problem also expanded from m to $m+1$ entries.

Saad [Saa96] did also work out other useful relations between FOM and GMRES. The exact relation between the residue of FOM ϱ^F and GMRES ϱ^G is given by

$$\varrho_m^F = \sqrt{1+\left(\frac{h_{m+1,m}^m}{h_{m,m}^{m-1}}\right)^2}\varrho_m^G, \tag{2.54}$$

where m is the number of current iterations. This was first shown by Brown [Bro91]. To go a step further Cullum and Greenbaum [CG96] did examine eq. 2.54 more carefully. They obtained

$$\begin{aligned} \left(\varrho_m^G\right)^{-2} &= \sum_{i=0}^{m}\left(\varrho_i^F\right)^{-2} && (2.55)\\ &\leq \frac{m+1}{(\varrho_{m^*}^F)^2}, && (2.56)\end{aligned}$$

in which $\varrho_{m^*}^F$ is the smallest residual norm achieved in the first m steps of FOM. This imposes severe constraints on the residue of FOM. The residue of FOM cannot be lower than the residue of GMRES and will not be higher than $\sqrt{m}\varrho_m^G$ [Saa96].

2.2.8.2. The method

The original GMRES method begins with the basic Arnoldi relation as in eq. 2.22. Here $\tilde{H}_m$ represents a matrix that is quite similar to H_m, i.e. eq. 2.6, except that instead of an $m \times m$ block-matrix we have an $m+1 \times m$-dimensional matrix. Therefore $\tilde{H}_m$ looks like

$$\tilde{H}_m = \begin{bmatrix} h_{1,1} & h_{1,2} & h_{1,3} & \dots & h_{1,m} \\ h_{2,1} & h_{2,2} & h_{2,3} & \dots & h_{2,m} \\ & h_{3,2} & h_{3,3} & \dots & h_{3,m} \\ & & h_{4,3} & \ddots & \vdots \\ & & & \ddots & h_{m,m} \\ 0 & & & & h_{m+1,m} \end{bmatrix}_{m+1\times m}. \tag{2.57}$$

Since we can rewrite any vector x which lies in $x_0 + \mathcal{K}_m$ to

$$x = x_0 + V_m y, \tag{2.58}$$

where y is an m-vector, we can modify our original equation for the residue. By doing this we obtain

$$\begin{aligned} b - Ax &= b - A(x_0 + V_m y) = r_0 - AV_m y = && (2.59)\\ &\overset{(2.22)}{=} \beta v_1 - V_{m+1}\tilde{H}_m y = V_{m+1}(\beta e_1 - \tilde{H}_m y). && (2.60)\end{aligned}$$

Algorithm 7 Basic GMRES method

```
Input  $A \in \mathbb{C}^{n\times n}$, $b \in \mathbb{C}^n$, $x_0 \in \mathbb{C}^n$, $m \in \mathbb{N}$, $T \in \mathbb{R}_+$
Output  $x \in \mathbb{C}^n$
  $j := 1$    $x := x_0$    $r_0 := b - Ax_0$    $\tilde{h} := \mathbb{C}^{m+1\times m}$
  $\beta := \|r_0\|_2$
  $v_1 := r_0/\beta$
  repeat
    Perform step $j$ of Arnoldi
    if $\tilde{h}_{j+1,j} = 0$ then
      Stop iterating (Singularity detected)
    end if
    Apply all former Givens rotations to new elements
    Perform $j$-th rotation to minimize $\|\tilde{h}_m y - \gamma\|_2$
    if $|\gamma_{j+1}| < T$ then
      Stop iterating (Convergence occurred)
    end if
    $j := j + 1$
  until $j > m$
  Build solution:
  for $i := j, j-1, j-2, ..., 1$ do
    $y_i = (\gamma_i - \sum_{k=i+1}^{j} \tilde{h}_{i,k} y_k)/\tilde{h}_{i,i}$
  end for
  for $i := 1 \to j$ do
    $x := x + y_i v_i$
  end for
```

Here y is a vector that minimizes the function $\|\beta e_1 - \tilde{H}_m y\|_2$. This minimizer is inexpensive to compute since it's just the solution of an $(m+1) \times m$ least-squares problem with a (usually) small m.

In Arnoldi's method (alg. 2) we perform all m iterations. In the GMRES method we will only compute the j-th Arnoldi iteration (for the j-th cycle). Each GMRES cycle will only contain the computation of one Arnoldi step. For example in the third GMRES cycle we will execute Arnoldi's method with $j = 2$.

The algorithm should stop if the computation encounters a singularity in the implementation of Arnoldi's method or if the residue r_j that is retrieved by performing the Givens rotation is smaller than the desired level of tolerance T.

Taking γ_{j+1} as the residue in alg. 7 is motivated by the fact that we have a $n+1 \times n$

Hessenberg matrix. Since we can only have a diagonal $n \times n$ block-matrix leaving the last element $h_{j+1,j}$, which influences the residue. The old residue γ_j is then multiplied with $h_{j+1,j}/\beta \geq 1$ which results in γ_{j+1}.

One thing that will be precious is memory. For large matrices we have a lot of vectors with many values to be saved. Since most memory consumption is coming from the Krylov subspace we are building up, we need a method to accomplish the same result without requiring such a big subspace. A very simple approach to this problem will be explained now.

2.2.8.3. GMRES(k)

The GMRES(k) method is sometimes known as GMRES with restarts. It does not build up a big Krylov subspace with m elements but instead builds up a rather small Krylov subspace with $k \ll m$ vectors. So the GMRES(k) limits the size of the subspace to k elements, i.e. we are not using $\mathcal{K}_m$ but $\mathcal{K}_k$.

After the maximum size of the subspace is reached, while the residue is still not smaller than the set tolerance, we compute the current approximate result and start over again. Therefore we are working with a smaller subspace but a more accurate one since we replace $A^n x_0$ by $A^n x_i$ for the i-th restart.

The algorithm of GMRES(k) is quite similar to alg. 7. Instead of taking the maximum subspace size of m we take k with $k < m$. We then wrap our basic GMRES algorithm in another loop computing the problem and looking for convergence.

In alg. 8 we stop after a certain amount of iterations or if convergence occurred. In practical implementations we see that this approach is heavily dependent on k. If we set k too low we will end up with too many iterations in our GMRES(k) loop. If we set k too high we face the other possible problem of ending up with too many iterations in our original GMRES algorithm.

Therefore there is a lot of research going on in order to determine the best number for k [ZN05]. Another option is to accelerate the convergence using a so called acceleration technique. Those methods attempt to copy the convergence of full GMRES more closely. So by using such a technique we would end up with the same convergence (no gain) using a much smaller subspace (probably huge gain) than the original GMRES.

One possible acceleration technique is a block variant of GMRES [BDJ06]. An implementation of this algorithm works with or without the use of a preconditioner. Furthermore the algorithm is quite easy to implement. However, it could not been taken as a suitable preconditioner nor does it prevent stagnation more effectively than GMRES(k).

Instead the original GMRES(k) could be used as a preconditioner. This has been

Algorithm 8 GMRES(k) method

```
Input $A \in \mathbb{C}^{n \times n}$, $b \in \mathbb{C}^n$, $x_0 \in \mathbb{C}^n$, $T \in \mathbb{R}_+$, $m \in \mathbb{N}$, $k \in \mathbb{N}$
Output $x \in \mathbb{C}^n$
  $j := 1$
  repeat
    Compute $x$ from basic GMRES using ($A$, $b$, $x_0$, $k$ and $T$)
    if GMRES did succeed with $|\gamma_{j+1}| < T$ then
      Stop iterating (Convergence occurred)
    end if
    $x_0 := x$
    $j := j + 1$
  until $j > m$
```

proposed by Kharchenko [KY95]. Another interesting possibility is the recycling of existing Krylov subspaces. Already computed Krylov subspace base vectors could be reused in order to save computation time. This is an excellent way if the source vector b changes only slightly. One existing algorithm is **GCR** [PSM+06].

Another interesting possibility is to focus on the restart process. In alg. 8 we did trash everything we computed in the inner loop, i.e. in alg. 7. An approach is to use the Hessenberg matrix again for improving not only the solution x but also the matrix A. We will now take a look at the details of such an algorithm.

2.2.8.4. GMRES-DR

A restart algorithm as proposed by Morgan [Mor02] does solve some problems that come up with ordinary restarted GMRES methods. A common problem of ordinary GMRES(k) is that the method could start stagnating at a large residual norm. The method built by Morgan uses a thick restarting technique, which was proposed by Wu and Simon [WS00].

Obviously in order to do a deflated restart we have to determine what we want to deflate. It is well known that the convergence of GMRES usually depends quite heavily on the distribution on eigenvalues. Therefore removing the eigenvectors corresponding to the small eigenvalues (i.e. damping those small eigenmodes) can greatly improve the convergence process.

Basically there are several ways of obtaining this deflation. Once a Krylov subspace grows big enough deflation occurs automatically. However, restarted methods usually do not develop such a big Krylov subspace. In our case we will use the properties of harmonic Ritz pairs as described in section 2.2.5. So we are doing deflation with the harmonic Ritz

Algorithm 9 GMRES with deflated restarts

```
Input $A \in \mathbb{C}^{n\times n}$, $b \in \mathbb{C}^n$, $x_0 \in \mathbb{C}^n$, $T \in \mathbb{R}_+$, $m \in \mathbb{N}$, $k \in \mathbb{N}$, $q \in \mathbb{N}$
Output $x_k \in \mathbb{C}^n$
  Apply GMRES with $A$, $b$, $x_0$, $T$, $k$ to obtain $x_k$, $V_{k+1}$ and $\tilde{H}_k$
  $j := 1$
  repeat
    $\beta := h_{k+1,k}$
    $x_0 := x_k$
    $r_0 := b - Ax_k$
    Compute the $q$ smallest eigenpairs $(\tilde{\theta}_i, \tilde{g}_i)$ of $H_k + \beta^2 H_k^T e_k e_k^T$
    Orthonormalize the $q$ $\tilde{g}_i$'s (separate into real and imaginary parts)
    $P_q := \mathbb{C}^{k\times q}$ containing the orthonormalized $\tilde{g}_i$
    Extend $P_q$ with an empty row resulting in $P_q \in \mathbb{C}^{k+1\times q}$
    Orthonormalize the vector $\gamma - \tilde{H}_k y$ against them
    Store resulting vector in next column of $P_q$ called $p_{q+1}$
    $\tilde{H}_q := P_{q+1}^T \tilde{H}_k P_q$
    $V_{q+1} := V_{k+1} P_{q+1}$
    Reorthogonalize $v_{q+1}$ against $v_i$ with $i = 0, ..., q$
    Apply Arnoldi to form $V_{k+1}$ and $\tilde{H}_k$
    $\gamma := V_{k+1}^T r_0$
    Solve $\min \|\gamma - \tilde{H}_k y\|_2$ for $y$
    $x_k := x_0 + V_k y$
    $r := b - Ax_k$
    $j := j + 1$
  until $\|r\|_2 < T$ or $j > m$
```

vectors.

In order to compute the eigenvectors (i.e. harmonic Ritz vectors) the Implicitly Restarted Arnoldi (alg. 5) can be used. For solving the system we use again a QR method in form of Givens rotations. The deflation techniques are already quite popular in lattice QCD [MW02, Lüs07a].

This is mostly because in lattice QCD the eigenvalues of our system have a big range with the smallest one messing up the condition of the matrix. Deflating those small eigenvalues is a crucial step in improving the condition of the operator and has been a topic of active research in the last years [Lüs07b]. Several ways to optimize the convergence even more can be found in the literature.

One example is “Loose” GMRES or short LGMRES [BJM05]. This algorithm is basically a combination of several other algorithms that try to speed up convergence of standard GMRES. What prevents us from using LGMRES is mainly that the algorithm does not solve the stagnation problem of standard GMRES(k). This is why we have to improve GMRES even further. We will now discuss the possibility of not only preconditioning the matrix once but for every Arnoldi iteration.

2.2.8.5. FGMRES

The flexible GMRES (**FGMRES**) is based on the same algorithm as alg. 7. The difference lies in the Arnoldi process. Instead of the standard Arnoldi iteration shown in alg. 2 FGMRES uses a flexible version, which is a technique called flexible preconditioning.

Flexible preconditioning means that each Arnoldi step uses a different preconditioning matrix M denoted M_j for simplifying the problem. Now the Arnoldi relation from eq. 2.22 does not hold any more. Instead of the usual relation we will use

$$AZ_m = V_{m+1}\tilde{H}_m. \tag{2.61}$$

Obviously each vector z_j in Z_m is represented by $z_j = M_j v_j$. The idea of applying a preconditioning matrix M_j is that the computation will give us better approximate solution vectors. These vectors will accelerate the convergence.

FGMRES uses the flexible Arnoldi process as shown in alg. 10. In the non-flexible case, where we have $M_j = M \ \forall j$, we see that eq. 2.61 becomes eq. 2.22 by substituting A with AM. Therefore the Krylov subspace is now built up on different vectors than before, resulting in a different search space than before.

Overall with the right preconditioning algorithm the search space is hugely optimized resulting in far less iterations than the non-flexible GMRES version. However, in order to integrate further improvements it is necessary to combine the flexible Arnoldi process as shown in alg. 10 with deflated restarts, which is outlined in alg. 9.

2.2.8.6. FGMRES-DR

Flexible GMRES with deflated restarting is the combination of the algorithms described in sections 2.2.8.4 and 2.2.8.5. Here the trick lies in the connection between the two algorithms. So the question is how to implement the flexible Arnoldi as outlined in alg. 10 in a GMRES with deflated restarts as described in alg. 9.

A possible implementation was shown by Nobile [NZF]. In this implementation other tricks are used as well. One possible enhancement is the usage of a so called iterative refinement as shown in alg. 11. The advantage that can be obtained by using this algorithm

Algorithm 10 Flexible Arnoldi process

```
**Input** $A \in \mathbb{C}^{n\times n}$, $b \in \mathbb{C}^n$, $m \in \mathbb{N}$
**Output** $V_{m+1} \in \mathbb{C}^{n\times m+1}$, $Z_m \in \mathbb{C}^{n\times m}$, $H \in \mathbb{C}^{m+1\times m}$
  $\beta := \|b\|_2$
  $v_1 := b/\beta$
  for $j := 1 \to m$ do
    Obtain preconditioner $M_j$ for new basis vector
    $z_j := M_j v_j$
    $w := A z_j$
    for $i := 1 \to j$ do
      $h_{ij} := w^\dagger v_i$
      $w := w - h_{ij} v_i$
    end for
    $h_{j+1,j} := \|w\|_2$
    $v_{j+1} := w/h_{j+1,j}$
  end for
```

Algorithm 11 Iterative refinement

```
**Input** $A \in \mathbb{C}^{n\times n}$, $b \in \mathbb{C}^n$, $x_0 \in \mathbb{C}^n$
**Output** $x^h \in \mathbb{C}^n$
  $r_0^h := b - A x_0$
  $r_0^l := C(r_0^h)$, i.e. convert from high to low precision
  Solve $A x^l = r_0^l$
  $x^h := x_0 + C(x^l)$, i.e. convert from low to high precision
```

is to use less memory (and therefore less computation power) in order to obtain the same result. This is done by using a lower precision arithmetic for some parts of the algorithm.

One problem of such a combination is that the relation in eq. 2.61 does contain a usually small inconsistency. Due to rounding errors and summation an amplification of the error can occur resulting in instabilities or even divergence. Therefore it is important to develop an economic and efficient method in order to check the sanity of the flexible Arnoldi relation.

The implementation of Nobile does contain such a check that proceeds with a *clean* restart, discarding all eigenvectors, as in the classical iterative refinement technique. This is achieved by comparing the implicit norm of the residue with the explicit one, computed after each cycle. The ratio of the two norms together with a set threshold form the condition if the method should be restarted.

2.3. Lattice QCD

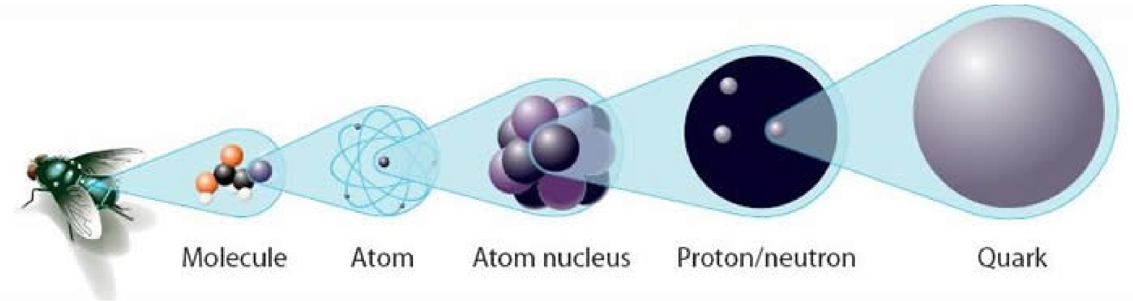


Figure 2.8.: As far as we know today quarks are the smallest building blocks of matter (taken from the Nobel Price in Physics 2008 press release [oS08]).

Quantum Chromo Dynamics is the theory of the strong force, i.e. Quarks interacting over exchange particles called gluons. Quarks form structures like Protons and Neutrons as shown in fig. 2.8. Lattice QCD is a way to determine some physical observables of QCD from *first principles* via simulations [Mag11].

2.3.1. Continuum QCD

The success of Quantum Electro Dynamics (**QED**), which is based on the assumption of a local gauge symmetry, suggested to promote the same principle to a theory of the strong interaction. While QED builds up on the $U(1)$ symmetry group, $SU(3)$ turned out to represent the known particle spectrum for QCD. The postulation of color charge, carried by the quarks inside hadrons, was necessary to explain the existence of some baryons. Otherwise the Pauli exclusion principle would have been violated.

