Report of Independent Studies

Performed under the supervision of Tom DeGrand

Florian Rappl, Student ID 100-580-287

Spring 2011

Extending the CG routine of a lattice QCD code to perform deflation

Florian Rappl

Department of High Energy Physics, University of Colorado at Boulder, 80309, CO, USA and Institut für Theoretische Physik, Universität Regensburg, D-93040 Regensburg, Germany

with help and instructions from Tom DeGrand

Department of High Energy Physics, University of Colorado at Boulder, 80309, CO, USA

(Dated: April 26, 2011)

Abstract

This paper is the final report of the Independent Studies under supervision of Professor Tom DeGrand at the University of Colorado. The topic deals with extensions in form of deflation performed on the MilcI6 code in the research field of lattice Quantum Chromodynamics. The task in this Spring Term 2011 for the Independent Studies was to get in touch with a state of the art lattice QCD code as well as doing some extensions and measurements with it and therefore making some conclusions about the effectiveness of those alternations.

The extensions of the code were done in order to implement a deflation technique. In the first step new functions are called before the conjugate gradient routine is executed, i.e. using the original CG routine. In the next step the CG routine has been modified in order to support approximate eigenmodes.

Another technique that was being used in order to achieve this was a Schur decomposition of the matrix which has to be inverted using the CG routine. Finally a lookup matrix that contains all the information of a little Dirac operator had to be calculated and then inverted.

I. INTRODUCTION

The most successful approach to the theory of Quantum Chromodynamics (QCD) is lattice QCD. This is mainly due to the strong coupling in QCD which makes pertubation theory useless for accurate predications by calculating a doable amount of terms. Another reason is the improvement in computation power as well as the availibility of low priced and effective cluster systems.

A huge step for modern lattice QCD simulations was the introduction of the so called Hybrid Monte-Carlo (HMC) algorithm, which has been published by Gottlieb et al.[6] after the invention and extensions to HMC by Scaletter et al.[8] and Duane et al.[7]. This algorithm is a combination of Molecular Dynamics and a normal Monte-Carlo algorithm as it is described by Thomas DeGrand[1]. Even though state of the art HMC algorithms are highly specialized for their calculation purposes a lot of routines and computations can still be improved or optimized. In the last years a lot of effort has been powered into research for better or improved conjugate gradient routines. The main cost of a QCD program lies often in the calculation of $\psi = M^{-1}\eta$. Since we are not able to calculate this exactly (or would suffer from the huge cost of this calculation) we use a conjugate gradient (CG) routine to solve $M\psi = \eta$ with a source vector η and a solution vector ψ.

In the recent research we are able to see a lot of (combined) algorithms in order to improve the standard CG routine. We see a so called biconjugate gradient (*BiCG*), a stabilized bigconjugate gradient (*BiCGstab*) and other popular choices in many current lattice QCD codes. Another set of questions can be raised in form of which preconditioner to pick. One possible choice would be the mass preconditioner introduced by Hasenbusch[5] and used by Wenger[4]. Having the appropriate preconditioner for a specific configuration is a topic that has already been discussed in a paper by Schaefer[13]. Also other authors like Rebbi[14] tried to come up with some solution to minimize the cost of this computation part lattice QCD codes.

However one of the problems is that even though a new method might work faster (and more stable) on a certain configuration, it might work slower on a different configuration. In this paper we try to make improvements on the (standard) conjugate gradient routine by using a deflation with (approximate) eigenmodes in a Schur-decomposition method. We will look at the advantages and disadvantages and will work on the required modifications of the Milc code.

II. DYNAMICAL CLOVER APPLICATION OF THE MILC CODE

The dynamical clover application was placed the in folder `clover_dyn_test` and is quite similar to the original one that is placed in `clover_dynamical`. The application deals with dynamical clover fermions and includes a general gauge action. As with the other applications we have several build methods integrated in the makefile such as a refreshed molecular dynamics algorithm, phi algorithm as well as hybrid Monte Carlo algorithm. We always build the later one, including LU preconditioning. This changes the dynamical fermion matrix to be the LU preconditioned matrix, on even sites only. It is important to know that the conjugate gradient routine for this application is contained in the file `d_congrad2_cl_field.c` in the `generic_clover` directory.
The workflow for our extensions looks like this:

1. Write a routine to test if the existing code is working properly.
2. Make corrections to the code if necessary.
3. Start with a simple deflation just modifying the wrapper function - not the conjugate gradient routine.
4. Run several tests and evaluate the outcome of those.
5. Generalize the deflation in order to support even not exact eigenmodes.
6. Extend the wrapper as well as the conjugate gradient routine to support approximate eigenmodes.
7. Replace the exact eigenmode finder with an approximate but faster one.
8. Run several tests and evaluate the outcome of those.

A list of directories, header files and important macros is given in MILC documentation[9]. The Schrödinger Functional (SF) (which is not used by this application, but is part of the MILC Code) is described in more detail in Tom DeGrand's notes[10].
As a remark it is important to know that all evaluations have been performed in the single processor mode of the clover dynamical application. Since we need more communication with deflation than without deflation another set of evaluations concerning the MPI performance of the code with and without deflation is required in order to make predictions about (practical) effectiveness. It is also important to know that we did not implement optimizations in the new routines. Therefore our evaluations of the deflation represent the lower boundary of the performance curve.

III. THEORY OF THIS SPECIFIC DEFLATION METHOD

This topic has been brought up by Lüscher[11] and has since then be discussed a lot by most lattice QCD research groups. According to Lüscher[12] we could gain a lot of speed if we implement this kind of deflation correctly.

A. The basic deflation

We start with an equation to be solved that reads (in Dirac notation)

$$H|\psi\rangle = |\eta\rangle, \tag{1}$$

where $|\psi\rangle$ is the state that we search and H as well as $|\eta\rangle$ are given. In our code we have $H \equiv M^\dagger M$. If we find eigenmodes for this H, we get for each of those k eigenmodes

$$H|k\rangle = \lambda_k |k\rangle. \tag{2}$$

Those eigenmodes built a basis that we can use. Since we suppose that we did not compute all eigenmodes, which is very expensive and obsolete since we then would not have to call the conjugate gradient routine, we have only part of a basis. Therefore every state can be deflated in a part which already lies in this basis and a part which is kind of a residue. For our states we write

$$\begin{aligned} |\psi\rangle &= |\psi_L\rangle + |\psi_H\rangle = \sum_k c_k |k\rangle + |\psi_H\rangle, & (3)\\ |\eta\rangle &= |\eta_L\rangle + |\eta_H\rangle = \sum_k |k\rangle\langle k|\eta\rangle + |\eta_H\rangle. & (4) \end{aligned}$$

If we take again eq. 2 and perform it on eq. 3 instead of $|k\rangle$ we see

$$H|\psi\rangle = H|\psi_H\rangle + \sum_k \lambda_k c_k |k\rangle \overset{!}{=} |\eta_H\rangle + \sum_k |k\rangle\langle k|\eta\rangle, \tag{5}$$

which gives us (due to orthogonality) the coefficient

$$c_k = 1/\lambda_k \langle k|\eta\rangle. \tag{6}$$

This calculation already reduces eq. 1 to

$$H|\psi_H\rangle = |\eta_H\rangle. \tag{7}$$

Using eq. 3 the full state $|\psi\rangle$ can be restored from the original $|\eta\rangle$ and the k eigenmodes. We can rewrite this basic deflation in a matrix form, where H is a diagonal matrix with submatrices A

and X on the diagonal. Our solution can therefore be obtained by

$$\begin{pmatrix} \psi_H \\ \psi_L \end{pmatrix} = \begin{pmatrix} A^{-1} & 0 \\ 0 & X^{-1} \end{pmatrix} \begin{pmatrix} \eta_H \\ \eta_L \end{pmatrix}. \tag{8}$$

The form of the submatrix A can be determined using eq. 2, eq. 7 and eq. 5 obtaining

$$A = H - P_L H P_L = P_H H P_H, \tag{9}$$

when we define

$$P_H \equiv (1 - P_L), \qquad P_L \equiv \sum_k |k\rangle\langle k|, \tag{10}$$

which is true in a diagonal basis, i.e. if we assume to work with exact eigenmodes.

B. The Schur decomposition

However since we already mentioned that accurate eigenmodes are expensive to compute we will have to deal with approximate eigenmodes. In this case our matrix H will not be diagonal but in a more general form

$$H = \begin{pmatrix} P_H H P_H & P_H H P_L \\ P_L H P_H & P_L H P_L \end{pmatrix} = \begin{pmatrix} A & B \\ C & X \end{pmatrix}. \tag{11}$$

The trick now is to see that if the number of eigenvectors is small we are actually able to invert the submatrix X exactly and with low computation cost. This is based on the so called Schur decomposition. We can rewrite H to be

$$\begin{aligned} H &= \begin{pmatrix} A & B \\ C & X \end{pmatrix} = \\ &= \begin{pmatrix} 1 & BX^{-1} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} A - BX^{-1}C & 0 \\ 0 & X \end{pmatrix} \begin{pmatrix} 1 & 0 \\ X^{-1}C & 1 \end{pmatrix}. \end{aligned} \tag{12}$$

The inverse of this rewritten H is therefore given by inverting the matrices in eq. 12,

$$H^{-1} = \begin{pmatrix} 1 & 0 \\ -X^{-1}C & 1 \end{pmatrix} \begin{pmatrix} (A - BX^{-1}C)^{-1} & 0 \\ 0 & X^{-1} \end{pmatrix} \begin{pmatrix} 1 & -BX^{-1} \\ 0 & 1 \end{pmatrix}. \tag{13}$$

With this rewritten H we can go back to solve our basic equation by computing the right hand side of eq. 8 directly with a non-diagonal matrix, i.e. the Schur decomposition (eq. 13),

$$H^{-1} \begin{pmatrix} \eta_H \\ \eta_L \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ -X^{-1}C & 1 \end{pmatrix} \begin{pmatrix} (A - BX^{-1}C)^{-1} & 0 \\ 0 & X^{-1} \end{pmatrix} \begin{pmatrix} \eta_H - BX^{-1}\eta_L \\ \eta_L \end{pmatrix}. \tag{14}$$

We now introduce some new (intermediate) states which we define as

$$\begin{aligned}\eta_+ &\equiv \eta_H - BX^{-1}\eta_L \\ \eta_- &\equiv \eta_L. \end{aligned} \tag{15}$$

Using this we perform another matrix multiplication and obtain

$$H^{-1}\begin{pmatrix}\eta_H \\ \eta_L\end{pmatrix} = \begin{pmatrix} 1 & 0 \\ X^{-1}C & 1\end{pmatrix}\begin{pmatrix}(A - BX^{-1}C)^{-1}\eta_+ \\ X^{-1}\eta_-\end{pmatrix}. \tag{16}$$

Doing another redefinition as in eq. 15 we define the states on the right as

$$\begin{aligned}\Phi_1 &\equiv (A - BX^{-1}C)^{-1}\eta_+ = \\ &= (A - BX^{-1}C)^{-1}(\eta_H - BX^{-1}\eta_L) \\ \Phi_2 &\equiv X^{-1}\eta_- = X^{-1}\eta_L. \end{aligned} \tag{17}$$

Therefore through the redefinition in eq. 17 the result of our equation has become

$$H^{-1}\begin{pmatrix}\eta_H \\ \eta_L\end{pmatrix} = \begin{pmatrix}\Phi_1 \\ \Phi_2 - X^{-1}C\Phi_1\end{pmatrix} \tag{18}$$

If we can capture the true low eigenmodes in X the submatrix $A - BX^{-1}C$ will be better conditioned and therefore work better than the original conjugate gradient call without this deflation. If this requirement is not met, the method may not work well, which is why this deflation method has either to be implemented as a program execution option or may only be active if a certain condition in the program is satisfied.

IV. EXTENDING THE CODE

Before extending the code some tests had to be done in order to verify the correctness of the matrix to be solved and deflated. The test should verify that the matrix the eigenmode solver is working with is the same matrix that the conjugate gradient routine is applied to. The eigenmodes will be calculated using the trick which has been introduced by Kalkreuter[2].

A. Tests and corrections before the implementation

In order to perform this verification a simple piece of program code has been added in the `congrad_cl_wrapper()` routine of the application, right before the conjugate gradient routine is called.

```
void EquivalenceTest()
{
  int j;
  double_complex lambda;
  wilson_vector *tmp = (wilson_vector*)malloc(sites_on_node * sizeof(wilson_vector));
  for(j = 0; j < Nvecs_h0; j++)
  {
    Matrix_Vec_mult(eigVec0[j], tmp);         /* computes M_adjoint * M * j = tmp */
    dot_product(eigVec0[j], tmp, &lambda);   /* computes j_adjoint * tmp          */
    /* prints value of this computation and lambda_j (value as it should be)      */
    printf("Eigenvalue %d is %f\t|\tTestvalue %d is %f\n", j + 1, eigVal0[j], j + 1, lambda.real);
  }
}
```

The output of this little subroutine shows us if the computed eigenvalue (using the current matrix and multiplying it with the j-th eigenvector and its adjoint), which is named *testvalue* and stored in the variable `lambda` is the same as the original eigenvalue, which is stored in the variable `eigVal0[j]`.