All formulations in this section will be in Euclidean spacetime, since it is the most suitable for lattice calculations. Describing both quarks and gluons, the QCD action is build up from a fermionic and a bosonic (gauge) part, i.e.

$$S_{\text{QCD}} = S_{\text{fermionic}} + S_{\text{gauge}}. \tag{2.62}$$

The strongly interacting quarks are described by Dirac spinors usually denoted with $\psi^{f}_{\alpha,c}(x)$. They depend on the spacetime position x and do contain 3 indices: one for the flavor f (between 1 and N_f), as well as one for the spin α (between 1 and 4) and one for the color c between 1 and 3. So called antiquarks are introduced by conjugating the representation of quarks.

We can build up a theory consisting of quarks and antiquarks only. This would give us a static theory without dynamic properties. Therefore a dynamic theory requires a kinetic term, which is introduced with a so called gauge field $A^a_\mu(x)$. Here the color index a runs from 1 to 8, because the gauge field is in the adjoint representation of the color group, which is eight-dimensional. μ represents the Lorentz index (between 1 and 4).

The action is then constructed by taking several constraints into consideration. Those constraints are motivated by the gauge invariance that is requested from the theory. In the end the fermionic part of the action reads

$$S_{\text{fermionic}} = \sum_f \int d^4x \overline{\psi}^f(x) \gamma_\mu D^f_m(x) \psi^f(x), \tag{2.63}$$

where $D_m(x)$ is the Dirac operator for a mass m. By using the covariant derivative $D_\mu(x) = \partial_\mu + iA_\mu(x)$ we have

$$D_m(x) = \gamma_\mu D_\mu(x) + m. \tag{2.64}$$

The gauge action is also similar to the one of QED. In QCD we have

$$S_{\text{gauge}} = -\frac{1}{2g^2} \int d^4x \operatorname{Tr}\left\{F_{\mu\nu}(x) F_{\mu\nu}(x)\right\}. \tag{2.65}$$

Here g is the coupling constant. The field strength tensor $F_{\mu\nu}(x) = -i[D_\mu(x), D_\nu(x)]$ is a generalized formulation of the one used in QED, where the potential A_μ represented the photon field. The explicit version of the commutator is given by

$$F_{\mu\nu} = \Big(\partial_\mu A^a_\nu(x) - \partial_\nu A^a_\mu(x) - f^{abc} A^b_\mu(x) A^c_\nu(x)\Big) t^a. \tag{2.66}$$

Here t^a are the generators of the $SU(3)$ group given by $[t^a, t^b] = if^{abc}t^c$. The f^{abc} are called structure constants. We rarely need the explicit form for them, which is a scalar value of either 1, 1/2, $\pm\sqrt{3}/2$ or 0 depending on the indices.

2.3.2. Reasons for (lattice) QCD

Actually, no single approach to solve QCD is applicable to the entire energy range of interest. Perturbative methods become unfeasible at small momentum transfers, since the magnitude of the QCD coupling constant increases with the inverse momentum (that is equivalent to increasing distance). Non-perturbative methods like lattice QCD can treat strong interactions at all energy scales up to some cut-off.

As one of many examples lattice QCD studies can directly address the questions whether hadronic bound states survive the transition to high temperatures or at which temperatures they dissociate. Also the question if the hadronic low temperature states are replaced by some loosely bound or “quasi-bound” states can be obtained.

The importance of these processes is emphasized by the Standard Model of particle physics. QCD effects are crucial for the interpretation of measurements at heavy-ion collisions. In fact, only with the understanding of those effects it is possible to determine otherwise unquantifiable background contributions to discoveries of physics beyond the Standard Model at experiments. Such experiments are usually conducted in state of the art particle colliders like the LHC in Switzerland.

2.3.3. Moving to the Lattice

For handling QCD on the lattice we have to discretize space-time and the action of continuum QCD to fit onto a finite lattice with periodic boundary conditions. The quarks are located on the lattice sites as fermion fields ψ while the gluon fields sit on the links between the lattice sites as gauge links U.

The evaluation in continuum theory involves the computation of an infinite four-dimensional integral over all possible field configurations. This is called a Feynman path integral. Since such integrals can not be computed numerically we have to use different techniques in order to calculate expectation values of observables. Lattice QCD uses Monte Carlo based algorithms which contain techniques like importance sampling in order to solve such integrals.

There are some major advantages that favor a discrete version of QCD on the lattice over the continuum theory:

- Lattice QCD is naturally regularized by the lattice, i.e. it does not contain the infinities encountered in continuum theory.

- Non-perturbative results from lattice QCD can give answers to physics questions where perturbative calculations fail.

- Results are easier to compare to the experiment in order to identify signs of effects from new physics.

However, lattice QCD does have some disadvantages as well:

- The discretization is not unique, i.e. there are several lattice versions which are equivalent in the continuum limes.

- There are symmetries of the continuum theory which cannot be represented completely like chiral symmetry.

- We are only able to calculate on a finite volume instead of an infinite one. Therefore we will encounter some errors.

In order to stay close to reality we have to use a small enough lattice spacing a with a big enough lattice volume. The lattice volume is determined by n_x, n_y, n_z space sites and n_t time sites. The total number of sites is therefore calculated over

$$N = n_x n_y n_z n_t. \tag{2.67}$$

2.3.4. Physical derivation

The most important observable is the energy E_n of an energy eigenstate $|n\rangle$ of a Hilbert space with a well defined set of quantum numbers [GL09]. On the lattice we are able to measure Euclidean correlation functions

$$\langle \hat{O}_2(t)\hat{O}_1(0)\rangle_T = \frac{1}{Z_T}\int D[\psi(x,t)]\exp(-S[\psi(x,t)])O_2[\psi(x,t)]O_1[\Phi(x,0)], \tag{2.68}$$

where Z_T denotes the partition function and S the action. The partition function is defined as

$$Z_T = \int D[\psi(x,t)]\exp(-S[\psi(x,t)]), \tag{2.69}$$

with the integration measure over all possible "paths" ψ, i.e.

$$D[\psi(x,t)] = \prod_n d\psi_n(x,t). \tag{2.70}$$

In this form eq. 2.68 is not computable. This path integral form was obtained by extracting the energies E_n of the correlators in the spectral representation of those, i.e.

$$\lim_{T\to\infty}\langle \hat{O}_2(t)\hat{O}_1(0)\rangle_T = \sum_n \underbrace{\langle 0|\hat{O}_2(t)|n\rangle}_{M_n}\langle n|\hat{O}_1(0)|0\rangle\exp(-tE_n). \tag{2.71}$$

Here M_n are the matrix elements of the operators. The state with the given quantum numbers and the lowest energy E_0 is the easiest to measure, as for large Euclidean times t the sum in eq. 2.71 is proportional to $\exp(-tE_0)$.

The *path* denoted with $\psi(x,t)$ in the context of lattice QCD are configurations of gauge fields U between lattice sites. Even in the case which includes dynamical fermions one can integrate out those degrees of freedom coming from fermions. The remnant of this integration is a so called "fermionic determinant", which, being a determinant of a very large matrix, is very expensive to evaluate numerically.

2.3.5. Single flavor case

Consider the case of a single flavor of fermions where the partition function, which has been described in Thomas DeGrand's book [DD06], reads

$$Z = \int [dU][d\overline{\psi}][d\psi]\exp[-S_G(U) - \overline{\psi}M(U)\psi], \tag{2.72}$$

where $M = D + m$ and $S_G(U)$ denotes the gauge field action. The variables ψ and $\overline{\psi}$ are so-called Grassmann variables.

Definition A Grassmann number θ_i anti-commutes with every other Grassmann number θ_j and commutes with ordinary numbers x, i.e.

$$\theta_i \theta_j = -\theta_j \theta_i, \qquad \theta_i x = x \theta_i. \tag{2.73}$$

So θ_i^2 is zero in order to obey this relation.

After integration over the fermionic Grassmann variables ψ and $\overline{\psi}$ the partition function looks like

$$Z = \int [dU] \exp[-S_G(U)] \det M(U). \tag{2.74}$$

The determinant in the integral expression is non-local. So we see that most cost of computing the expression lies in computing the change of the determinant under a change of the gauge field. However it is possible to introduce a so called *pseudo-fermion* field Φ, which is a scalar and represents a color-triplet. Now we can use the formal identity

$$\det M(U) = \int [d\Phi^* d\Phi] \exp[-\Phi^* M^{-1} \Phi]. \tag{2.75}$$

In this form the action is computable. In practice, it is generally too difficult to find a *paired* operator that sums only over the paired eigenmodes and that can be constructed without knowledge of the eigenvectors. The solution involves an explicit doubling with corrections to come later. Because of γ_5-hermiticity we have $\det D = \det D^\dagger$ and the determinant for two flavors is $\det D^2 = \det D^\dagger D$, which is just what we aimed for. Our pseudo-fermion action has become

$$S_{\text{eff}} = S_G(U) + \Phi^* [M(U)^\dagger M(U)]^{-1} \Phi = S_G(U) + \Phi^* X. \tag{2.76}$$

To get X we have to solve the large sparse linear system

$$(M^\dagger M) X = \Phi. \tag{2.77}$$

Since M depends on the gauge field we *must* solve this linear system at each step in the gauge field molecular dynamics integration. For this reason, including fermions in the path integration adds considerably to the computational cost and therefore effective algorithms for solving eq. 2.77 are required.

2.3.6. Hybrid Monte Carlo

With improved actions (clover term etc.), more specialized algorithms and more available computation power lattice QCD is becoming more and more easy to handle. One breakthrough was certainly the invention of the Hybrid Monte Carlo (**HMC**) algorithm [SSS86]. This algorithm reduces the parallel cost of executing a lattice QCD simulation due to the highly non-local action. The exact details are described in the introductory paper [DKPR87].

HMC produces configurations in the Markov chain which are more independent than in purely random Monte Carlo procedures. This is achieved by integrating up a set of classical equations of motion with the help of a trajectory over a certain time, usually chosen to be 1. The final state of the system is a proposal for a new configuration. This new configuration is accepted or rejected with the help of the Metropolis algorithm. This algorithm therefore satisfies *detailed balance*, which is required for our ergodic Markov chain of configurations.

Other demands on the algorithm are the conservation of the integration measure as well as reversibility. The integration is done by a leapfrog algorithm with a certain step size ϵ. All in all we have N integration steps for the position with $\epsilon = 1/N$ if we set the time to 1. For the momenta we have $N + 1$ integration steps, where the first and the last step have a different step size with $\epsilon/2$.

For QCD we perform alternate updates of the sites and the gauge links. By generating a complex vector χ with distribution $\exp(-\chi^\dagger \chi)$ we are able to create momenta which represent the different sites. The potential is created with the help of a molecular dynamics trajectory that has the sites as a fixed external field.

3. Improving FGMRES-DR

In order to improve currently used algorithms for solving linear systems we have to compare different implementations and find criteria for their usage. A particularly interesting comparison is a direct evaluation of GMRES versus FOM.

Both algorithms are very similar. However, it seems that FOM sometimes performs slightly better than GMRES and vice versa. We will implement a flexible version of FOM with deflated restarts in order to do a number of evaluations and find a proper condition for selecting the superior algorithm depending on the problem. This new version of FOM will be called **FFOM-DR**.

3.1. Development of FFOM-DR

FFOM-DR is quite similar to FGMRES-DR. However, there are some differences that have to be observed and present main characteristics of each algorithm. The differences will be discussed in the next section 3.2. In this section we will focus on the development basis and the similarities.

3.1.1. The Chroma package

Lattice QCD codes are usually quite long and tied up with huge libraries for specialized mathematics[1] and physics[2]. Instead of writing a complete code for simulating QCD from scratch we will use an existing code base and rewrite specific methods.

Luckily a very extensible code basis already exists with **Chroma**. On the project's website [Col11] the creators [EJ05] write that

> "The Chroma package supports data-parallel programming constructs for lattice field theory and in particular lattice QCD. It uses the SciDAC QDP++ data-parallel programming (in C++) that presents a single high-level code image to the user, but can generate highly optimized code for many architectural systems including single node workstations, multi-threaded SMP workstations

---

[1] e.g. linear algebra and complex numbers

[2] e.g. γ-matrices and handling of spinors

> (soon to come), clusters of workstations via QMP, and classic vector computers."

The Chroma package is a collection of applications for lattice QCD and does support data-parallel programming constructs for lattice gauge theories in general. One advantage of Chroma is that it uses *SciDAC QDP++ data-parallel programming* which is written in C++. QDP++ itself uses other packages like QIO for data parallel IO, QMT for multi-threading or QMP for parallel communications. We can use the included subroutines. The different layers of the Chroma architecture are shown in fig. 3.1.

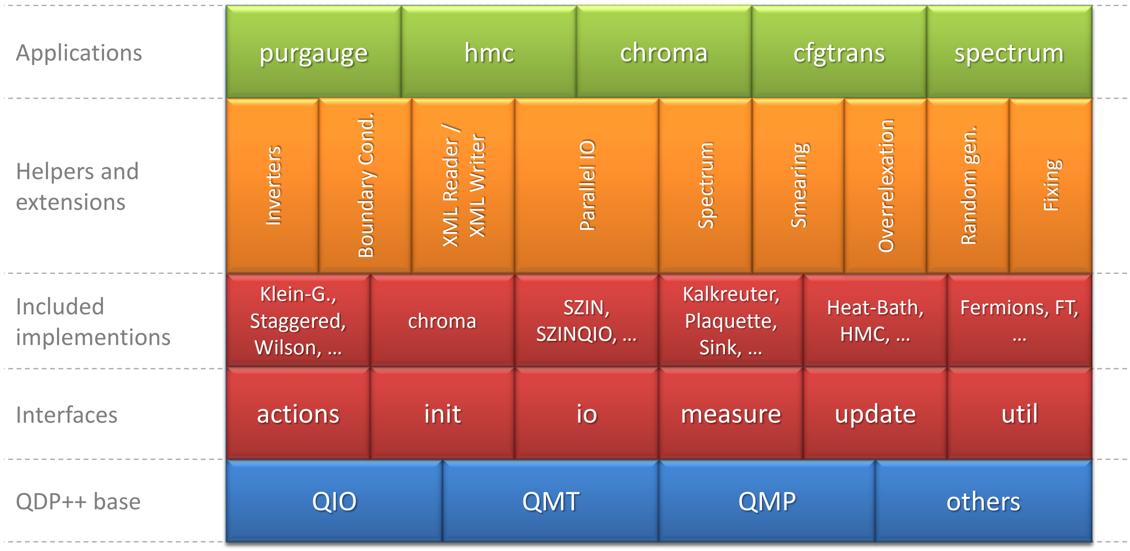


Figure 3.1.: The main layers of the Chroma software package including some implementation examples.

This gives us a broad basis, which can be used, altered and extended very easily. Inverters like CG, BiCGSTAB and GMRES are already included. This accelerates the development process and allows us to run comparisons between different algorithms. The level of abstraction is such that user code using the interfaces can run unchanged on a single processor or multiprocessors with parallel communications.

An advantage of using Chroma is that it basically compiles on any target machine, i.e. PCs, clusters and supercomputers. Even better, depending on the target machine, several optimization packages can be considered. These packages offer better performance on some targets like SSE2 based machines, IBM BlueGene and several others. However, we want to extend Chroma beyond the standard capabilities in order to measure possible performance improvements as well as to test new algorithms and different ways of simulating lattice gauge theories.

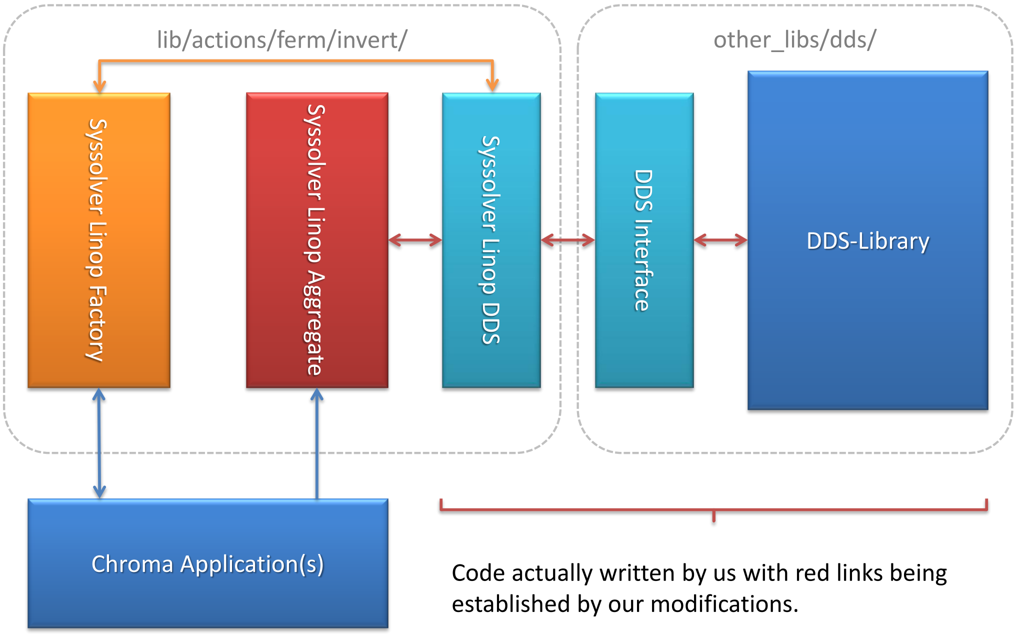


Figure 3.2.: Our approach of extending Chroma with another inverter.

3.1.2. Extending Chroma

In order to extend Chroma we have to change the main Makefile[3] that is contained in the basic package. We will include a list of C/C++ header and source files located in a subdirectory that is placed in the *other_libs* directory of Chroma. This is not a requirement, but a strong recommendation - following the guided design patterns will certainly help to extend Chroma. We will also have to add our interfaces to the specific hash-map in Chroma.