With this simple test we could see that there were some important function calls missing, which resulted in a different matrix which has been used by the CG routine than the one used for calculating the eigenmodes. The required additions were some `make_clovinv(ODD)` calls, which sometimes were missing after `make_clov(CKU0)` was called. In this case `ODD` is a macro that just assigns the value of **1** and `CKU0` declared as a real number with the value

$$CKU0 \equiv \kappa \cdot (clov_c)/(u_0^3).$$

After the missing function calls were implemented the output showed the same values at each CG call and each trajectory calculation. Since we were now sure to work with the right matrix we could go one step further and try to implement our deflation method.

B. The basic deflation

1. *Extensions to the code*

The basic deflation has been implemented directly in the `congrad_cl_wrapper()` routine of the program. Two new functions have been introduced, which have been called

`ProjectResidueVector()` as well as `RestoreFullVector()`. Those functions perform

$$
\begin{aligned}
|\eta_H\rangle &= |\eta\rangle - \sum_k c_k \cdot |k\rangle, \qquad (19)\\
c_k &= \langle k|\eta\rangle,
\end{aligned}
$$

(using eq. 4) as well as

$$
\begin{aligned}
|\psi\rangle &= |\psi_H\rangle + \sum_k c_k \cdot |k\rangle, \qquad (20)\\
c_k &= 1/\lambda_k \cdot \langle k|\eta\rangle, \qquad (21)
\end{aligned}
$$

analog to eq. 3. The `congrad_cl_wrapper()` routine is now extended from the simple CG routine call to a function that makes the deflation of η, then calls CG routine with the deflated vector η_H and finally does the restoration of the full ψ. The program code therefore reads:

```
int iters;
register int i;
register site *s;
wilson_vector *chi = (wilson_vector*)malloc(sites_on_node * sizeof(wilson_vector));
build_h0(1); /* restart eigenvectors */
/* save the old chi for rest. the full psi */
FORALLSITES(i,s)
  copy_wvec(&(s->chi), &(chi[i]));
ProjectResidueVector(); /* Project out chi to chi_r */
/* Normal wrapper but now solves: M_adjoint * M * psi_r = chi_r */
iters = congrad_cl(niter, rsqmin, final_rsq_ptr);
FORALLSITES(i,s) /* Restore the original chi */
  copy_wvec(&chi[i], &(s->chi));
RestoreFullVector(); /* Bring psi_r to psi */
return(iters);
```

Those implementations have been checked carefully and tested on some test configurations with 4^4 and 8^4 lattices.

2. Analysis of the performance after the extension

After having checked the code for problems by calling the CG routine twice and comparing output results with the same starting configuration and input parameters for a deflated and a non-deflated application we could do some analysis of the implemented deflation method. For all performance analysis runs we started with the same starting configuration and input parameters (except the number of eigenmodes to compute - since this variation was the crucial part of the analysis) and computed one trajectory.

For this computation a lattice was chosen which has a volume of 8^4 and pretty realistic (and

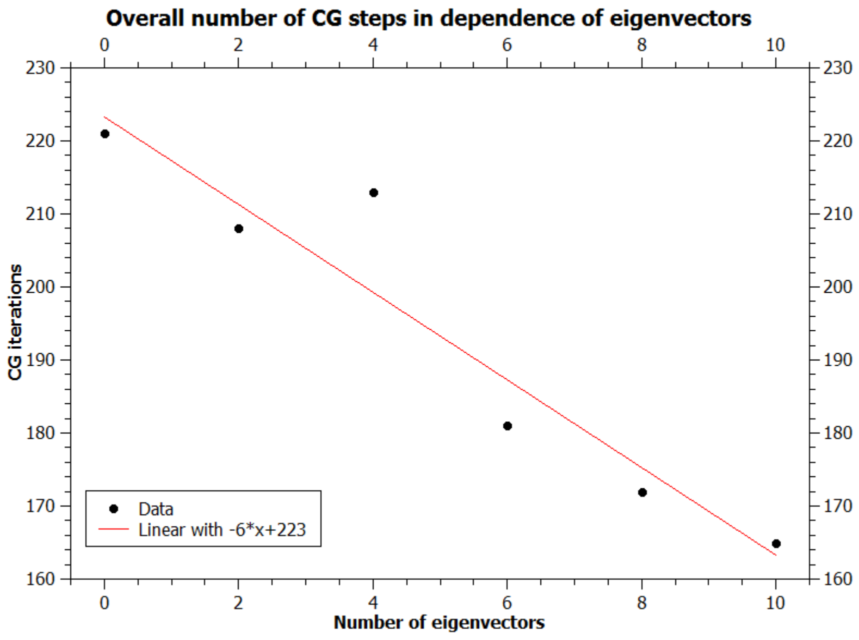

FIG. 1: The overall number of CG iterations required for computing one trajectory

therefore practical) parameters. The parameters are listed in detail in the appendix section. In fig. 1 we see a confirmation of the theory. With the parameters and the lattice chosen for this configuration we achieved a better performance for the conjugate gradient routine. This can also be seen by viewing the residue $r^\dagger r$ in dependence of the CG time (iteration).

The plot shown in fig. 2 is exactly what we were looking for. We see that the condition of our matrix is better due to the removal of the low eigenmodes from the solution vector. Therefore the state we were searching for was easier to find for the conjugate gradient routine. We also see that finding more eigenmodes is actually helpful but not required since we do not benefit from finding those in form of a reduced computation time.

We are able to see this problem in fig. 3. While the application with no deflation active and zero eigenmodes to compute takes the least time but has the most conjugate gradient steps, the time won by the deflated applications due to less CG steps is more than just eaten up by the computation of the low frequency eigenmodes. Therefore we need to work with approximate eigenmodes and we do need to make some decomposition to our matrix $H \equiv M^\dagger M$.

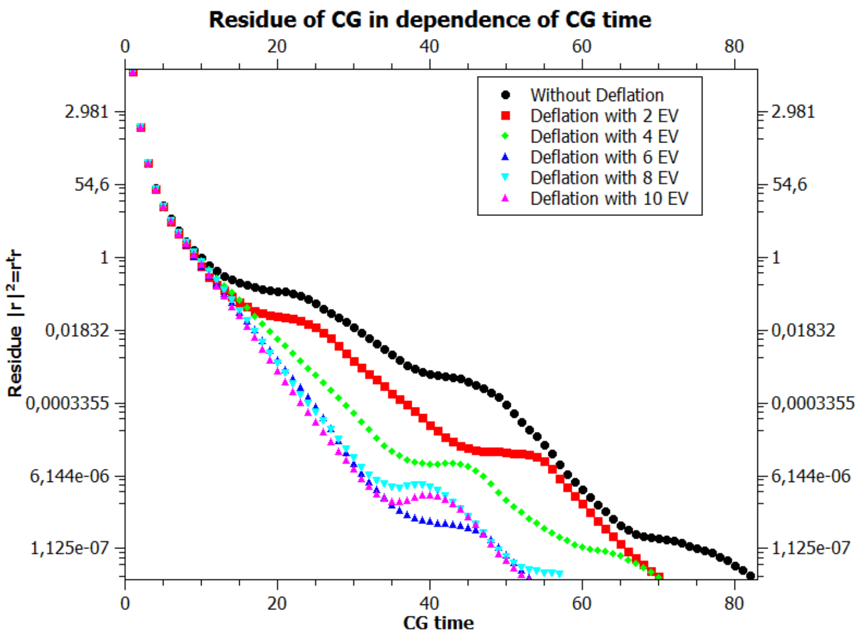

FIG. 2: The residue of the CG routine in dependence of the iterations for the first CG call

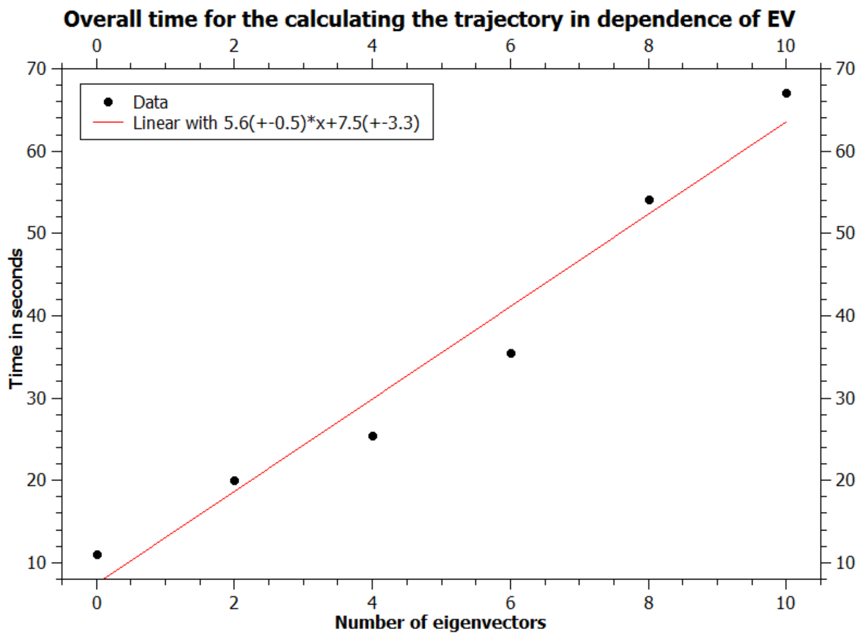

FIG. 3: The time needed for the whole application to compute the first trajectory

C. The Schur decomposition

1. *Generalizing the algorithm*

From eq. 18, the equation of the Schur decomposition theory, and the redefinitions we are able to determine an algorithm for the modifications required in the program. We have to change the routine for the conjugate gradient - especially those parts where the matrix is used on the state. We have to write three new subroutines, which perform the following tasks:

- Project a mode $|m\rangle$ (given in form of a parameter) with P_L, i.e. $\sum_k |k\rangle\langle k|m\rangle$ and return the projected mode.
- A routine to calculate and invert $\langle k|H|k'\rangle$. The resulting $N \times N$ matrix, where N is the number of (approximate) eigenmodes, will be saved for further usage.
- Project a mode $|m\rangle$ (again given as a parameter) with the calculated matrix values from the routine above, i.e. $\sum_{k,k'}(\langle k|H|k'\rangle)^{-1}|k\rangle\langle k'|m\rangle$, and return the projected mode.

Additionally we outsource the calculation of $H|m\rangle$ that is performed on several parts in the function. Therefore we just have to use the new (outsourced) routine with the proper parameters instead of a copy of the same routine with different variables over and over. All these new subroutines are motivated by the following calculation. We start from

$$\left(A - BX^{-1}C\right)|\psi_H\rangle = |\eta_H\rangle - BX^{-1}|\eta_L\rangle, \tag{22}$$

$$\Rightarrow \qquad \tilde{H}\psi_H = \tilde{\eta}, \tag{23}$$

and insert the definitions of A, B, C, X. Using identities like $P_L P_L = P_L$ and $1|m\rangle = |m\rangle$ we are able to rewrite eq. 22 with a special bracket notation to

$$[|\psi_H\rangle - \{|\psi_H\rangle\}] \equiv |\psi_1\rangle, \tag{24}$$

$$\psi_1 - [(|\psi_1\rangle)] \equiv |\psi_2\rangle, \tag{25}$$

$$[(|\eta_L\rangle)] \equiv |\eta_1\rangle, \tag{26}$$

$$\Rightarrow \qquad \psi_2 - \{|\psi_2\rangle\} \overset{!}{=} |\eta_H\rangle - \eta_1 + \{|\eta_1\rangle\}, \tag{27}$$

where we set the special notation to be:

- $[\cdots]$ donates the use of H (existing, outsourced routine) on the state,
- $\{\cdots\}$ donates the use of P_L (new routine) on the state and

- $(\cdots)$ donates the use of $(P_L H P_L)^{-1}$ (new routine) (i.e. including $(\langle k|H|k'\rangle)^{-1}$) on the state.

The algorithm is now performed like this:

1. We split $|\eta\rangle = |\eta_H\rangle + |\eta_L\rangle$ as before.

2. We use the conjugate gradient routine to solve a subsystem in the form of eq. 22 as outlined with eq. 27 using the two new subroutines.

3. We take the solution $|\psi_H\rangle$ and get $|\psi_L\rangle$ using (this is **n**o bracket notation!)

$$|\psi_L\rangle = \sum_{k,k'} |k\rangle \frac{1}{\langle k|H|k'\rangle} \left(\langle k'|\eta_L\rangle - \langle k'|H \left[|\psi_H\rangle - \sum_{k''} |k''\rangle\langle k''|\psi_H\rangle \right] \right). \tag{28}$$

4. Now the full solution can be obtained using the already known

$$|\psi\rangle = |\psi_H\rangle + |\psi_L\rangle. \tag{29}$$

For a first precision and algorithm test we excluded the calculation of $\langle k|H|k'\rangle$ and its inverse and used $\delta_{k,k'}/\lambda_k$ instead. After the code was sure to be working correctly we implemented another set of functions. Those functions performed the following tasks:

- Calculating the matrix elements of $\langle k|H|k'\rangle$, which is an $N \times N$ matrix, where N is the number of (approximate) eigenvectors.