Basically our application could compile on its own into object code. Because we did not write any main function we've basically just written a library that can be used. Additionally we can modify and include calls to our library in the original Chroma code. In order to achieve this we need an interface class that can be called from Chroma and will communicate with our own classes. This interface links Chroma with our own library and is responsible for calling the required functions and giving back the necessary values.

3.1.3. Integrating our inverter

In our case we want to extend the list of available inverters. Therefore we modify the available factory-class in the *actions/ferm/invert* directory by adding calls to the interface file that we have created to link Chroma and our library together. Chroma uses that

---

[3] A Makefile is used in order to create a set of compilation rules and state compilation dependencies

interface class to talk to our library in a (for inverters) standardized way. This specific way is described by a template pattern, i.e. an abstract upper class that has to be inherited in order to implement methods that will be known and can be called. This is a property of static programming, since methods always need to know the exact type or one of the inherited types of the instance it is talking to. So all important functions have to be known from a related class that the interface actually uses as a first construction plan.

The basic outline for extending Chroma is displayed in fig. 3.2. Essentially the following steps are required:

1. We create our library that is completely independent of Chroma. We write a file with an entry point (a *main()*-function) to test our library.

2. We outsource all required parameters to a single structure. This is important to keep the code clean and to distinguish between inverter parameters and other parameters.

3. Once our library seems to work we write a class that should function as a connector class between Chroma and our package. In our case this interface class is called *dds_interface*. It contains one important function that will be called from Chroma, delivering the *struct* we just added before. With the important parameters delivered, we can call the necessary methods in order to invert the matrix.

4. Another file we have to write is placed in the Chroma inverter package. In our case we call this file *syssolver_linop_dds*. This file contains references to the *dds_interface* class and knows how to talk to our library. It also talks to Chroma, so it will receive all parameters from the Chroma system. One important task of this file is to convert the important parameters to the structure we've just written. After the structure is filled we can pass it in the list of arguments to our entry point in *dds_interface*.

5. Since Chroma has to talk to our class in *syssolver_linop_dds* we need some paradigms from object-oriented-programming. Here we have to consider inheritance in order to give our class some functionality and for Chroma to treat our class with this basic functionality. The class we have to inherit from is called *LinOpSystemSolver*$\langle T \rangle$.

6. We have to tell Chroma that we've written a lot of new files and added a new library. This is done in the file *syssolver_linop_aggregate*. Here we tell Chroma to register our own solver in its factory-system. Therefore Chroma will call the register method that we've provided in the file *syssolver_linop_dds*, which adds a reference in the hash-map of *syssolver_linop_factory*. We do this the following way:

```
#ifdef COMPILE_DDS
success &= LinOpSysSolverDDSEnv::registerAll();
#endif
```

This calls the register function in a namespace that we've created. It will do the registration that is described next. We use the condition because we still want to be able to compile without DDS. In this case we do not want any references to our library (that will not be included in the build) in Chroma as this would lead to errors. The boolean variable is used together with a bit operation (binary and) in order to state if all registrations have been executed successfully. If any of those registrations fail then Chroma will not execute.

7. The registration is done by calling a method in the *TheLinOpFermSystemSolverFactory* namespace located in the Chroma namespace. The code reads:

```
Chroma::TheLinOpFermSystemSolverFactory::Instance().registerObject(name,
    createFerm)
```

Here we use the static method *Instance()* to call *registerObject()* with two arguments. The first one is the keyword to use for our solver. The second one is a reference to a method for creating an instance of our solver. The method has to have a specific signature, i.e. the factory will always call this method with same general arguments and expect a specific type to be returned.

8. If the right keyword is set in the Chroma configuration, which is supplied as an XML[4]-file, the factory knows how to construct our class. It will roll out an instance of our solver, that can be called via the overridden methods in *syssolver_linop_dds*, which just talk to *dds_interface*, the interface that knows how to talk to the methods of our library.

This means that the software package just knows that our object is based on a well known construction plan. Because Chroma does not know our class in detail it cannot invoke specific functions. However, the functions that Chroma knows are the ones it can invoke. They will be executed, e.g. when the matrix has to be inverted. To create the connection between the interface class and our library the functions, which are outlined from the construction plan, have to be implemented in a custom way. This is called overriding. The advantage of this process is that we can give a static language a dynamic behavior without losing the advantages of strong-types.

---

[4] XML stands for *eXtended Markup Language* and represents a tag based ASCII text file with strict rules

3.1.4. High Performance Computing

Chroma does already include MPI, the message passing interface library. This is very helpful in High Performance Computing (HPC), since it allows us to write multi-core applications easily. One advantage of this is that we can focus on the actual parallel implementation and do not have to worry about machine based issues. We just write our application in a generic fashion and it will run with as many cores as we want to support on a single computer, a cluster of computers or a supercomputer.

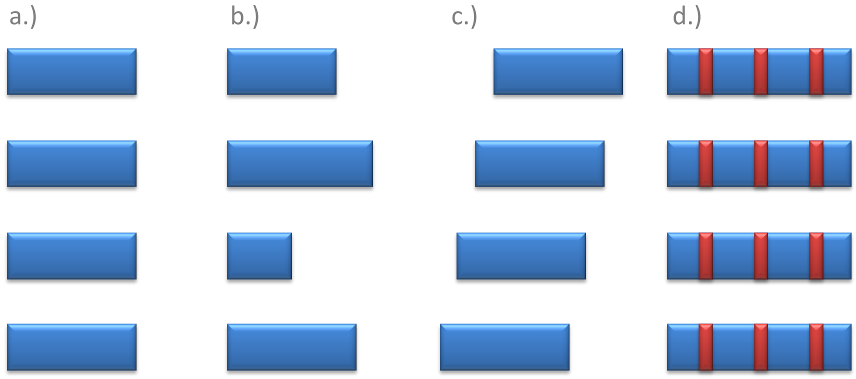


Figure 3.3.: Usually it is not possible to achieve perfect concurrency as shown in a). Unbalanced load as shown in b), a kind of latency in c) or overhead due to communication as shown in d) are common.

In order to optimize the application for HPC, i.e. creating a good parallel implementation of the algorithm, we have to do several tests and benchmarks using special tools. The results are then analyzed and motivate further improvements to make the application as efficient as possible. The most efficient HPC algorithms try to be as close as possible to model a) shown in fig. 3.3. Here we have a perfect concurrency with no overhead and therefore a perfect load-balance. Even embarrassingly parallel algorithms do not have such an efficiency since communication is required at least at the beginning and in the end of the execution. Communication does not work instantly, resulting in an offset, which is then shifting the starting time of the routine. Also computation time might not be the same on all cores or machines - even if the specifications are equal.

The reason for this behavior lies in the basic execution flow of a computer. From time to time a computer has to check devices like the keyboard in order to get data from them. This is the so called interrupt. Also the operating system is assigning computation time to applications, which results in some cycles not being spent on the actual computation but on basic management routines of the underlying operating system. All those statements

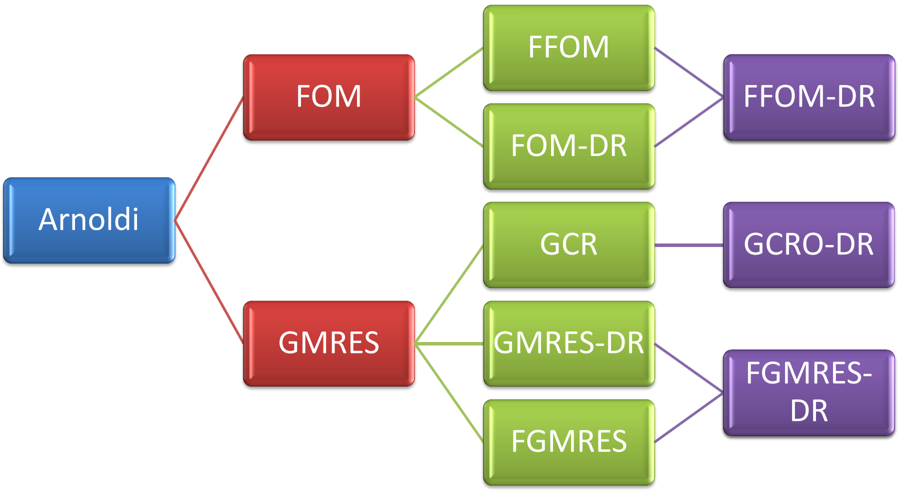


Figure 3.4.: FGMRES-DR is a direct ancestor of FGMRES and GMRES-DR, which share the common base of GMRES together with other algorithms like GCR.

support the point that a perfect parallel implementation is not achievable.

Nevertheless spending additional time on increasing the performance of our application for parallel machines can give us a significant advantage. When moving from one platform to the other we will most certainly deal with more cores instead of more direct computation power. In order to get more computation power out of a multi-core system we need to adjust our applications. If this is done before, our application does scale really well and can be used for a long time.

3.2. Major differences to FGMRES-DR

To alter the existing code that uses the FGMRES-DR algorithm correctly, we have to work out the differences between FFOM-DR and FGMRES-DR. We will do this by emphasizing the distinctions between two different sub-algorithms like general FOM vs. general GMRES.

This approach is possible because FGMRES-DR itself is just a combination of FGMRES and GMRES-DR. Therefore it is not a unique algorithm, relying on its own methods and calling procedures. So FFOM-DR would have a similar hierarchy as FGMRES-DR does have in fig. 3.4.

3.2.1. General differences

We already recognized that the Hessenberg matrix H of FOM looks different to $\tilde{H}$ of GMRES since $H \in \mathbb{C}^{m \times m}$ and $\tilde{H} \in \mathbb{C}^{m+1 \times m}$. This means that the last row of the least squares problem is deleted, resulting in a shorter solution vector. Similarly the right hand side of the equation $Hy = \gamma$ is changed with γ having one entry less than in the GMRES version.

Excluding the last row of H compared to GMRES results generally in more iterations since some information gets lost in the process. Computing the residue is also harder than in the GMRES version. GMRES uses γ_{m+1}, which is already computed. In FOM we do not have that entry therefore we have to use the more complicated expression given in eq. 2.35.

Generally we could think that FOM is more expensive than GMRES since we can at best achieve the same iteration count. We also have two additional operations (multiply and divide) in order to compute the residue. However, taking the more efficient memory allocation as well as the not-performed last rotation into consideration gives FOM some possibility of performing better than GMRES. The last argument comes from the fact that an $m \times m$ matrix needs $m - 1$ rotations in order to be triangulated while an $m + 1 \times m$ matrix requires m rotations.

3.2.2. Differences for restarting

The main change from GMRES-DR to FOM-DR lies again in the different matrix H instead of $\tilde{H}$. In order to form the new transformation matrix $P_{k+1} \in \mathbb{C}^{m+1 \times k+1}$ we do not have to orthonormalize $\gamma - Hy$ to form the new column p_{k+1}. Instead, we can just use $p_{k+1} \equiv e_{m+1}$.

Morgan [Mor02] worked out the main differences of GMRES-DR and FOM-DR already. We can apply his work to simplify our problem. According to him most of the code remains unchanged with just small alterations by replacing $\tilde{H}$ with H. One thing we have to be careful about are matrix dimensions, since the matrix represents a two dimensional array.

This means that changing the dimension of the matrix, i.e. from $(m+1) \times m$ to $m \times m$, results in changing nearly every call of the array. The reason for this lies in the storage attitude of the computer. The access to data is given by a number, which results in a one dimensional stream of data. Consequently we have the following transformation for going from a two-dimensional (matrix) to a one-dimensional (data) array:

$$M_{i,j} = a\left[i \times m + j\right], \tag{3.1}$$

where M is an $(m + 1) \times m$ matrix and a is an array with length $L = (m + 1) \times m$. Since we altered the dimensions of our matrix we have to modify several array allocations, as well as some array accesses, because there is no $Mm + 1, j$ with $j = 1, ..., m$.

3.2.3. Flexible comparison

The preconditioner does not have to be changed since it examines the matrix independently of the solver. The flexible implementation of FOM is quite similar to the flexible implementation of FGMRES. In both cases we lose the validity of our original Arnoldi relation as described in eq. 2.22 and replace it with a new equation like eq. 2.61. We see that $\tilde{H}$ is just replaced by H in case of FFOM.

Even though the preconditioner is specialized for our solver, e.g. with the iterative refinement algorithm as described in section 2.2.8.6, we will use this implementation for our first runs with FFOM-DR. We will have a look at further improvements in section 3.5. In that section we will discuss improvements in handling the preconditioner for having an even faster implementation of our algorithm. Right now we want to take the chance to implement FFOM-DR as simple as possible, i.e. just considering the most obvious and required changes.

3.2.4. Savings with FFOM-DR

By using FFOM-DR we end up with less memory being used than with FGMRES-DR. However, since the allocated Krylov subspace mostly does not exceed 30 vectors, the total memory savings can be neglected. This is also the reason why the Hessenberg matrix H_m is usually just allocated as an $(m+1) \times m$ block (GMRES). The extra savings by omitting the empty cells do not justify the higher complexity that comes with accessing elements of H_m.

More interesting are the savings on the computational side. Most of those savings are situated in reduced loops by omitting one full iteration. This results in one Givens rotation less as well as less overhead for building up the projectors to perform deflated restarts. We now look at the savings that can maximally be obtained by using a full cycle of GMRES before doing a deflated restart.

The deflated restart will be performed with $m/2$ vectors where m is the number of currently saved Krylov subspace vectors. Summing up the floating point operations (that are initially set to single precision, but could also be computed in double precision) we obtain that the total savings are about

$$S \approx 24 \cdot \left[\left(\frac{m}{2} + 1 \right) + (m+1) \right] = \mathcal{O}(m). \tag{3.2}$$

The overall savings are proportional to the size of the Krylov subspace. Considering a usual subspace size of 30 we would save about 1200 floating point operations, which can be neglected considering the systems we use. This means that the computational savings would not be sufficient to pick FOM over GMRES as the core algorithm.

3.2.5. Summary

From the mathematical perspective FFOM-DR and FGMRES-DR are equivalent. On the computational side there are some important differences. The implementation of FFOM-DR by modifying an existing FGMRES-DR is not hard to accomplish, since most of the changes are not extensions but small modifications, e.g. by accessing or allocating the array.

Some changes like in the orthonormalization step of the deflated restarts part of the algorithm require more knowledge. However, since various papers deal with similar problems the theoretical framework has already been built. Using this as a basis we can alter the existing code to obtain an implementation of FFOM-DR.

3.3. FFOM-DR

The algorithm uses the basis of alg. 6 with extensions as shown in alg. 9 for deflated restarts and in alg. 10 for flexible preconditioning. The combination forms a new algorithm called FFOM-DR, which is quite similar to FGMRES-DR. The main differences have been discussed in sec. 3.2.

Obviously a lot of other algorithms are used to provide FFOM-DR. All algorithms that are used in alg. 12 are included in this work. Some of these algorithms are also described. The code for building the solution is exactly the same as in alg. 7. The Givens rotations are performed using alg. 12.

We will now explain the interesting parts of the algorithm. The first thing to notice is the so called *sanity check*. This is necessary because of the usage of mixed precision and will not be required any more when using double precision. It will basically do two important things:

1. Recalculate the vectors by using the information provided in the Hessenberg matrix H_m.
2. Calculate the true residue by performing the matrix-vector product $b - Ax_d = r_d$. If the real residue is more than a magnitude different from the approximate one the algorithm will restart.

Originally the real residue is replacing the approximate one. However, for FOM the relations from GMRES do not hold any more. This is discussed in depth in chapter 2.

The modified residue implies modifications in the methods for obtaining the true residue. The original method calculated the true residue of the system and inserted this value back into the solver in order to make the calculation more accurate. It also provided a safe

Algorithm 12 Flexible FOM with deflated restarts and mixed precision

```
Input  $A \in \mathbb{C}^{n\times n}$, $b \in \mathbb{C}^n$, $x_0 \in \mathbb{C}^n$, $T \in \mathbb{R}_+, m \in \mathbb{N}, k \in \mathbb{N}$
Output  $x_l \in \mathbb{C}^n$
  $r_0 := Ax_0 - b$
  $q := 1$
  $\gamma := \|r_0\|$
  while  $\gamma > T$  do
    for  $j = q \to m$  do
      Do Arnoldi step $j$ with preconditioning matrix $M_j$
      Do Givens rotations on the $j$-th column to get updated $\gamma$
    end for
    Build approximate solution $x_j$
    Check sanity (obtain true residue)
    $z := \min(k, j/2)$
    if  $z \leq 1$  then
      Restart and forget subspace, i.e. $q := 1$
    else
      Perform deflated restart with $z$ eigenvectors
    end if
  end while
```

mechanism to check if the iterative refinement algorithm did have unwanted effects on the overall outcome. By modifying the algorithm for this method we can still benefit from it.

Beside some formal differences the main change lies in the algorithm for the deflated restart. This change has already been discussed. Nevertheless it is important to realize that the switch from harmonic Ritz values to regular Ritz values will be responsible for nearly all the differences visible in benchmarks between FFOM-DR and FGMRES-DR.

3.4. Implementation issues

Changing the existing FGMRES-DR to get FFOM-DR is a task that does not require a lot of modifications. There are no additional methods and variables needed in order to obtain a suitable implementation. However, there are some subtleties one should be aware of.

One important thing to think about are Fortran-style matrices. On several points in the code some methods from Fortran or Fortran based libraries are called. Applications built with the programming language Fortran save and access two dimensional arrays (matrices) different than those written with the programming language C and its descendants.

One result of this difference is that eq. 3.1 does not hold any more and looks different on the right hand side. The new relation within the same notation is now

$$M_{i,j} = a\left[i + j \times (m+1)\right], \tag{3.3}$$

where M is still a $(m+1) \times m$ matrix. We see that the different style of accessing matrices results in a different way of storing and accessing data of the matrix on the computer.

An important issue came up while examining the ZQR function of LAPACK. This function is responsible for performing a QR factorization and was called by FGMRES-DR previously. In our new implementation we realized that the result given by this function is sometimes not consistent with the true residual insertion. The reason for this behavior lies in the QR factorization itself, since the result from the factorization is not unique. It turned out that the wrong result had a sign flipped with respect to the true residual orthonormalized Ritz vector in the lattice basis.

To correct this wrong behavior we have to insert a check before we use the returned values. This check will have a look at the last entry of the vector. This entry has to be positive since it is the norm of the residue vector. Knowing this we can just flip the sign and obtain the real residue.

3.5. Further improvements

By comparing FFOM-DR to FGMRES-DR we see that we will not gain much at all. In some special cases we might even be punished by omitting some important information about our subspace. This raises the question if we could include improvements with methods that are not possible in the framework of the original FGMRES-DR.

3.5.1. No mixed precision

One problem is that we have to use iterative refinement. This technique was required in the original code since it enabled us to use single precision for our calculations. Now that we switched to FOM we lost some accuracy. Therefore we need some other tricks to gain accuracy. Switching to double precision and deactivating the iterative refinement is an obvious solution to this problem.

Since the gain of using single precision is much smaller than a factor of two we can neglect the advantages provided by using this technique to obtain more accuracy. The basic idea behind this is to win more by less iterations via more accurate eigenvectors than to loose by more memory consumption and lower computation speed. A key factor will be the fact that most of the computation time will still be spent in single precision with the preconditioner.

The implementation of this change requires some careful work. It has to be determined which variables have to stay in single precision and which variables have to be changed into double precision. This segmentation is required since the preconditioner will not be changed. As mentioned the preconditioner itself has to remain in single precision. This seems odd but has some good reasons, mainly that the precision of the preconditioner will never become sufficient to justify double precision.

This means that all single precision variables that are directly used by FFOM-DR for computations will be changed while variables which work only as a buffer or connection point to the preconditioner will stay in single precision. The work was basically split up into two parts:

1. Find out what has to be in double precision and change the variables (and allocations).
2. Add or alter code in order to have the preconditioner working in single precision with the rest of the program being in double.

With the code now being completely in double precision we can neglect the sanity check and omit the computation of the true residue. This could be an important check for the user, however, since it requires a full matrix vector product we cannot justify its appearance any more.

The gained stability is the key argument for the removal of the iterative refinement technique. The Krylov subspace and the deflation subspace are very stable in double precision. This makes the consistency check obsolete, while most of the computation is still done in single precision, thus retaining most of the single precision advantages.

3.5.2. Variation of κ

Another interesting modification is made possible by the observation that κ can be different for the preconditioner. This means that we can speed up the preconditioner very flexible without affecting the actual solution of the whole problem. This has the same reason as our choice to use single precision for the preconditioner. Since the preconditioner is not used to solve the system, but only to guess a rather coarse approximation to get a better condition for the solver, we can use it with a slightly different mass than the original solver.

The required changes are straight forward. Before the preconditioner calls the Dirac operator the mass is shifted resulting in a lower κ. After the Dirac operator has been applied we reset the mass shift in order to restrict this change to the preconditioner. The important lines in the implementation are:

```
#ifdef PREC_SHIFT_MASS
  sw_term_s(0.012);
#else
  sw_term();
#endif
  (*ifail) |= assign_swd2swbgr(GCR_BLOCKS, ODD_PTS);
  gauge_updated = 0;
#ifdef PREC_SHIFT_MASS
  sw_term();
#endif
```

We used the definition of *PREC_SHIFT_MASS* to enable or disable the mass shifting for the preconditioner. What can be seen is that a fixed shift of 0.012 has been used. This number lowers κ by increasing the bare mass m_0. Therefore the shift does not act directly on κ. The shift could be outsourced as a variable parameter. However, since we know that $m_0 + 0.012$ should be sufficiently small to be solved by the preconditioner in a minimal time we can leave this “magic number” in the code in any case.

3.5.3. Adjusted deflation

The last improvement is based on the fact that the first Ritz eigenvectors have not converged well. So we are moving inappropriate vectors around - even using them for deflation. This is obviously not good for the computation speed as well as the overall performance measured in iterations. Thus it is our goal to reduce the usage of bad (too high or too inexact) vectors by restarting in the beginning of the program execution.

The number of vectors to keep will be adjusted corresponding to the execution time of the program measured in iterations and weighted with a factor of the convergence. This approach sounds reasonable, but its also contains some problems that have to be solved:

1. There is no criterion that can tell us if vectors have or have not converged well.
2. Usually we do not know if we are obtained the smallest or just some eigenvectors by looking at Ritz vectors.
3. The execution time of the program depends on the problem. It is possible that some matrix is solved in 40 iterations, while a similar one needs 80 iterations.

The FGMRES-DR algorithm already contains conditions to the number of eigenvalues. The following conditions are either necessary in order to work or have been proven to be

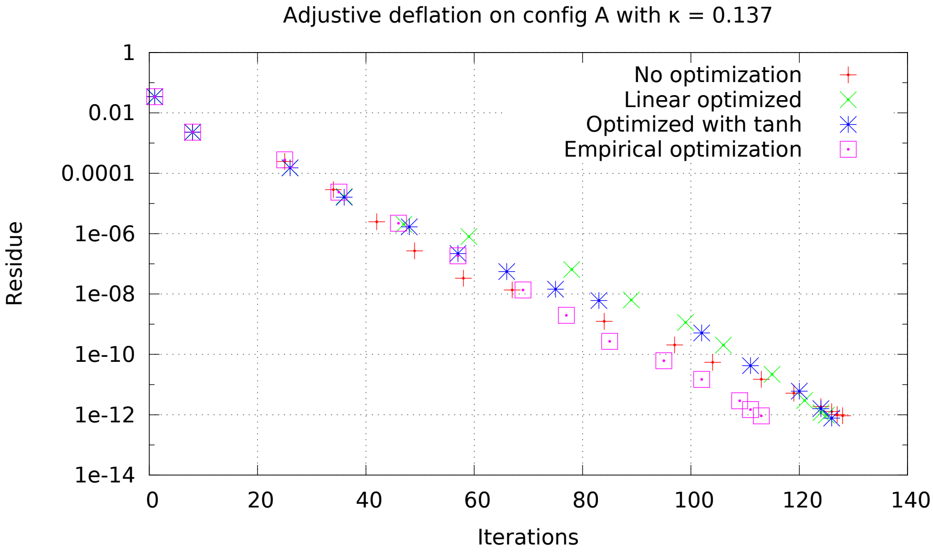


Figure 3.5.: Trying out various simple and empirical variations of determining the number of deflation vectors at one configuration motivates the creation of the adjusted deflation algorithm.

efficient for FGMRES-DR:

- The number of Ritz values is restricted to the number of currently stored Krylov subspace vectors.
- Only the lowest Ritz values will be picked.
- We are not allowed to pick more than half of the available Ritz values.
- We are not allowed to pick more than the set default number of eigenvalues, i.e. the number that has been set up by the user.

With u denoting the number of eigenvalues that should be used for deflation (set by the user) and the number of eigenvalues available with f we see that the number of eigenvalues to actually use for deflation d is computed by

$$d = \min\left\{u, \frac{f}{2}\right\}. \tag{3.4}$$

We will now try to introduce the technique of adjusted restarting to FFOM-DR. The main motivation has been given by the plots in fig. 3.5. Here we see that it is possible to straighten the derivative and gain a lot of iterations just by using a variation of deflation vectors. Therefore adjusted deflation should give us the following key advantages of the (more or less constant) restarting:

- Trying to deflate only with converged eigenvalues, which minimizes the risk of deflating with false eigenvalues.
- Detecting the deflation with false eigenvalues and exclude them from further usage.
- Minimizing possible stagnation and increasing the probability for a monotonically decreasing residue in the logarithmic view.

The plot shows the original deflation in action and two possible methods to reach adjusted deflation. Both methods are based on the current progress p with $0 \leq p \leq 1$. The linear optimized deflation increases the amount of deflation vectors linearly with the progress. In the end the maximum number of deflation vectors is the same as the maximum number of deflation vectors set by the user. This method performed slightly better than the standard (constant) deflation. Also another approach using $\tanh(p)$ as method to determine the number of deflation vectors brought nearly the same results as the linearly determined version. The best result was obtained by a manually set version with empirically tuned parameters. Here we already gained a lot more iterations and were able to straighten the line of progress in the logarithmic residue in dependence of iterations plot.

One of the problems in implementing this technique is the limitation in information regarding the system. If we would know what we know afterwards, the implementation would be straightforward. However, this is not the case, i.e. we are restricted to some pieces of information. We will have to draw conclusions based on those pieces. The adjusted deflation algorithm is based on the following information:

- The current cycle of deflated restarts s, with $s = 1$ for the first cycle.
- The current progress p of the solver, which is 0 in the beginning and 1 in the end defined as
  $$p(I) \equiv \frac{\log r(I)}{\log T}, \tag{3.5}$$
  with r being the current residue and T being the desired residue. I represents the number of iterations.
- The current slope of the progress in the residue per iteration plot. We can calculate the slope by approximating the derivative, i.e.
  $$p'(s) \equiv \frac{dp(I)}{dI} \approx \frac{p(I_s) - p(I_{s-1})}{I_s - I_{s-1}}. \tag{3.6}$$
  Here the number of iterations I will be set either to current number of iterations I_s or to the previous number of iterations I_{s-1}.

Algorithm 13 Adjusted deflation

```
Input $s \in \mathbb{N}$, $r \in \mathbb{R}$, $I_s \in \mathbb{N}$, $f \in \mathbb{N}$, $T \in \mathbb{R}$, $u \in \mathbb{N}$
Output $v_t$
  $p_s := \log r / \log T$
  $p'_s := (p_s - p_{s-1})/(I_s - I_{s-1})$
  if $s = 1$ then
    $\Delta := u/4$
    $L := \Delta$
    $v_s := 2 \cdot \Delta + 1$
  else
    if $v_t = v_{s-1}$ then
      $l := \Theta(p'_s - p'_{s-1})$, where $\theta$ is the step function
      $L := \max\{l \cdot v_{s-1}, L\}$
      $v_s := \max\{L, v_{s-1} - 1 + l \cdot (\Delta + 1)\}$
    else
      $v_s := v_{s-1} + 1$
    end if
    $v_s := \min\{v_s, u\}$
  end if
  $v_t = \min\{v_s, f/2\}$
```

With these three main quantities identified we need to develop an algorithm that uses them wisely in order to avoid magic numbers in our code. The problem that we would face using empirical identified magic numbers is that they tend to work only on some systems and not in general. We would like to avoid this scenario and let our algorithm determine the optimum number of steps in order to optimize the deflation.

The algorithm is displayed in alg. 13. It should be used in combination with FFOM-DR and can be used with FGMRES-DR as well. This algorithm replaces the current condition for the number of deflation vectors in alg. 12, which is denoted by z there. Therefore we would use v_t instead of z by inserting the adjusted deflation algorithm into the main algorithm.

The scheme shown in fig. 3.6 illustrates a possible outcome using the adjusted deflation algorithm in a very simple example based on four restarts. In the first restart with $s = 1$ we initialize the proposed number of vectors to be $2\Delta + 1$ and the lowest level to be at Δ. Δ is set to the $u/4$, where u is the user defined number of maximum deflation vectors. In the next restart the previously taken number of deflation vectors is compared with the

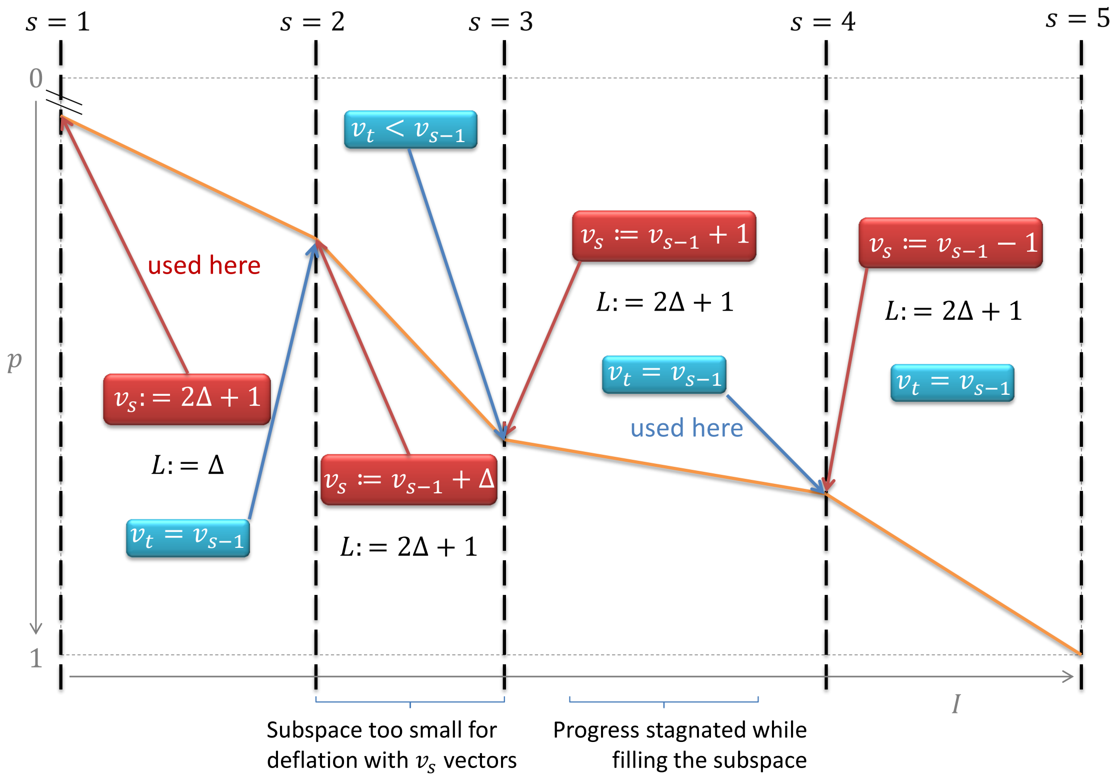


Figure 3.6.: Scheme of adjusted deflation in a run with four restarts with the current progress p, the number of iterations I, the proposed number of deflation vectors v_s, the lowest number of deflation vectors L and the actually taken deflation vectors v_t.

actually taken number. In this example we had enough iterations so that our subspace could grow big enough. The progress seemed to be satisfying, which is why we set the lowest reachable level to be equal to v_{s-1}. We also increase the number of proposed vectors by Δ. The progress performs even better now, but this results in a small subspace.

One of the problems that is attacked by adjusted deflation is that the derivative of the progress is less important than the subspace size. Obtaining a smaller subspace means that the system obviously converged fast enough. In this case we just increment the previously proposed number of deflation vectors. Otherwise we cannot make any statement. This results in the lowest level not being changed at all. In the final restart with $s = 4$ we see that the progress did dramatically decelerate resulting in a lot of unnecessary iterations. Therefore we decrement the currently proposed number of deflation vectors. Finally the systems converged making another round of adjusted deflation obsolete.

The adjusted deflation algorithm utilizes fixed levels in order to take a guaranteed number of eigenvalues. The fixed level starts very low to ensure that critical systems will fall back to a minimum amount of deflation eigenvectors. Besides the starting number of eigenvalues has been set to be slightly above half of the user defined number of deflation vectors. Since the gap of the levels is a quarter of the deflation vectors this is equivalent to using two times the level spacing plus one.

The next condition is whether the number of proposed vectors could be used or not. If not then the system did not have enough Krylov subspace vectors. This can only be the case if the system did converge fast enough to the current intermediate solution. In this case we can increase the amount of proposed eigenvectors. Otherwise we look if the derivative of the current progress is bigger than the former derivative. We use the Heaviside step function in order to illustrate the outcome. We could rewrite the argument of the step function to obtain

$$\Theta(p'_s - p'_{s-1}) = \Theta\left(\frac{p'_s}{p'_{s-1}} - 1\right) = \begin{cases} 1 & \text{current derivative is bigger,} \\ 0 & \text{otherwise.} \end{cases} \tag{3.7}$$

This concept is special since it shows that we use a relative quantity. Relative quantities are interesting due to their generic nature. Using such values will enable us to make general comparisons, e.g. is bigger than the identity or smaller than the identity. The last lines of the algorithm just make sure that in case of a smaller derivative we decrement the amount of eigenvectors as well as staying inside the bounds defined by the current level L and the user defined deflation vectors u.

3.6. Merging both algorithms to form AOM

Our big goal is to find some criterion to determine when one algorithm is superior to the other. In order to follow our path and be flexible with our solver algorithm we will have to bring both algorithms, i.e. FOM and GMRES, into one implementation which picks the most suitable algorithm for the current system. The final algorithm will be called Adjusted Orthogonalization Method or short **AOM**. The AOM algorithm will provide the following properties:

- A double precision version of FGMRES-DR in the core for problems under the condition that FFOM-DR will certainly provide worse results. The criterion for this condition is somewhat difficult to find, however, we will see in a later chapter that one is able to find empirical data there. The criterion could also be outsourced to be optimized by a sophisticated user.

- FFOM-DR in the core for problems that do not match the condition of using FGMRES-DR as the core algorithm.