- Inverting this matrix and therefore getting $(\langle k|H|k'\rangle)^{-1}$.

We stored these computed values in a global two-dimensional double array, so that the function which requires these values can easily access them. For a first precision test we worked with our exact eigenvalues and therefore could see that the program works correctly.

2. Extensions to the code

To have a much easier life it was necessary to change the definition of the type definition `site` in the program. Therefore the file `lattice.h` has been edited. With the help of our `DEFLATION` macro we added the following code to the header file:

```
#ifdef DEFLATION
  wilson_vector psi_high; /* solution vector high mode */
  wilson_vector psi_low;  /* solution vector low mode */
  wilson_vector chi_high; /* source vector high mode */
  wilson_vector chi_low;  /* source vector low mode */
#endif
```

The function `congrad_cl_wrapper()` therefore changed, i.e. the function did not require the temporary wilson vectors after this change. Also the functions `ProjectResuidueVector()` and `RestoreFullVector()` did change in this matter - both are now using the new site variables.
To make life simpler again another build type has been added in the `Maketemplate` file. The macro that we introduced to switch on the deflation code has been outsourced into a special build instructed by that makefile. The mode to build the deflation application is been called by using `"su3_hmc_def"` as make argument. This new build also uses a new version of the `d_congrad2_cl_field.c` file. The new file is called `d_congrad2_cl_def.c` and contains several code changes that will be explained in the next lines.
The code in the function `congrad_cl()` already contained some repetitions of code lines. Since we now needed some of those code lines even more often (like applying the matrix on some wilson vector) we just outsourced those lines and set appropriate parameters. After all we added or outsourced the following functions:

- `project()` - projects the source vector on the space spanned by the (approximate) eigenmodes.

- `xinv()` - applies the operator X^{-1} on the source vector $|s\rangle$, i.e.

$$X^{-1}|s\rangle = (P_L H P_L)^{-1}|s\rangle = \sum_{j,k} |j\rangle \left(\langle j|H|k\rangle\right)^{-1} \langle k|s\rangle. \tag{30}$$

- `perform_h()` - executes the operator H on the source vector.

- `perform()` - performs the whole left side of our new CG equation as shown in eq. 22 and as listed in algorithm form in eq. 27. This has to be done in every CG iteration.

- `source()` - performs the whole right side of our new CG equation as shown in eq. 22. Unlike the left side this only has to be computed at the beginning of the CG routine.

- `build_lookup()` - builds the matrix with the elements $\langle k|H|k'\rangle$ and then calls the function to create the actual inverse matrix out of this.

- `build_inverse()` - takes the input matrix and inverts it to obtain $(\langle k|H|k'\rangle)^{-1}$.

The `xinv()` function is kind of interesting since it does not only project it once but twice and (since we have approximate eigenmodes $\langle k|k'\rangle \neq \delta_{k,k'}$ in general) does multiplies by the inverse matrix elements. Here we have:

```
register int i;
register site *s;
int j, k;
complex cd, ctmp, css;
/* Make sure destination vector is empty */
FORMYSITESDOMAIN(i,s)
  clear_wvec(&(dst[i]));
/* this is the outer loop - |k><k| */
for(k = 0; k < Nvecs_h0; k++)
{
  css.real = (Real)0; css.imag = (Real)0;
  /* this is the inner loop - |k'><k'| */
  for(j = 0; j < Nvecs_h0; j++)
  {
    cd.real = (Real)0; cd.imag = (Real)0;
    /* get <j|src> */
    FORMYSITESDOMAIN(i,s)
    {
      ctmp = wvec_dot(&(eigVec0[j][i]), &(src[i]));
      CSUM(cd, ctmp);
    }
    /* multiply with <k|H|j>^-1 */
    CMULREAL(cd, inv[k][j], ctmp); CSUM(css, ctmp);
  }
  /* and make it a vector again |k> * 1/<k|H|j> * <j|src> */
  FORMYSITESDOMAIN(i,s)
    c_scalar_mult_add_wvec(dst + i, &(eigVec0[k][i]), &css, dst + i);
}
```

Another interesting function is the `source()` routine. Here we see how the algorithm from the RHS of eq. 27 can be applied in code. Basicly the same style of code had to be written for the routine `perform()`. The difference next to the order of function calls is in the length of that function. The code for changing η to $\tilde{\eta}$ from eq. 23 looks like:

```c
register int i;
register site *s;
wilson_vector *chi2, *chi1, *tmp;
/* allocate fields = malloc calls */
FIELD_ALLOC(chi2, wilson_vector)
FIELD_ALLOC(chi1, wilson_vector)
FIELD_ALLOC(tmp, wilson_vector)
/* P_L ( H ( X^-1 |eta_L> ) ) */
xinv(chi_low, tmp); perform_h(tmp, chi1); project(chi1, tmp);
/* |eta_H> - [ P_L ( H ( X^-1 |eta_L> ) ) - H ( X^-1 |eta_L> ) ] */
FORMYSITESDOMAIN(i,s)
{
  sub_wilson_vector(chi1 + i, tmp + i, chi2 + i);
  sub_wilson_vector(&(s->chi_high), chi2 + i, chi_new + i);
}
/*Cleaning up*/
free(tmp); free(chi1); free(chi2);
```

The functions to build the lookup table for the inverse matrix elements are straight forward. We compute all N^2 possibilites of $\langle i|H|j\rangle$ and store it in a two dimensional array. Then we start the matrix inverter which basicly solves $\langle i|H|j\rangle = \delta_{ij}$. We assume a non-singular matrix but we do not assume a good ordering for performing the inversion. Therefore we built in a pivot-search to always shuffle the rows in a way that we have the best possible accuracy.

3. *Analysis of the performance after the extension*

After all the extensions were implemented we had to run several tests and evaluate the results obtained in those. A first test was to check if we get the same results as with the specialized deflation method, i.e. that we use eigenmodes that have the same accuracy. In fig. 4 we can see that the graph shows basicly the same behavior (especially with zero eigenmodes we get the same result as with deflation off - which is the way it should be). However we note some differences when we compare this graph to fig. 2.