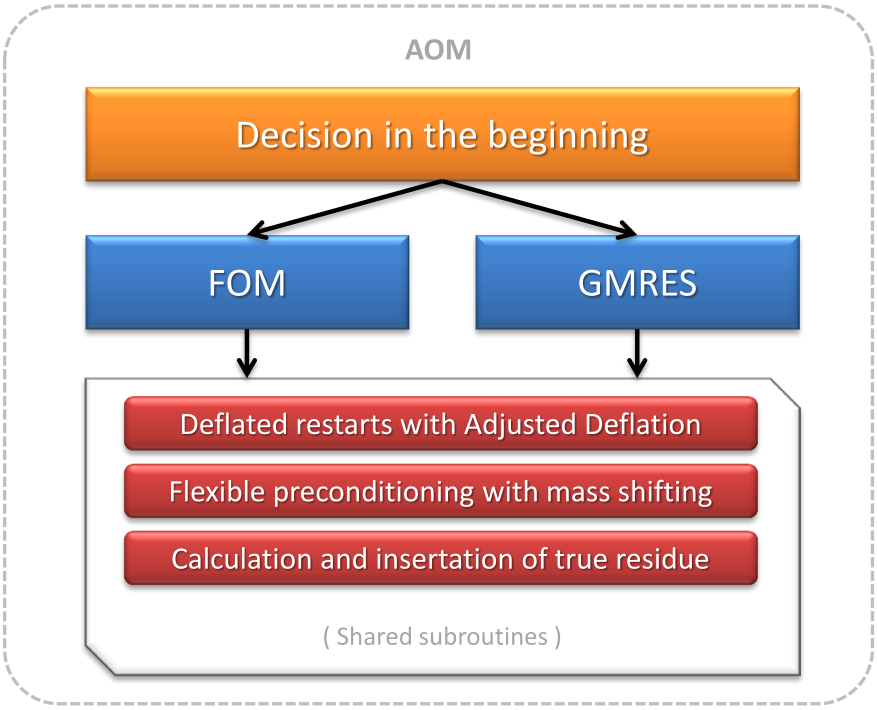

AOM
Decision in the beginning
FOM
GMRES
Deflated restarts with Adjusted Deflation
Flexible preconditioning with mass shifting
Calculation and insertation of true residue
(Shared subroutines)

Figure 3.7.: AOM combines both algorithms and includes all the improvements, which have been implemented in FFOM-DR previously in order to form an optimized solver for lattice QCD.

- Everything (except the preconditioner) will be computed in double precision to avoid malicious behavior such as stagnation. This will also ensure higher precisions and tolerance levels.
- Including a mass shift in the preconditioner to speed up the calculation of the approximate inverse even more. Gaining speed in the preconditioner while keeping the iteration count about equal will give us a great benefit for the overall performance.
- Providing a more sophisticated way to calculate the optimal number of deflation eigenvectors. This is done by the adjusted deflation algorithm. The user only has to define the maximum number of vectors that should be deflated. This maximum number will be taken to define certain levels and jumps in order to determine the optimal number of deflation vectors.

The main features of AOM are the improvements already discussed. Those improvements are also interesting and applicable for FGMRES-DR. Therefore merging both algorithms was a logical step. Due to intelligent software engineering a lot of the methods have been reusable or could have been extended in order to support the FOM part as well as the

GMRES based part. AOM also leaves out redundant features of FGMRES-DR such as the Iterative Refinement algorithm. The lost time is rewarded with a more robust algorithm and speed ups due to gained accuracy.

The basic construction of AOM is shown in fig. 3.7. We see that AOM is mainly about two branches combined by a decision which to take and some shared methods. The preconditioning and the deflated restart itself as well as the calculation of the true residue do not have to be changed at all. Methods like building the projector for the deflated restart and inserting the true residue have to be specialized for the two basic algorithms. Furthermore there are some routines like the Givens rotations and others which can be shared easily. Basically every routine that does not require a format or a calculation that is specific to either FOM or GMRES can be shared.

4. Benchmarks and Tests

In order to measure the success of our project we need to perform several benchmarks and tests. We also need to define the word *success*. In case of our implementation we talk about a success if FFOM-DR performs better than its competitor, i.e. in our case FGMRES-DR. We will measure the performance by using the iteration count. Therefore we need to perform the tests and benchmarks using FFOM-DR with the same configurations and the same environment using FGMRES-DR. The Chroma application helps us to run practical tests on both algorithms in order to create comprehensive plots and meaningful data.

We will use several pre-made lattice configurations, which are available on the network drive of *kerndata8*. The specifications for these configurations are listed in tab. 4.1. A more detailed listing of those configurations can be found in appendix A.

	type (smearing)	n_x	n_y	n_z	n_t	n_f	β	c_{sw}	κ	κ_c
config A	wilson (—)	16	16	16	32	2	5.29	1.91	0.1355	0.137
config B	wilson (stout)	16	16	16	32	$2+1$	5.5	2.65	0.1203	0.1283
config C	wilson (stout)	16	16	16	32	$2+1$	5.5	2.65	0.1203	0.1285
config D	wilson (stout)	16	16	16	32	$2+1$	5.5	2.65	0.1203	0.1287
config E	wilson (stout)	16	16	16	32	$2+1$	5.5	2.65	0.1203	0.1285
config F	wilson (stout)	16	16	16	32	$2+1$	5.5	2.65	0.1203	0.1288

Table 4.1.: List of configurations used for the benchmarks and the parameters of their production

A better performance can be determined by a faster execution time for the same problem. Since execution time and iteration count are somehow related we can use the iteration count after determining an approximate relation between those two values. We will discuss the advantages of using iteration count over execution time later.

4.1. Basic evaluation of FFOM-DR

Starting with *configuration A* we produced a plot showing the development of iterations and execution times of the original FGMRES-DR in dependence of κ used for the solver.

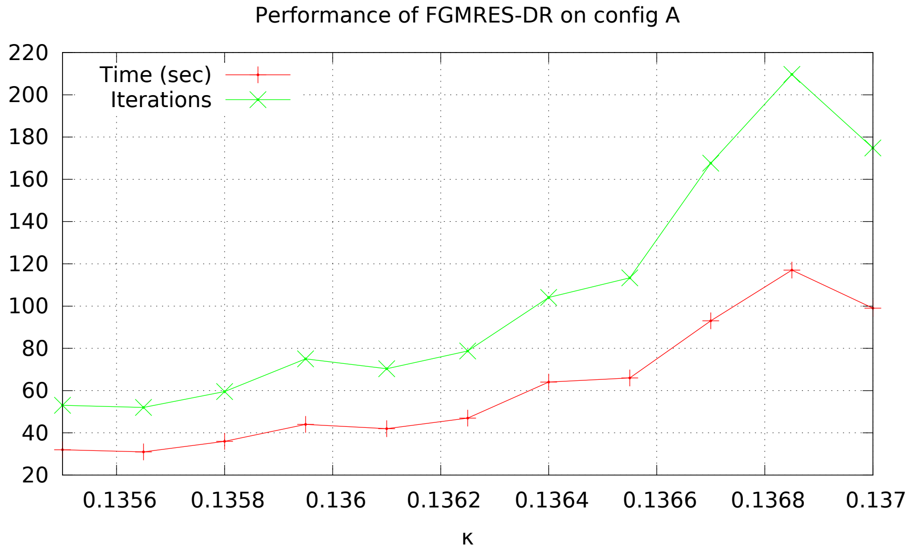


Figure 4.1.: Performance of FGMRES-DR on **config A** with different values of κ.

We note, however, that tuning the κ for our solver while using a fixed configuration with fixed κ as right hand side can only give us a qualitative idea of the performance of these algorithms. This plot is show in fig. 4.1. We can easily observe that the values are correlated. The execution time has a different factor than iterations resulting in a higher time scale (red) than iteration scale (black). We prefer to use the iteration number instead of the execution time, since the latter varies slightly as explained previously in section 3.1.4.

We can also see some interesting properties. First of all the iteration count does not increase monotonically with κ. This is counter intuitive and has no direct reason and solution. What we can say is that certain matrices are more favoured by FGMRES-DR than others. At $\kappa = 0.137$ which represents κ_c, we do have a very tough system with eigenvalues very close to zero. But we can see that FGMRES-DR does solve this problem faster than the matrix with $\kappa = 0.13685 < \kappa_c$.

After the test of FGMRES-DR we perform the same benchmarks with FFOM-DR expecting that it follows roughly the trend of FGMRES-DR. Basically this intuitive estimation is right (fig. 4.2), but the differences are quite surprising. The plot showing a direct comparison of both algorithms using the iteration count is displayed in fig. 4.5.

First of all we notice that $\kappa = \kappa_c$ does require a lot more work with our algorithm compared to FGMRES-DR. We expected our algorithm to be slower, but not that the relative factor was as large as 0.3 or 30%. The next property was the drop around $\kappa = 0.13655$. This did not happen in our evaluation with FGMRES-DR. The result is that all

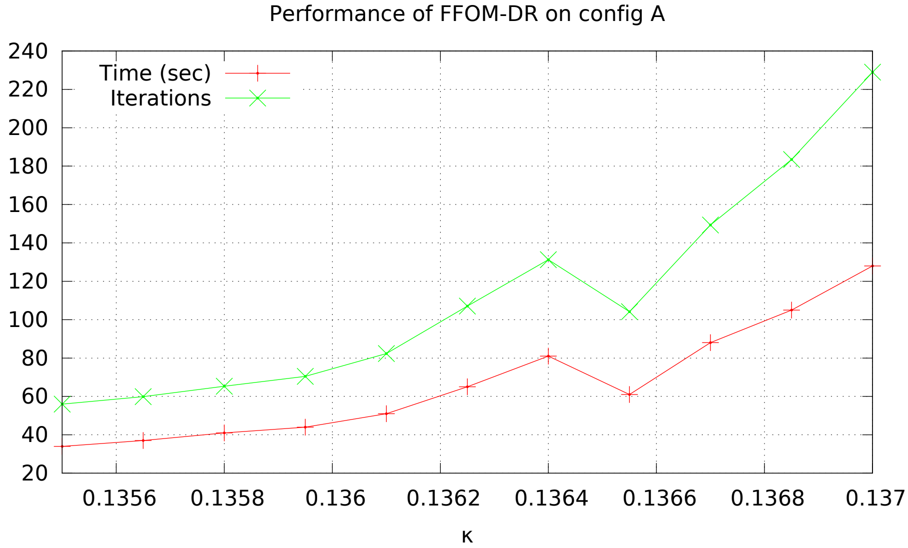


Figure 4.2.: Performance of FFOM-DR on **config A** with different values of κ.

but the last evaluations using κ close to κ_c are faster in the new algorithm.

Another interesting point occurred at $\kappa = 0.13595$. Here we have the same number of iterations for both algorithms. By measuring the time we could observe that our algorithm can save computation time while having the same number of iterations. Again time does not say as much as iterations do - so we will not use this in order to make a general statement. However, this one data point emphasizes our statement of FGMRES-DR performing worse than our algorithm in some situations.

The two plots basically show that is not a trivial job to decide which algorithm to choose for a certain system. Even though a matrix might look harder to solve than another it does not state which algorithm is more suitable for actually solving it. For configurations with a κ being far away from κ_c, i.e. $\kappa \ll \kappa_c$, we can state that the decision between FFOM-DR and FGMRES-DR does not matter. In the worst case the loss is lower than 10%.

However, once the problem becomes more difficult, i.e. if $\kappa_c - \kappa \leq 0.001$, we can significantly boost our computation by using the better algorithm for the job. The iteration count of both algorithms is shown in tab. 4.2. An more complete version, including the time values, is shown in appendix B.

Using the plots that have been shown previously we are able to make a first statement: The (original) FFOM-DR can be faster than FGMRES-DR and accelerates convergence for some problems. However, the risk of losing time can be bigger than the possible gain taking into account the chance of picking the right problem and the execution time that

κ	**FGMRES-DR**	**FFOM-DR**
0.1355	32	34
0.13565	31	37
0.1358	36	41
0.13595	44	44
0.1361	42	51
0.13625	47	65
0.1364	64	81
0.13655	66	61
0.1367	93	88
0.13685	117	105
0.137	99	128

Table 4.2.: FGMRES-DR vs. FFOM-DR on **config A** with various values for κ

is actually saved.

Another interesting quantity is how the computations really behaved in detail, i.e. the residue in dependence of the iteration count. This gives us some insight what could have gone wrong in the computation, i.e. where the computation did have problems. In fig. 4.3 we see that nearly all problems, e.g. stagnation or a slow down in the residue, are situated in the range of $10^{-9} \leq r^\dagger r \leq 10^{-6}$.

The interesting results do not include any hint about a possible explanation for the drops of FFOM-DR nor FGMRES-DR at certain values of κ. So we started with a simple evaluation of the eigenvalue spectrum in dependence of the iteration count. From the theory we know that the following properties will be found:

- With more iterations we will find less eigenvalues but more precise ones.
- Already found eigenvalues will either converge to the real ones or can be omitted.
- The eigenvalues will lie on the positive real axis. If this would not be the case then our κ is beyond κ_c resulting in a non-physical lattice.
- Easier problems will have a less scattered, thus more concentrated, eigenvalue spectrum.
- Harder problems will have a less concentrated eigenvalue spectrum with some eigenvalues being quite close to zero on the real axis.

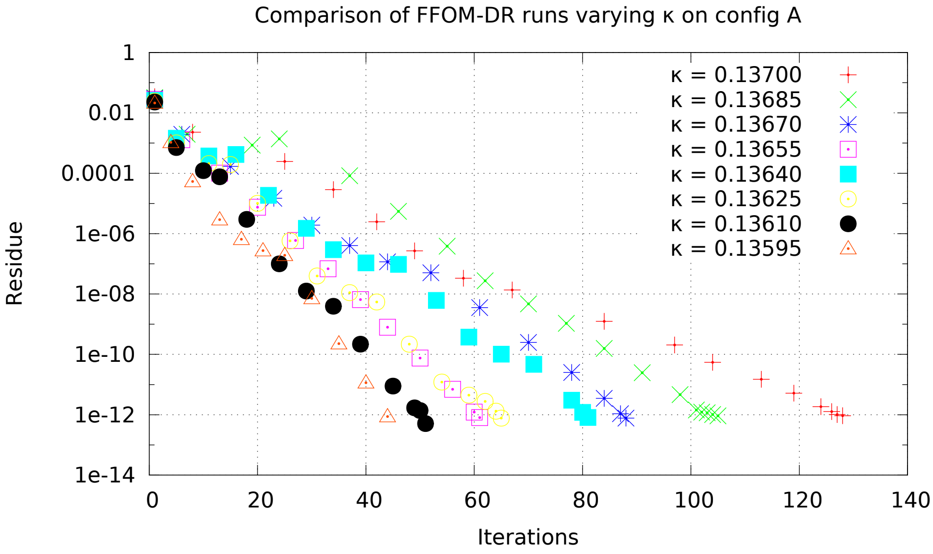


Figure 4.3.: Comparison of several FFOM-DR runs on **config A** using a different κ.

In fig. 4.4 we see these properties. One problem that can be observed from the spectrum at $\kappa = 0.13685$ is that most of the eigenvalues that we find have to be disposed of since they do not converge to real eigenvalues. This is a huge problem, because we have to keep them in memory (and might omit real eigenvalues in the process). They also make the problem harder to solve than it actually is.

The first evaluation gave us some insight where FFOM-DR might have advantages over FGMRES-DR and where we have to think about improvements. In order to be more accurate we have to do more evaluations with this implementation before we can test further improvements.

4.2. More evaluations of FFOM-DR

In the other evaluations concerning the configurations config B to config F from tab. 4.1 we observe the same behavior. With help of these evaluations we are able to make a rough empirical statement when we should really prefer one algorithm over the other. The plots are shown in fig. 4.5 and 4.6. In 23 out of 41 measurements FGMRES-DR was ahead of FFOM-DR. It is noticeable that all 6 cases where $\kappa = \kappa_c$ belonged to those 23. This means that FFOM-DR is unfavorable at κ_c, which is usually the κ where measurements are done in lattice QCD. However, this also implies that at $\kappa \neq \kappa_c$ we can pick FFOM-DR as well, since it performs about equal to FGMRES-DR.

We investigate the results in order to determine a threshold for κ. This gives us a clear

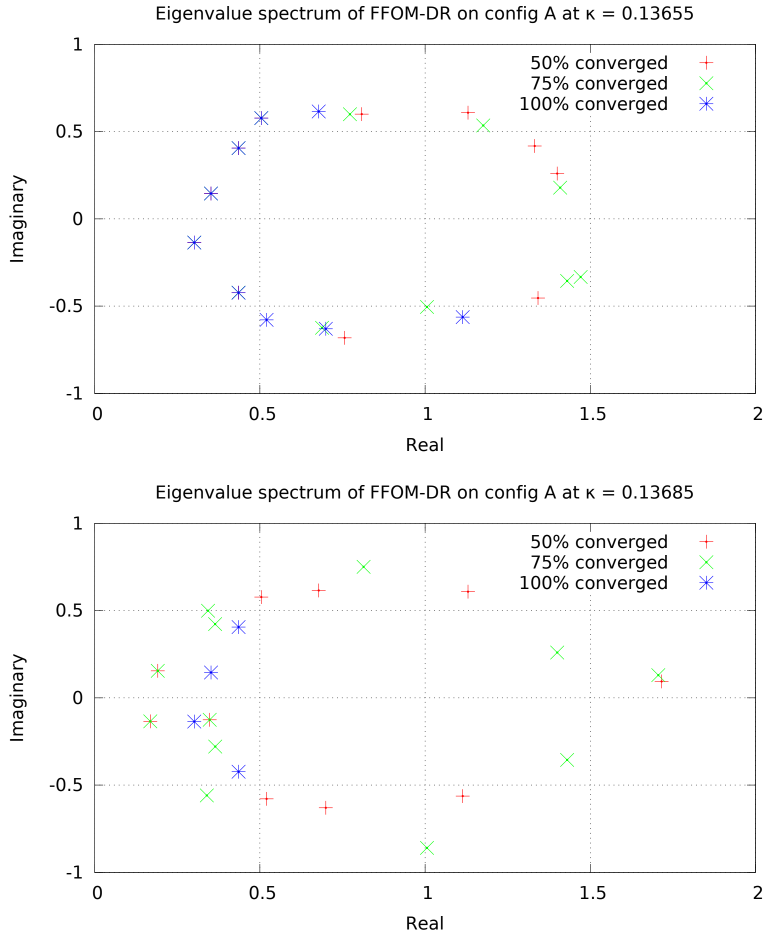


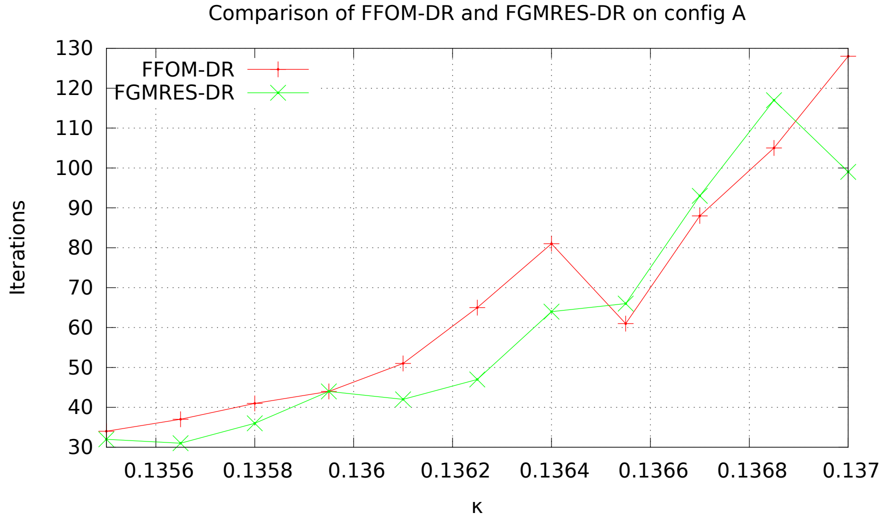
Figure 4.4.: Eigenvalue spectrum of FFOM-DR on **config A** using $\kappa = 0.13655$ shown in the first plot and $\kappa = 0.13685$ shown below.

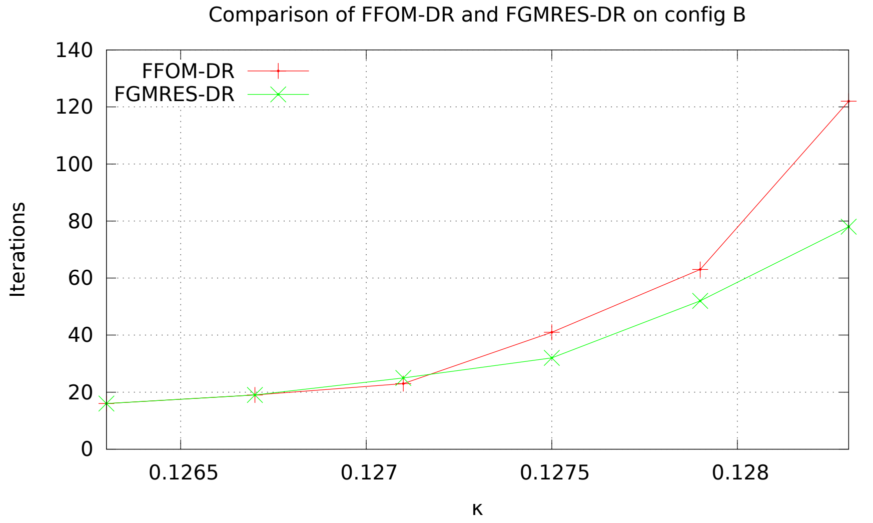
indicator when it would be better to prefer our algorithm over FGMRES-DR. Looking at the numbers we can identify such a threshold. FFOM-DR is usually the better choice if

$$\kappa/\kappa_c < 0.99 \quad \Rightarrow \quad \frac{\kappa_c - \kappa}{\kappa_c} > 0.01. \tag{4.1}$$

That leaves us with the most interesting region being dominated by FGMRES-DR. Except for $\kappa = \kappa_c$ we observe that FFOM-DR outperforms FGMRES-DR in 18 out of 35 cases, however, in $\kappa_c > \kappa > 0.99\kappa_c$ only in 7 out of 24 cases. This means that there is only a small chance that our algorithm accelerates the computation speed. The question now is how much is the cost if that is not the case, i.e. is it pure gambling to pick FFOM-DR? In order to find an answer to this question we have to sum up the execution time differences of all evaluations in the interesting κ region.

Unsurprisingly it is indeed a risky choice to handle every lattice with our implementation. Considering all cases, we calculated that the time spent in our algorithm, denoted





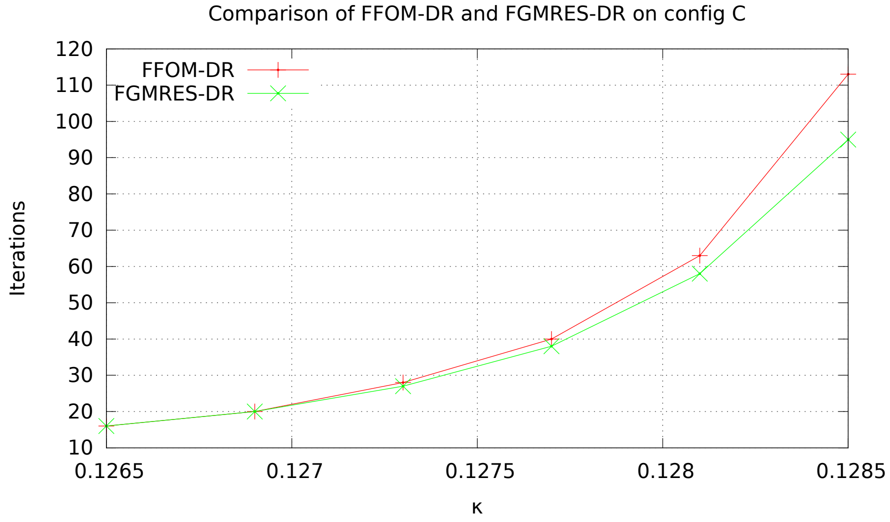


Figure 4.5.: FFOM-DR vs. FGMRES-DR on **config A**, **config B** and **config C** with different values of κ.

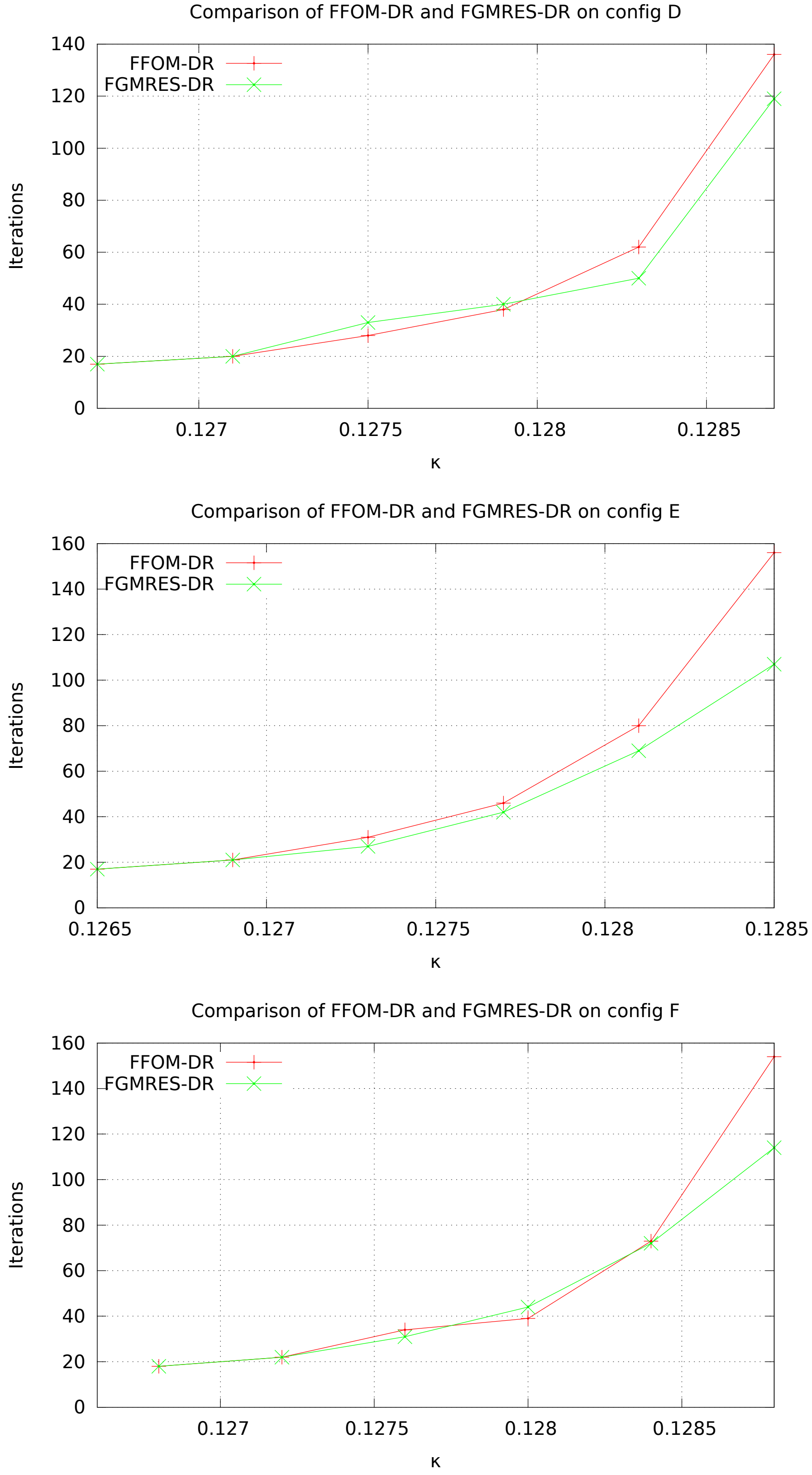


Figure 4.6.: FFOM-DR vs. FGMRES-DR on **config D**, **config E** and **config F** with different values of κ.

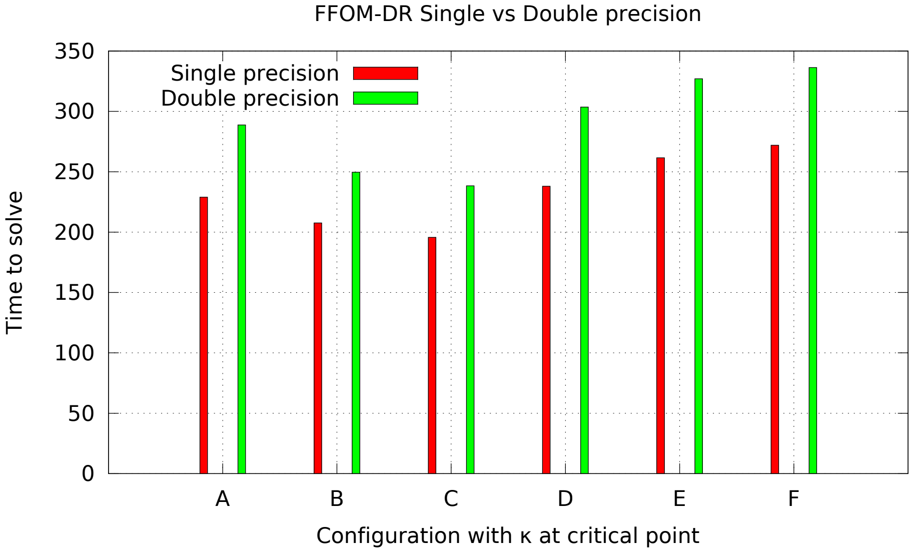


Figure 4.7.: Comparison of FFOM-DR in single vs double precision of all configurations with $\kappa = \kappa_c$ at each configuration.

by t_F, is

$$t_F = (1.05 \pm 0.03)t_G. \tag{4.2}$$

This is approximately five percent more than the FGMRES-DR time t_G. In the special case κ_c we even measured

$$t_F = (1.28 \pm 0.06)t_G. \tag{4.3}$$

Therefore we have to improve FFOM-DR in order to keep the strengths and get rid of the problems. We will now show benchmarks and tests executed for several versions of our implementation as outlined in the previous chapter.

We will discuss short evaluations of each step individually and then conclude with the full AOM algorithm as discussed in 3.6. The full AOM implementation will not only include all improvements, but will pick the best algorithm, i.e. FOM or GMRES, for the specific system.

4.3. Evaluation of each improvement

Converting the code to double precision did not have any impact on the iteration count of our test systems. However, the time spent in the algorithm, i.e. the seconds per iteration, did increase. This behavior was expected, since values stored in double precision have 64 instead of 32 bits and therefore need to use more registers on the CPU than their single

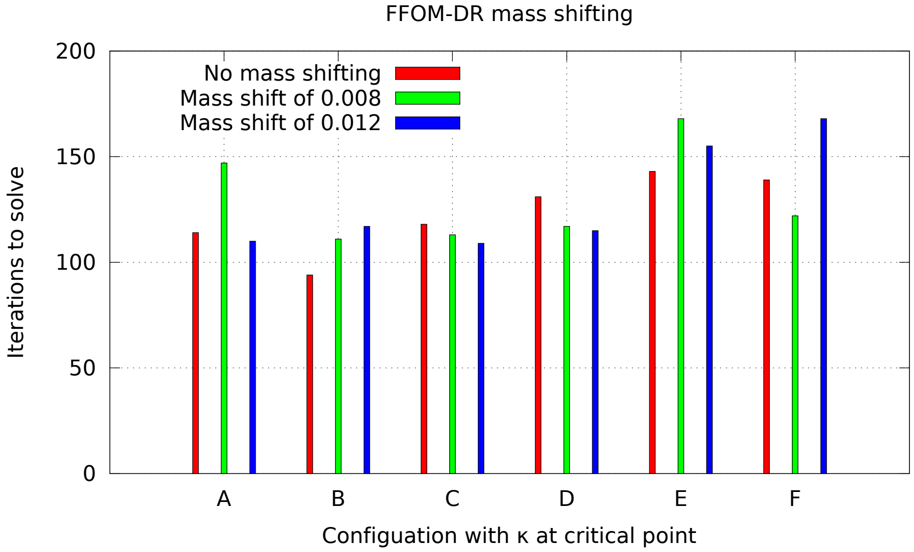


Figure 4.8.: Comparison of FFOM-DR with a mass shifting of 0, 0.008 and 0.012 at $\kappa = \kappa_c$ at each configuration.

precision counterparts. On average using the algorithm in double precision takes at least 20% longer than the single precision version.

The measurement of the code in single and double precision is shown in fig. 4.7. By comparing the iteration count we did not see any changes at all. This does not mean that our improvement is a failure, since we did not change the code from single to double precision in order to reduce iterations, but to avoid inconsistencies, make the code more robust and provide a solid basis for the next improvements. The two main features expected for this approach are the following:

1. Getting at least the same iteration count as before, if not better. This seems to be the case.
2. Not losing more than 50% of the time. Since our evaluations have been between 20% and 25% we have also achieved this goal.

The values of the evaluation can be found in tab. B.7 of appendix B.

Having a lower κ in the preconditioner just makes sense by using double precision in the solver. Therefore the evaluations concerning the usage of a lower κ in the preconditioner have been performed by FFOM-DR with enabled double precision. Also the mass shift just works with a different *BlkRel* parameter in the solver's configuration section of the Chroma input file. We will therefore only compare the algorithm using the *BlkRel* parameter with and without the mass shifting.

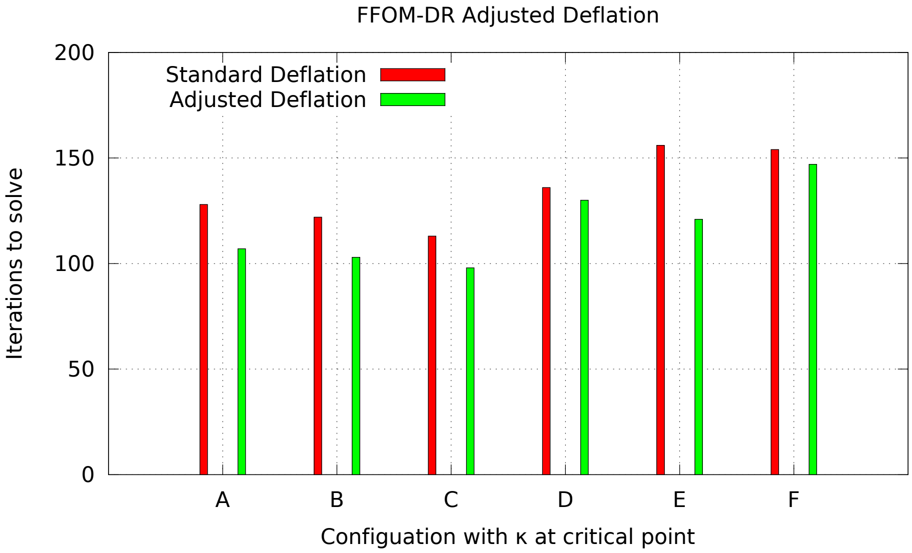


Figure 4.9.: Comparison of FFOM-DR with and without adjusted deflation of all configurations with $\kappa = \kappa_c$ at each configuration.

The evaluations of all systems at $\kappa = \kappa_c$ can be found in fig. 4.8. Here we just plot the number of iterations. Since the preconditioner works faster with a higher mass it is only necessary to see if we lose all that speed with a higher iteration count. Indeed this seems again like a gamble. However, it should be noted that the lower κ will boost GMRES more than FOM resulting in a more optimized version of GMRES than FOM. This is another reason for creating AOM. The complete data can be found in appendix B tab. B.8.

Comparing the versions with a mass shift of 0.012 and no mass shift we could assume that a mass shift between 0 and 0.012 like 0.006 saves some of the beneficial effects while removing most of the unwanted contributions. Nevertheless we can observe that this is only partially the case. Overall a lower mass shift made the result even more unpredictable with respect to the iteration count.