We see that the conjugate gradient routines seems more optimized than before, because the slope of the residue is really straight and does not stall. The difference is remarkable for more than

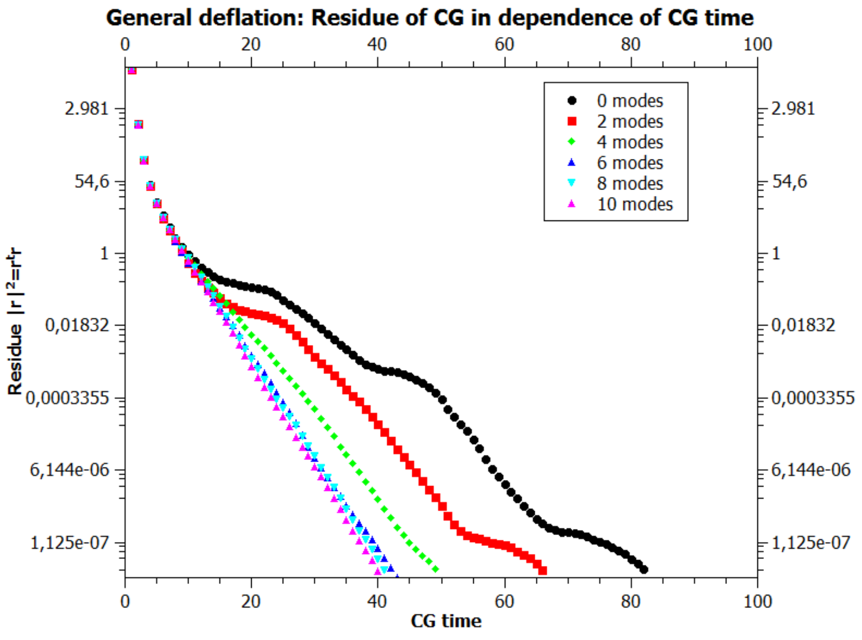

FIG. 4: The residue of the general CG routine in dependence of the iterations at first call

two eigenmodes. After the correctness of our implementation is verified we wanted to answer the question when deflation is really useful. To answer this we had to investigate how many eigenmodes we have to set and which accuracy those eigenmodes have to fulfill in order to work properly.

Fig. 5 shows a comparison of different runs with two eigenmodes each with a different level

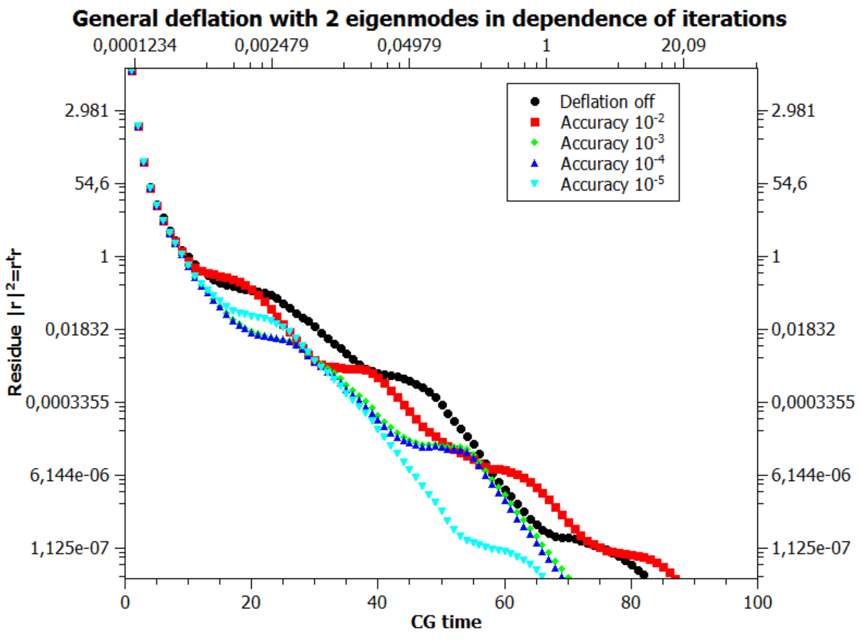

FIG. 5: Comparison shows which accuracy is required for two eigenmodes to decrease iterations

of accuracy. We see that only $10^{-}5$ exact eigenmodes fulfill the minimum requirement of never coming really close (i.e. touching) the black line which indicates the result that would have been achieved without deflation.

While two eigenmodes seem to need high accuracy in order to work properly we can see that four

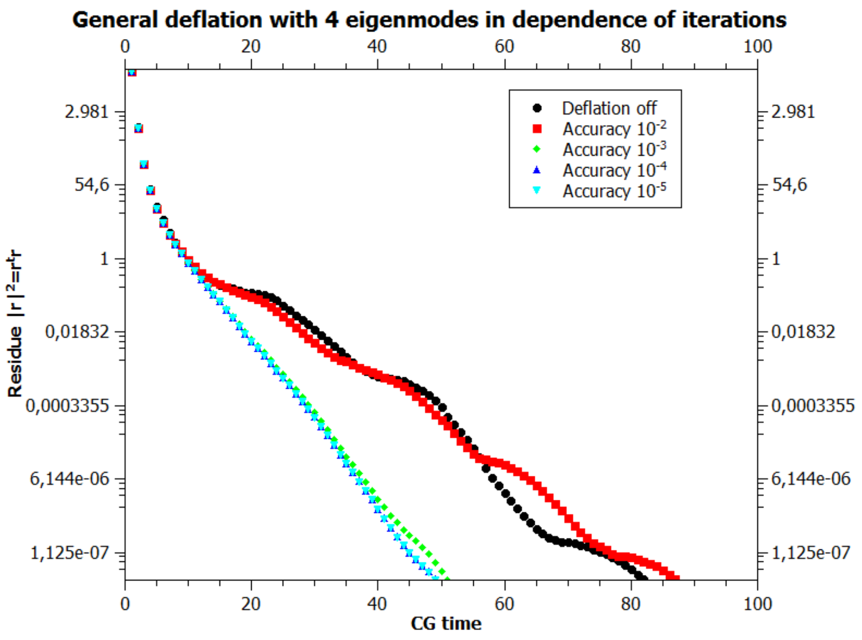

FIG. 6: Comparison shows which accuracy is required for four eigenmodes to decrease iterations

eigenmodes do a quite good job in decreasing iterations. Here it does not really matter if we take $10^{-}3$ or $10^{-}5$ as tolerance level for the eigenmode finder. However we do also see that a tolerance of $10^{-}2$ would be to low. Since there is a big jump in the iteration count from $10^{-}2$ to $10^{-}3$ eigenmodes we have to find the best accuracy in between if we want to work with four eigenmodes. The same conclusion can be drawn for 6, 8 or 10 eigenmodes. We always see that $10^{-}2$ is too low and that $10^{-}3$ to $10^{-}5$ have all the same slope in our $|r|^2(i)$ diagram. The next question that then arises is to see what we are actually solving. We know that we solve $H\psi = \eta$ but since we changed the CG routine in many ways we actually solved $\tilde{H}\psi_H = \tilde{\eta}$. Recalling the original CG routine that solved $H\psi = \eta$ instead of

$$f_L(H)\psi_H = f_R(\eta_H, \eta_L), \tag{31}$$

can give us an answer to that question: how far is the obtained solution away from the real solution? Fig. 7 shows basicly the same diagram as done in our first analysis of the general deflation with fig. 4. We now see that a low amount of eigenmodes (in this case two) returns a pretty accurate solution while an higher amount of eigenmodes tends to be inaccurate. We also do see that this