The data of the adjusted deflation evaluations can be found in tab. B.9. The important benchmarks are shown in fig. 4.9. We were able to gain up to 20% by implementing alg. 13 presented in section 3.5.3. Using this algorithm never increased the amount of iterations. However, it should be noted that this scenario is possible, even though it is not very likely and has not been observed in any measurement. The question what adjusted deflation will do for FGMRES-DR, which is the algorithm to pick for $\kappa = \kappa_c$ will be answered in the next section. For now we observe that the iteration count benefits greatly from using the adjusted deflation algorithm. Also the time spent with solving the system decreased.

A more detailed look at the residue of **config A** at $\kappa = \kappa_c$ is shown in fig. 4.10. Here we

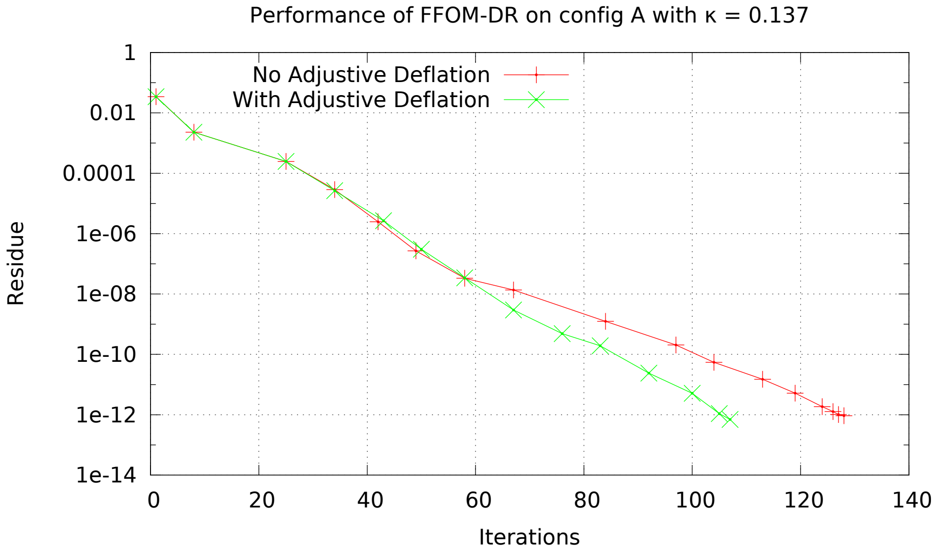


Figure 4.10.: Comparison of the residue in dependence of iterations from FFOM-DR with and without adjusted deflation of configuration A at $\kappa = \kappa_c$.

see that a main effect of the algorithm is indeed to keep the progress in a straight line in the semi logarithmic plot. The algorithm just kicks in once the Krylov subspace has grown large enough. Before this point it does not matter if the adjusted deflation algorithm is enabled or not - since the Krylov subspace size is still below twice the number of set up deflation vectors. After this point we can observe a noticeable difference between the two measurements. While the slope of the version with standard deflation becomes more flat the slope of the version with enhanced deflation remains about constant.

4.4. Evaluation of AOM

Combining all the improvements mentioned in section 4.3 results in the AOM algorithm, which has been introduced in section 3.6. The core of the algorithm is either GMRES or FOM, depending on the system. In order to make fair comparisons, we have to compare the best evaluation, which is either FGMRES-DR or FFOM-DR, depending on the system against the result of AOM on the same configuration. It is necessary to say that many lattice QCD codes just use the technique of iterative refinement due to many GPUs being much faster in single precision than in double precision. Here much faster means a factor of 10. However, already today's NVIDIA Fermi GPUs, and of course the future RISC based Intel MIC accelerators, feature a much smaller gap between single and double precision. Here the factor is about 2.

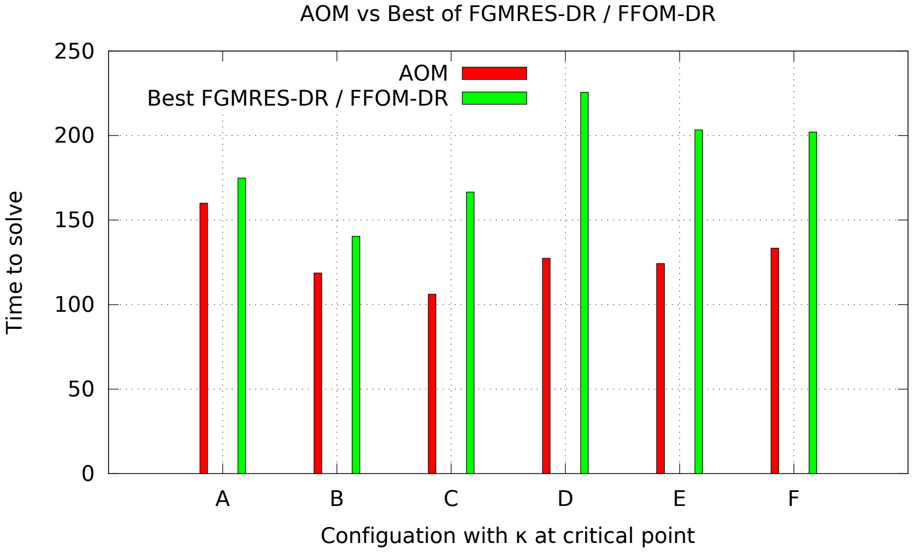


Figure 4.11.: Comparison of the AOM vs the best value of FGMRES-DR or FFOM-DR for each configuration at $\kappa = \kappa_c$.

The complete data set is shown in tab. B.10 in the appendix. For $\kappa = \kappa_c$ we obtain a plot like the one shown in fig. 4.11. Here we see that AOM offers us gains in the computation time ranging from about 10% to approximate 45%. The iteration count also drops by 20%-30%. All our modifications are now being beneficial to us since one of the greatest advancements in the implementation of AOM is the higher tolerance between restarts. Less restarts means less eigenvalue computation and greater eigenvalue accuracy.

If we have a look at the residue in dependence of the iteration count, which is shown in fig. 4.12, we see that the plot shows some differences to the one shown in fig. 4.3 of section 4.1. First of all the scale used for iterations is different, which indicates the better performance of AOM. Also the slopes seem to be optimized resulting in nearly straight lines in this logarithmic plot. The last and most important detail is the optimized order of iteration counts. Now the harder the problem is, the more iterations it requires. This is a significant difference to the data we acquired before, where the implementation proofed to be unstable in some regions. This instability was not only restricted to FOM, but could also be observed for GMRES.

The gained accuracy along with the more stable implementation is possible due to the switch from single to double precision. The lower computation time is mostly due to the lower kappa in the preconditioner as well as the implementation of the adjusted deflation algorithm. The optimized order is related to the switching feature of AOM, where the

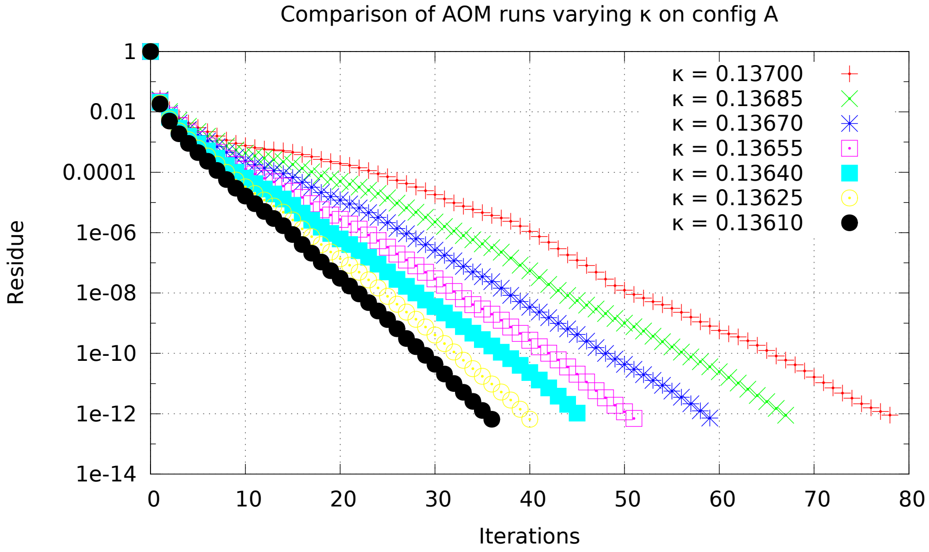


Figure 4.12.: Comparison of several AOM runs on **config A** using a different κ.

algorithm dynamically picks the best core algorithm - either FOM or GMRES.

4.5. Conclusion

With help of the benchmarks we were able to confirm that FOM can be faster than GMRES. Nevertheless the evaluations did also show that GMRES is the optimum choice in most cases as the core algorithm for any solver implementation. With our proposed changes we were able to make the existing algorithm more stable, while winning time and accuracy.

A main feature of AOM is the decision to take FOM or GMRES as a basis. This decision is now more or less empirical and has to be determined systematically. One way of doing this could be by using the Kalkreuter-Simma [KS96] algorithm for finding the lowest eigenvalue. The real part of the lowest eigenvalue will be approximately zero for $\kappa = \kappa_c$. Therefore a threshold for the real part of the lowest eigenvalue could be taken to set the core algorithm. The Kalkreuter-Simma algorithm itself is an extension of the CG method and is able to find the lowest eigenvalues by applying the Ritz-variational method.

In order to reduce the computational overhead for calculating the lowest eigenvalue just for the purpose of making a qualified decision whether to use FOM or GMRES as a basis one could propose the following: Using this (numerically exact up to a desired order) lowest eigenvalue as a solid criterion for the convergence of eigenvectors in the solver. Also bringing in one already converged eigenvector would give the code some additional boost.

Our proposed improvements have been successfully implemented and work fine so far. We did achieve a better performance than before and were able to include some new tweaks to the code like a bigger gap between two consecutive restarts. Those tweaks would not have been possible with the iterative refinement technique. Excluding this technique also helped us in gaining stability for the algorithm. We could successfully remove the sanity check and all other artifacts that have been placed to avoid any instability issues of the iterative refinement method.

We can summarize this work by one sentence:

> While FGMRES-DR is mostly better than FFOM-DR, AOM is the preferred choice in any case.

5. Summary

This thesis discussed possible enhancements of the latest GMRES based linear system solver for lattice QCD. We could confirm the suspected behavior of FOM being a better choice than GMRES for certain systems. We observed that this effect is mostly in regions that are not of great interest for current lattice QCD research. Other key aspects were the improvements to the current algorithm, FGMRES-DR. We started by replacing the core of the algorithm, GMRES, by FOM. Then we included some new techniques like the mass shift for the preconditioner or the adjusted deflation technique.

In order to remove odd side effects we turned off the iterative refinement and left only the preconditioner in single precision. Our final algorithm has to be able to decide between picking FOM or GMRES as a the core solver. Right now picking FOM as the core algorithm seems unlikely, however, no one knows what kind of systems or actions will be interesting in the future. Also this final algorithm known as AOM includes all of our improvements making it the fastest algorithm for solving systems in lattice QCD with the Wilson or improved Clover action.

Further research is necessary in order to even more improve the criterion that is used by AOM to distinguish between using GMRES or FOM. Previously the user was required to switch from GMRES to FOM manually. In the future this could be done by looking at the exact lowest eigenvalue of the system. This raises the question when the lowest eigenvalue is close enough to zero to pick GMRES. Therefore a certain threshold has to be determined. Calculating the lowest eigenmode could be done by using the Kalkreuter-Simma algorithm. Even though this calculation would be quite expensive it would be beneficial in multiple ways.

On the one hand it would provide an exact criterion for the base decision of AOM. On the other hand the vector could be used to have at least one fully converged eigenvector. Also since this eigenvalue happens to be the lowest one, the system would benefit greatly from having this lowest eigenvector fully deflated. The condition of the matrix would decrease significantly, which results in much less iterations required for full convergence.

Another possibility to determine the criterion without calculating expansive eigenvectors can be found in the research area of machine learning with data mining techniques [WFH11]. The research here would start by creating a number of different systems which are then solved by using first FOM and then GMRES as the core solver algorithm. The result will then be saved along with some properties of the system. After a very high number of trials one is able to get the probabilities and correlations for the number of properties that have been taken into account. Every new system will then be tested on those properties and the data mining algorithm will automatically pick the most suitable solver depending on the probabilities which have been created using the test data.

Therefore through the work provided within this thesis it is possible to solve systems in lattice QCD faster than before. However, further research could provide even better algorithms and implementations based on AOM and some techniques mentioned here. One crucial point for further improvements is the newly presented adjusted deflation algorithm.

Acknowledgements

This thesis has been written to obtain the title Master of Science in Physics. It never could have been achieved in a vacuum. This is my opportunity to thank all people who contributed directly or indirectly to this work.

For a lot of interesting talks and the great influence on my motivation I do have to thank Rudolf Rödl. For the fruitful conversations, lectures in scientific writing and informative discussions about programming I have to thank Simon Wolfgang Mages. For correcting stupid sentences and pointing out typos I have to say thank you so much to Simone. Thank you as well for your patience!

This thesis is part of the research in the lattice Quantum Chromo Dynamics group of Tilo Wettig. Thank you for your support and for letting me be part of the great research done there. The biggest external contribution to this work was certainly done by my supervisor Andrea Nobile. Thank you for giving me an introduction to the current solver algorithm and helping me with my problems. I wish you all the best for your work in Jülich!

A. Configuration data

	config A	**config B**	**config C**	**config D**	**config E**	**config F**
type	wilson	wilson	wilson	wilson	wilson	wilson
smearing	*none*	stout	stout	stout	stout	stout
n_x	16	16	16	16	16	16
n_y	16	16	16	16	16	16
n_z	16	16	16	16	16	16
n_t	32	32	32	32	32	32
n_f	2	2+1	2+1	2+1	2+1	2+1
β	5.29	5.5	5.5	5.5	5.5	5.5
c_{sw}	1.91	2.65	2.65	2.65	2.65	2.65
κ	0.1355	0.1203	0.1203	0.1203	0.1203	0.1203
κ_c	0.137	0.1283	0.1285	0.1287	0.1285	0.1288

Table A.1.: Detailled list of configurations used for the benchmarks

B. Evaluation data

	FGMRES-DR		**FFOM-DR**	
κ	Iterations	Time [sec]	Iterations	Time [sec]
0.1355	32	53.050193	34	55.992649
0.13565	31	52.00161	37	59.919848
0.1358	36	59.560919	41	65.3308
0.13595	44	75.016602	44	70.45242
0.1361	42	70.366178	51	82.355866
0.13625	47	78.768025	65	107.057678
0.1364	64	104.105807	81	131.25352
0.13655	66	113.342451	61	104.132427
0.1367	93	167.641954	88	149.336266
0.13685	117	209.659023	105	183.404195
0.1370	99	174.851978	128	228.9527

Table B.1.: FGMRES-DR vs. FFOM-DR on **config A** with various values for κ

	FGMRES-DR		**FFOM-DR**	
κ	Iterations	Time [sec]	Iterations	Time [sec]
0.1263	16	25.573457	16	25.772947
0.1267	19	31.120361	19	30.117445
0.1271	25	41.284375	23	35.944375
0.1275	32	53.969522	41	67.12183
0.1279	52	84.430224	63	102.903589
0.1283	78	140.368451	122	207.634141

Table B.2.: FGMRES-DR vs. FFOM-DR on **config B** with various values for κ

	FGMRES-DR		**FFOM-DR**	
κ	Iterations	Time [sec]	Iterations	Time [sec]
0.1265	16	26.199628	16	25.116805
0.1269	20	32.120435	20	32.271459
0.1273	27	44.838726	28	45.184219
0.1277	38	64.068725	40	63.992296
0.1281	58	98.406527	63	100.882443
0.1285	95	166.541027	113	195.645708

Table B.3.: FGMRES-DR vs. FFOM-DR on **config C** with various values for κ

	FGMRES-DR		**FFOM-DR**	
κ	Iterations	Time [sec]	Iterations	Time [sec]
0.1267	17	28.524003	17	27.613844
0.1271	20	32.16795	20	31.80881
0.1275	33	56.452038	28	44.515734
0.1279	40	67.679241	38	61.761515
0.1283	50	84.679794	62	101.0093
0.1287	119	225.570019	136	238.015617

Table B.4.: FGMRES-DR vs. FFOM-DR on **config D** with various values for κ

	FGMRES-DR		**FFOM-DR**	
κ	Iterations	Time [sec]	Iterations	Time [sec]
0.1265	17	29.463513	17	27.610692
0.1269	21	35.862582	21	33.553901
0.1273	27	48.716739	31	49.855214
0.1277	42	75.534817	46	75.946358
0.1281	69	126.399418	80	130.888645
0.1285	107	203.357508	156	261.553935

Table B.5.: FGMRES-DR vs. FFOM-DR on **config E** with various values for κ

	FGMRES-DR		**FFOM-DR**	
κ	Iterations	Time [sec]	Iterations	Time [sec]
0.1268	18	31.764543	18	28.954031
0.1272	22	38.799835	22	34.194324
0.1276	31	59.298134	34	55.5352
0.1280	44	80.630753	39	63.997939
0.1284	72	117.534111	73	122.608217
0.1288	114	202.097065	154	271.971618

Table B.6.: FGMRES-DR vs. FFOM-DR on **config F** with various values for κ

	Single		**Double**	
System	Iterations	Time [sec]	Iterations	Time [sec]
A	128	228.9527	128	288.733842
B	122	207.634141	122	249.588725
C	113	195.645708	113	238.360825
D	136	238.015617	136	303.60973
E	156	261.553935	156	327.042477
F	154	271.971618	154	336.292265

Table B.7.: FFOM-DR single vs double precision on all configurations with $\kappa = \kappa_c$ on each

	Iterations		
System	$\Delta m = 0$	$\Delta m = 0.008$	$\Delta m = 0.012$
A	114	147	110
B	94	111	117
C	118	113	109
D	131	117	115
E	143	168	155
F	139	122	168

Table B.8.: FFOM-DR without mass shifting, with a mass shift of 0.012 and with a mass shift of 0.008 at $\kappa = \kappa_c$ of each configuration

	Standard		**Adjusted**	
System	Iterations	Time [sec]	Iterations	Time [sec]
A	128	228.9527	107	196.841384
B	122	207.634141	103	187.199551
C	113	195.645708	98	182.873322
D	136	238.015617	130	233.683992
E	156	261.553935	121	222.741473
F	154	271.971618	147	257.536326

Table B.9.: FFOM-DR in the original version and with adjusted deflation enabled at $\kappa = \kappa_c$ of each configuration

			Before		**AOM**	
System	κ	Alg.	Iterations	Time [sec]	Iterations	Time [sec]
A	0.1361	G	42	70.366178	36	57.424816
A	0.13625	G	47	78.768025	40	60.030578
A	0.1364	G	64	104.105807	46	68.378785
A	0.13655	F	61	104.132427	51	77.187692
A	0.1367	F	88	149.336266	59	89.243123
A	0.13685	G	105	183.404195	67	107.437217
A	0.1370	G	99	174.851978	78	160.016859
B	0.1271	F	23	35.944375	24	34.002675
B	0.1283	G	78	140.368451	68	118.668585
C	0.1277	G	38	64.068725	32	44.671319
C	0.1285	G	95	166.541027	66	106.140469
D	0.1275	F	28	44.515734	27	37.642651
D	0.1287	G	119	225.570019	84	127.353491
E	0.1277	G	42	75.534817	35	49.691306
E	0.1285	G	107	203.357508	83	124.20852
F	0.1280	F	39	63.997939	37	53.967403
F	0.1288	G	114	202.097065	91	133.389631

Table B.10.: Comparison of AOM against the best values of FGMRES-DR or FFOM-DR on several configurations with the best algorithm denoted by G for FGMRES-DR and F for FFOM-DR

C. Miscellaneous

Snippet of the XML configuration file used for performing the evaluations with parameters \$KAPPA for the κ of the configuration, \$CSW for the c_{sw} of the configuration and \$BLKREL for the block relations (either 0 or 1):

```xml
<?xml version="1.0"?>
<chroma>
  <Param>
    <InlineMeasurements>
    <elem>
      <Param>
      <!-- Other parameters -->
        <InvertParam>
          <invType>DDS_INVERTER</invType>
          <Kappa>\$KAPPA</Kappa>
          <Csw>\$CSW</Csw>
          <Nmr>5</Nmr>
          <Ncy>8</Ncy>
          <Nkv>30</Nkv>
          <DeflatedNV>8</DeflatedNV>
          <BlockSize>4 4 4 8</BlockSize>
          <RsdDDS>1.0e-12</RsdDDS>
          <MaxDDS>3000</MaxDDS>
          <BlkRel>\$BLKREL</BlkRel>
        </InvertParam>
      </Param>
    </elem>
    <!-- Other elements -->
    </InlineMeasurements>
  </Param>
  <!-- Other configuration elements -->
  <!-- Setting the (external) configuration file -->
</chroma>
```

Short MATLAB program used to evaluate the performance of a direct (LU) vs an iterative (GMRES with restarts) solver:

```
N = [4 32 128 256 512 1024 2048 4096 8192];
iter = length(N);
tol = 1e-7;
T = zeros(iter, 4);
for i = 1:iter
    Ni = N(i);
    T(i, 1) = Ni;
    A = rh(Ni); % Random SPD matrix
    b = eye(Ni, 1);
    x0 = zeros(Ni, 1);
    restart = min(30, Ni);
    % Direct solver (LU)
    tic
    xlu = gauss(A, b, 'par');
    %xlu = lux(A, b);
    T(i, 2) = toc;
    % Iterative solver (GMRES(k))
    tic
    xgk = gmres(A, b, restart, tol);
    T(i, 4) = toc;
end
```

List of Algorithms

1. The Conjugate Gradient Method 15
2. Arnoldi's method 16
3. j-th Givens rotation 18
4. q-step shifted QR 19
5. Implicitly Restarted Arnoldi (IRA) 20
6. Full Orthogonalized Method (FOM) 21
7. Basic GMRES method 30
8. GMRES(k) method 32
9. GMRES with deflated restarts 33
10. Flexible Arnoldi process 35
11. Iterative refinement 35

12. Flexible FOM with deflated restarts and mixed precision 53
13. Adjusted deflation 59

List of Figures

1.1. Already small lattices produce matrices which are too big for print. 6

2.1. Transistors are nowadays spent on more cores instead of higher frequencies. 7
2.2. The paradigm of parallel computing is splitting the problem into several tasks. 8
2.3. Comparison of a direct (LU) vs an iterative (GMRES with restart) method using MATLAB (for details see appendix C). 9
2.4. As memory size (purple) increases the access time (red) increases as well. . 14
2.5. Solving a PDE on a certain region with a boundary, a so called domain. . . 24
2.6. Finding a proper domain decomposition into equal Ω_i is a difficult task. . . 25
2.7. The GMRES algorithm uses a lot of different techniques. 28
2.8. As far as we know today quarks are the smallest building blocks of matter (taken from the Nobel Price in Physics 2008 press release [oS08]). 36

3.1. The main layers of the Chroma software package including some implementation examples. 44
3.2. Our approach of extending Chroma with another inverter. 45
3.3. Usually it is not possible to achieve perfect concurrency as shown in a). Unbalanced load as shown in b), a kind of latency in c) or overhead due to communication as shown in d) are common. 48
3.4. FGMRES-DR is a direct ancestor of FGMRES and GMRES-DR, which share the common base of GMRES together with other algorithms like GCR. 49
3.5. Trying out various simple and empirical variations of determining the number of deflation vectors at one configuration motivates the creation of the adjusted deflation algorithm. 57
3.6. Scheme of adjusted deflation in a run with four restarts with the current progress p, the number of iterations I, the proposed number of deflation vectors v_s, the lowest number of deflation vectors L and the actually taken deflation vectors v_t. 60

3.7. AOM combines both algorithms and includes all the improvements, which have been implemented in FFOM-DR previously in order to form an optimized solver for lattice QCD. 62

4.1. Performance of FGMRES-DR on **config A** with different values of κ. . . . 66
4.2. Performance of FFOM-DR on **config A** with different values of κ. 67
4.3. Comparison of several FFOM-DR runs on **config A** using a different κ. . . 69
4.4. Eigenvalue spectrum of FFOM-DR on **config A** using $\kappa = 0.13655$ shown in the first plot and $\kappa = 0.13685$ shown below. 70
4.5. FFOM-DR vs. FGMRES-DR on **config A**, **config B** and **config C** with different values of κ. 71
4.6. FFOM-DR vs. FGMRES-DR on **config D**, **config E** and **config F** with different values of κ. 72
4.7. Comparison of FFOM-DR in single vs double precision of all configurations with $\kappa = \kappa_c$ at each configuration. 73
4.8. Comparison of FFOM-DR with a mass shifting of 0, 0.008 and 0.012 at $\kappa = \kappa_c$ at each configuration. 74
4.9. Comparison of FFOM-DR with and without adjusted deflation of all configurations with $\kappa = \kappa_c$ at each configuration. 75
4.10. Comparison of the residue in dependence of iterations from FFOM-DR with and without adjusted deflation of configuration A at $\kappa = \kappa_c$. 76
4.11. Comparison of the AOM vs the best value of FGMRES-DR or FFOM-DR for each configuration at $\kappa = \kappa_c$. 77
4.12. Comparison of several AOM runs on **config A** using a different κ. 78

List of Tables

4.1. List of configurations used for the benchmarks and the parameters of their production 65
4.2. FGMRES-DR vs. FFOM-DR on **config A** with various values for κ 68

A.1. Detailled list of configurations used for the benchmarks 85

B.1. FGMRES-DR vs. FFOM-DR on **config A** with various values for κ 87
B.2. FGMRES-DR vs. FFOM-DR on **config B** with various values for κ 87
B.3. FGMRES-DR vs. FFOM-DR on **config C** with various values for κ 88
B.4. FGMRES-DR vs. FFOM-DR on **config D** with various values for κ 88
B.5. FGMRES-DR vs. FFOM-DR on **config E** with various values for κ 88
B.6. FGMRES-DR vs. FFOM-DR on **config F** with various values for κ 89
B.7. FFOM-DR single vs double precision on all configurations with $\kappa = \kappa_c$ on each 89
B.8. FFOM-DR without mass shifting, with a mass shift of 0.012 and with a mass shift of 0.008 at $\kappa = \kappa_c$ of each configuration 89
B.9. FFOM-DR in the original version and with adjusted deflation enabled at $\kappa = \kappa_c$ of each configuration 90
B.10. Comparison of AOM against the best values of FGMRES-DR or FFOM-DR on several configurations with the best algorithm denoted by G for FGMRES-DR and F for FFOM-DR 90

Bibliography

[Arn51] W. E. Arnoldi. The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quart. Appl. Math*, 9:17–29, 1951.

[Bau57] F. L. Bauer. Das Verfahren der Treppeniteration und verwandte Verfahren zur Lösung algebraischer Eigenwertprobleme. *Zeitschrift für Angewandte Mathematik und Physik (ZAMP)*, 8:214–235, 1957. 10.1007/BF01600502.

[BDJ06] A. H. Baker, J. M. Dennis, and E. R. Jessup. An efficient block variant of GMRES. *SIAM J. Sci. Comput.*, 27(5):1608–1626, May 2006.

[BJM05] A. H. Baker, E. R. Jessup, and T. Manteuffel. A Technique for Accelerating the Convergence of Restarted GMRES. *SIAM J. Matrix Anal. Appl.*, 26(4):962–984, April 2005.

[Bro91] P. N. Brown. A theoretical comparison of the Arnoldi and GMRES algorithms. *SIAM J. Sci. Statist. Comput.*, 12(1):58–78, jan 1991.

[CG96] J. Cullum and A. Greenbaum. Relations between Galerkin and norm-minimizing iterative methods for solving linear systems. *SIAM J. Matrix Anal. Appl.*, 17(1):223–247, jan 1996.

[Col11] Chroma Collaboration. The Chroma Library for Lattice Field Theory. *http://usqcd.jlab.org/usqcd-docs/chroma/*, oct 2011.

[DD06] T. DeGrand and C. DeTar. *Lattice Methods For Quantum Chromodynamics.* World Scientific Publishing, Singapore, 1 edition, 2006.

[DKPR87] S. Duane, A. D. Kennedy, B. J. Pendleton, and D. Roweth. Hybrid Monte Carlo. *Phys. Lett. B*, 195(2):216, 1987.

[EJ05] R. G. Edwards and B. Joó. The Chroma Soft ware System for Lattice QCD. *Nucl. Phys B1*, 40:832, 2005.

[FFG$^+$96] S. Fischer, A. Frommer, U. Glaessner, Th. Lippert, G. Ritzenhoefer, and K. Schilling. A Parallel SSOR Preconditioner for Lattice QCD. *Comput. Phys. Commun.*, 98(1):20–34, April 1996.

[Fra61] J.G.F. Francis. The QR Transformation, I. *The Computer Journal*, 4(3):265–271, 1961.

[Fra62] J.G.F. Francis. The QR Transformation, II. *The Computer Journal*, 4(4):332–345, 1962.

[Giv58] W. Givens. Computation of plane unitary rotations transforming a general matrix to triangular form. *J. Soc. Indust. Appl. Math.*, 6(1):26–50, jan 1958.

[GL09] C. Gattringer and C.B. Lang. *Quantum Chromodynamics on the Lattice.* Springer, Berlin, 1 edition, 2009.

[Jas04] Leonhard Jaschke. *Preconditioned Arnoldi Methods for Systems of Nonlinear Equations.* Wiku, 2004.

[KS96] T. Kalkreuter and H. Simma. An Accelerated Conjugate Gradient Algorithm to Compute Low-Lying Eigenvalues - a Study for the Dirac Operator in SU(2) Lattice QCD. *Comput. Phys. Commun.*, 93:33–47, 1996.

[KY95] S. A. Kharchenko and A. Y. Yeremin. Eigenvalue translation based preconditioners for the GMRES(k) method. *Num. Lin. Alg. with Appl.*, 2:51–77, 1995.

[LS96] R. B. Lehoucq and D. C. Sorensen. Deflation Techniques for an Implicitly Restarted Arnoldi Iteration. *SIAM J. Matrix Anal. Appl.*, 17(4):789–821, October 1996.

[Lüs03] M. Lüscher. Lattice QCD and the Schwarz alternating procedure. *JHEP*, 2003(05):052, 2003.

[Lüs04a] M. Lüscher. Schwarz-preconditioned HMC algorithm for two-flavour lattice QCD. *CERN-PH-TH*, 2004:177, 2004.

[Lüs04b] M. Lüscher. Solution of the Dirac equation in lattice QCD using a domain decomposition method. *Comput. Phys. Commun.*, 156(1):209–220, 2004.

[Lüs07a] M. Lüscher. Deflation acceleration of lattice QCD simulations. *JHEP*, 12:011, 2007.

[Lüs07b] M. Lüscher. Local coherence and deflation of the low quark modes in lattice QCD. *JHEP*, 07:081, 2007.

[Mag11] S. Mages. Hadronic Spectral Functions in the QCD Transition Region - Scale Setting and Anisotropy Parameters for Lattice QCD Simulations. Master's thesis, University of Regensburg, oct 2011.

[Mor02] R. B. Morgan. GMRES With Deflated Restarting. *SIAM J. Sci. Comput.*, 24(1):20–37, 2002.

[MS10] M. Marinkovic and S. Schaefer. Comparison of the mass preconditioned HMC and the DD-HMC algorithm for two-flavour QCD. *PoS LATTICE*, 2010:7, 2010.

[MW02] R. B. Morgan and W. Wilcox. Deflation of Eigenvalues for GMRES in Lattice QCD. *Nucl. Phys. B (Proc. Suppl.)*, 106:1067–1069, 2002.

[NZF] A. Nobile, P. Zigler, and A. Frommer. FGMRES-DR. In preparation.

[OBB$^+$10] J. C. Osborn, R. Babich, J. Brannick, R. C. Brower, M. A. Clark, S. D. Cohen, and C. Rebbi. Multigrid solver for clover fermions. *PoS LATTICE*, 2010:7, 2010.

[oS08] The Royal Swedish Academy of Sciences. The 2008 Nobel Prize in Physics - Press Release. pages 1–8, 2008.

[PPV95] C. C. Paige, B. N. Parlett, and H. A. Van der Vorst. Approximate solutions and eigenvalue bounds from Krylov subspaces. *Num. Lin. Alg. with Appl.*, 2(2):115–133, feb 1995.

[PSM$^+$06] M. L. Parks, E. D. Sturler, G. Mackey, D. D. Johnson, and S. Maiti. Recycling Krylov Subspaces for Sequences of Linear Systems. *SIAM J. Sci. Comput.*, 28(5):1651–1674, 2006.

[RD10] F. Rappl and T. DeGrand. Extending the CG routine of a lattice QCD code to perform deflation. *http://www.florian-rappl.de/?i=forschung/deflation*, apr 2010.

[Saa92] Y. Saad. *Numerical Methods for Large Eigenvalue Problems.* Halsted Press, New York, NY, 1992.

[Saa96] Y. Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing, Boston, MA, 1996.

[SBG04] B. Smith, P. Bjorstad, and W. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*, volume 1. Cambridge University Press, Cambridge, 1 edition, April 2004.

[Sch70] H. A. Schwarz. Über einen Grenzübergang durch alternierendes Verfahren. *Vierteljahrsschrift der Naturforschenden Gesellschaft in Zürich*, 15:272–286, 1870.

[SS86] Y. Saad and M. H. Schultz. GMRES: a generalized minimum residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput.*, 7:856–869, 1986.

[SSS86] R. T. Scalettar, D. J. Scalapino, and R. L. Sugar. New algorithm for the numerical simulation of fermions. *Phys. Rev. B*, 34(11):7911, 1986.

[SSW98] Y. Saad, A. Stathopoulos, and K. Wu. Dynamic Thick Restarting of the Davidson, and the Implicitly Restarted Arnoldi Methods. *SIAM J. Sci. Comput.*, 19(1):227–245, January 1998.

[SV96] G. L. G. Sleijpen and H. A. Van der Vorst. A Jacobi-Davidson Iteration Method for Linear Eigenvalue Problems. *SIAM J. Matrix Anal. Appl.*, 17(2):401–425, apr 1996.

[vMPG29] R. von Mises and H. Pollaczek-Geiringer. Praktische Verfahren der Gleichungsauflösung. *Zeit. angew. Math. Mech.*, 9:152–164, sep 1929.

[WFH11] I. H. Witten, E. Frank, and M. A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques.* Morgan Kaufmann, Burlington, 3 edition, 2011.

[WS00] K. Wu and H. Simon. Thick-restart Lanczos method for symmetric eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 22:602–616, 2000.

[ZN05] L. Zhang and T. Nodera. A new adaptive restart for GMRES(m) method. *ANZIAM J.*, 46(1):409–425, may 2005.

Erklärung

Eidesstattliche Erklärung zur Masterarbeit

Name: Rappl,
Vorname: Florian,
Geburtstag: 09.09.1984,
Geburtsort: Regensburg.

Ich versichere, die Masterarbeit selbständig und lediglich unter Benutzung der angegebenen Quellen und Hilfsmittel verfasst zu haben.

Ich erkläre weiterhin, dass die vorliegende Arbeit noch nicht im Rahmen eines anderen Prüfungsverfahrens eingereicht wurde.

Regensburg, den 10.4.2012. ....

(Unterschrift)