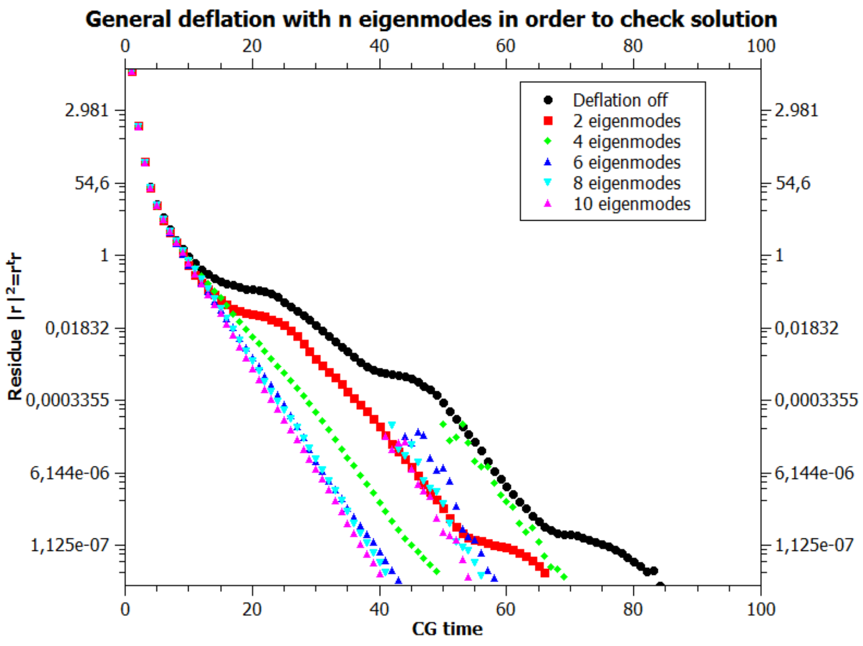

FIG. 7: The residue when calling the original CG routine after obtaining the solution with deflation

system again behaves in a way binary. Before we've seen that from 10^-2 to 10^-3 accuracy we have a huge jump in the iteration count, while 10^-3 to 10^-5 did not show any major differences. Here we have the same with the accuracy of the solution. No deflation and two eigenmodes behave pretty accurate while four eigenmodes and above need a certain amount of iterations with the original CG routine in order to satisfy the inbuilt accuracy condition of the CG function. The height of the jump from the end of the modified CG call to the beginning of the original CG is also pretty much the same for 4 to 10 eigenmodes.

Until now we've seen that for two eigenmodes we need high accuracy, while for four eigenmodes the obtained solution might need some fixing. The right answer must therefore lie between those two settings. In order to find the right balance we have to vision two very important data sets that are displayed in fig. 8. In the left plot we see the iteration count when we first called the conjugate gradient function for a different number of eigenmodes (x-axis) and different tolerance levels (labels). The right plot shows the iterations that could be computed per second - taking the same two variables (number of eigenmodes and tolerance level) into consideration.

We see that a tolerance level of 10^{-2} is too low for any number of eigenmodes. We also do see that the decrease of the effectiveness is quite linear in all cases, whereas the decrease of iterations does slow down. This problem is therefore hard to balance. We need on the one hand a decent decrease of the number of total iterations on the one side and a quite low decrease (maybe no decrease at all or even an increase) in the effective iterations per second. We made a further set of evaluations

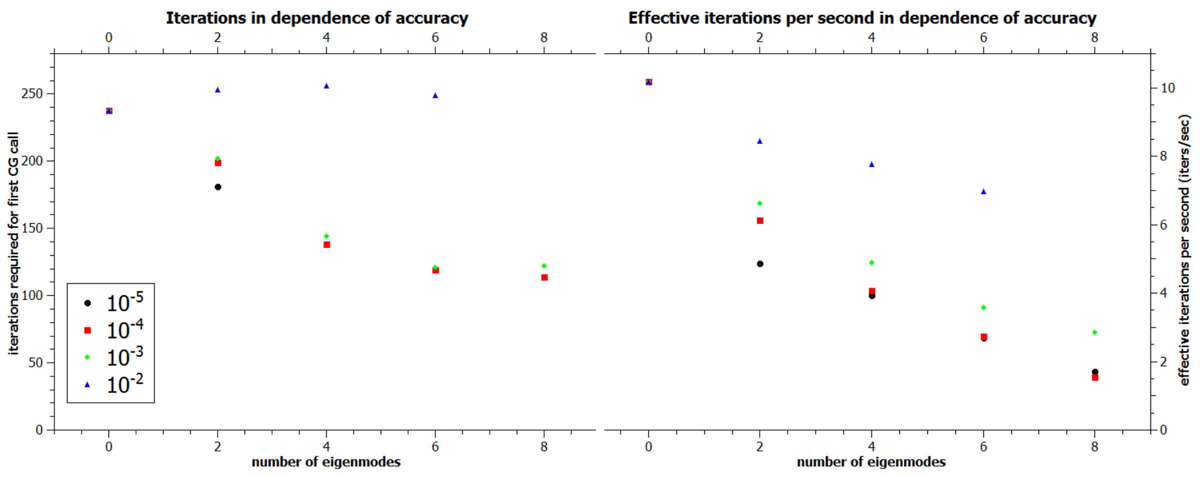

FIG. 8: The iterations and effective iterations per second in dependence of accuracy and eigenmodes

containing the interesting number of eigenmodes (which obviously lies between two and four, i.e. three eigenmodes) as well as the interesting region of tolerance between 10^{-2} and 10^{-4}. With this (improved) deflation algorithm we could reach the performance level of the standard conjugate gradient level. We found that three eigenmodes combined with an accuracy of 10^{-3} only requires 169 iterations in total (for one trajectory, i.e. three conjugate gradient calls) compared to 238 iterations for zero eigenmodes (no deflation). The time was a just a little bit slower with a value of 25.92953 s compared to 23.39351 s of the standard conjugate gradient. This was the best result since both, lower and higher accuracy required more time and higher accuracy did not result in sufficiently less iterations. The complete data can be found in tab. I in the appendix setion.

V. CONCLUSION

The deflation method discussed in this final report is certainly a good way to improve the conjugate gradient routine. For this paper no optimizations in the deflation routines were implemented. With optimizations in those new routines and a less expensive (but therefore less accurate) eigenmode finder it will certainly be possible to get a performance boost. However the deflation has also some negative sides. We need an eigenmode finder (which is expensive) and will probably get an inaccurate solution, which will force us to call a standard CG routine in order to fix this. That we came already close (without optimizations) to the performance of the previous CG execution time should be enough motivation in order to spend more research time on this topic.

Acknowledgments

I want to thank Tom DeGrand for the opportunity to work in this field and with an actual lattice QCD code as well as helping me out with answers to my questions and helpful advice regarding effective treatments of facing the problems that have been discussed in this paper. Also to have another set of eyes for debugging was really helpful. I learned a lot in those two semesters and always felt welcome with my questions and concerns. Thanks for the nice and interesting time!

[1] T. DeGrand and C. DeTar, "Lattice Methods for Quantum Chromodynamics," World Scientific Publishing, Singapore, ISBN **981-256-727-5**, (2006).

[2] 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), (1996) [hep-lat/9507023].

[3] T. Takaishi and P. de Forcrand, "Testing and Tuning new Symplectic Integrators for Hybrid Monte Carlo Algorithm in lattice QCD," Phys. Rev. E **73**(036706), (2006) [arXiv:hep-lat/0505020].

[4] C. Urbach, K. Jansen, A. Shindler and U. Wenger, "HMC Algorithm with multiple Time Scale Integration and Mass Preconditioning," Comput. Phys. Commun. **174**(87), (2006) [arXiv:hep-lat/0506011].

[5] M. Hasenbusch, "Speeding up the Hybrid-Monte-Carlo Algorithm for Dynamical Fermions," Phys. Lett. B **519**(177), (2001) [arXiv:hep-lat/0107019].

[6] S. A. Gottlieb, W. Liu, D. Toussaint, R. L. Renken and R. L. Sugar, "Hybrid Molecular Dynamics Algorithms for the Numerical Simulation of Quantum Chromodynamics," Phys. Rev. D **35**(2531), (1987).

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

[8] R. T. Scalettar, D. J. Scalapino and R. L. Sugar, "New Algorithm for the Numerical Simulation of Fermions," Phys. Rev. B **34**, (1986).

[9] C. Bernard, T. Burch, T. DeGrand, C. DeTar and others, "The MILC Code," The MILC Collaboration **6.20sep02**, (2000).

[10] T. DeGrand, Y. Shamir and B. Svetitsky, "Notes for Higher-Irrep SF," Notepaper, (2010).

[11] M. Lüscher, "Local coherence and deflation of the low quark modes," JHEP **07**, (2007)081.

[12] M. Lüscher, "Deflation acceleration of lattice QCD simulations," JHEP **12**, (2007)011.

[13] M. Marinkovic and S. Schaefer, "Comparison of the mass preconditioned HMC and the DD-HMC algorithm for two flavor QCD," PoS **LAT2010**, (2010)031.

[14] 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 **LAT2010**, (2010)037.

Appendix I

The parameters used for evaluating the deflation

The parameters listed below represent the configuration that has been used for testing and evaluating the deflation method described in this paper. The used lattice (**b**5.4k0.1272.d) was created and provided by Tom DeGrand. The number of eigenmodes `Number_of_h0_eigenvals` as well as the accuracy `eigenval_tol_low` has been changed for some evaluations in order to provide information about the efficiency of the deflation.

```
prompt 0
nflavors 2
nx 8
ny 8
nz 8
nt 8
iseed 6721827
warms 0
trajecs 1
traj_between_meas 1
beta 5.4
kappa 0.1272
clov_c 1.0
u0 1.0
microcanonical_time_step 0.2
steps_per_trajectory 1
max_cg_iterations 500
max_cg_restarts 10
error_per_site 1.0e-6
error_for_propagator 1.0e-6
Number_of_h0_eigenvals 2
Max_Rayleigh_iters 1000
Restart_Rayleigh 10
Kalkreuter_iters 10
eigenval_tol_low 1.0e-5
error_decr_low .3
reload_serial b5.4k0.1272.d
forget
```

Appendix II

Finding the optimum setting for those parameters

The following data has been obtained modifying the eigenvalue tolerance level as well as the number of eigenmodes. The goal was to come as close as possible (or even surcome) to the performance level that the standard conjugate gradient routine shows.

#	accuracy	iterations	time	iterations/sec
0	-	238	23.39351	0.09829205882353
3	0.008	249	30.01168	0.1205288353414
3	0.006	242	29.21451	0.1207211157025
3	0.004	220	28.51674	0.1296215454545
3	0.002	186	26.82217	0.1442052150538
3	0.001	*169*	**25.92953**	*0.1534291715976*
3	0.0008	169	26.09702	0.1544202366864
3	0.0006	168	27.5887	0.164218452381
3	0.0004	168	27.97978	0.1665463095238
3	0.0002	167	30.86919	0.1848454491018
3	0.0001	**167**	29.74855	0.1781350299401

TABLE I: Obtained data in order to find the optimum settings when running a full trajectory

Appendix III

The new application - clover_dynamical_deflation

In order to keep on working with this code everything has been packed into a tarball and submitted as a new application. Most filechanges were just pure cosmetics. Those changes include fixing the tabspaces as well as including some comments and keeping everything in the same programming style. The list of bigger changes includes:

- `Maketemplate` / a new target has been included called *su3_hmc_def*. This target builds everything like the `su3_hmc` application except that another macro is included which is called `DEFLATION`. This macro activates the deflation mode in the code.
- `generic_clover.h` / the new macro has been used in order to prototype the new routines that are included in this application.
- `control.c` / some bug fixes that have been detected by the `EquivalenceTest()` function are included in this version.
- `lattice.h` / the new macro has been used in order to extend the node's variables with χ_L, χ_H as well as ψ_L, ψ_H.
- `congrad_cl_wrapper.c` / this code is now extended with some functions and does contain some new macros. With the help of these macros it is possible to recall the standard conjugate gradient function or print out a comparison of eigenmodes (`EquivalenceTest()`).
- `d_congrad2_cl_def.c` / the new conjugate gradient code is extendend with the new functions that provide the projection, performing the Hamiltonian, building the inverse lookup table and other functionalities. The call of the conjugate gradient function has been extended in order to still provide support for a standard CG call in order to make the recall test for a solution.