Master Thesis
Performance Modelling and Analysis of the openQxD Lattice QCD Application

Roman Gruber

ETH Zürich, TODO:date, TODO: supervior(s)

Abstract

TODO

Contents

---

1 Introduction

TODO

Proposal 1.1: example proposal

Reference here with pp:one.

In QCD blabla see proposal 1.1. orange, yellow, blue, brown, pink, red, green, purple, turquois, lightblue, lightgreen, lightpink, darkblue, lightblue, lightpink, lightgreen, linkcolor

The result of integrating $\int \sqrt{1+x}\,dx$ is given by $\frac{2(x+1)^{\frac{3}{2}}}{3}$
Python says "Hello!"

2 Conventions

3 QCD

TODO: non-abelian, hadronic physics, importance, renormalization problems, running coupling, pert theory in high enery phsyics, not in low energy regime

4 lattice QCD

TODO: as a renormalization scheme, observables, ... boundary conditions, specially SF type,

5 Performance Models

TODO: why are they important? semi-analytical, analytical vs. empritical models

6 Software: openQxD

the software package openQxD: description * importance of CG in openQxD and what it does / how it's used in the software / why 90% computation time

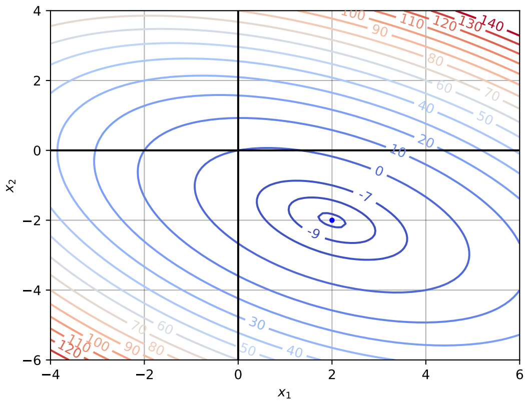

Figure 1: Quandratic form TODO

7 Conjugate Gradient algorithm

In many scientific computations large systems of linear equations need to be solved. Usually these systems are huge and the matrices and vectors are distributed among many ranks. The method to solve such systems should therefore be iteratively. The problem can be formulated mathematically in the following way. Let $n \in \mathbb{N}$ and let A be a $n \times n$-matrix with components in $\mathbb{C}$, Hermitian, positive definite and sparse

$$
\begin{aligned}
&& A^\dagger = A, && (\textbf{\textit{Hermitian}}) \\
\forall \vec{x} \in \mathbb{C}^n \setminus \{0\} \quad : && \vec{x}^\dagger A \vec{x} > 0, && (\textbf{\textit{positive definite}})
\end{aligned}
$$

as well as $\vec{b} \in \mathbb{C}^n$ be given, then the ***system of linear equations*** can be described as

$$
A\vec{x} = \vec{b}. \tag{7.1}
$$

We are interested in the ***solution*** vector $\vec{x}$, that is the one that satisfies the above equation, n is called the ***problem size***. First let us define a function that will be helpfull in the next sections.

Definition 7.1 (Quadratic form)**.** *The **quadratic form** depends on the problem matrix A as well as on the **source** vector $\vec{b}$ and is defined as*

$$
f(\vec{x}) = \frac{1}{2} \vec{x}^\dagger A \vec{x} - \vec{b}^\dagger \vec{x} + c,
$$

where $c \in \mathbb{C}$. When taking the derivative of this function with repect to $\vec{x}$, we find that

$$
f'(\vec{x}) = A\vec{x} - \vec{b}.
$$

Therefore finding the extrema of $f(\vec{x})$ is equivalent to solving the linear system of equations (7.1). The question whether the solution $\vec{x}$ is unique remains.

Lemma 7.1 (Uniqueness of the solution)**.** *The solution $\vec{x}$ in equation (7.1) is unique and the global minimum of $f(\vec{x})$ if A is Hermitian and positive definite* [1].

[1] Notice that, negative definiteness is sufficient as well and $\vec{x}$ would be the global maximum instead - just define $A' = -A$ which is positive definite and all of the argumentation that follows will hold as well. Indefinite matrices on the other hand might have local minima and maxima.

Proof. Let us rewrite $f(\vec{p})$ at an arbitrary point $\vec{p} \in \mathbb{C}$ in terms of the solution vector $\vec{x}$:

$$f(\vec{p}) = f(\vec{x}) + \frac{1}{2}(\vec{p} - \vec{x})^\dagger A(\vec{p} - \vec{x}). \tag{7.2}$$

This is indeed the same as $f(\vec{p})$ (inserting $A\vec{x} = \vec{b}$ and using $A^\dagger = A$ and of $\vec{a}^\dagger \vec{b} = \vec{b}^\dagger \vec{a}$),

$$\begin{aligned}
f(\vec{x}) + \frac{1}{2}(\vec{p} - \vec{x})^\dagger A(\vec{p} - \vec{x}) &= \frac{1}{2}\vec{x}^\dagger A\vec{x} - \vec{b}^\dagger \vec{x} + c + \frac{1}{2}\vec{p}^\dagger A\vec{p} - \frac{1}{2}\vec{p}^\dagger A\vec{x} - \frac{1}{2}\vec{x}^\dagger A\vec{p} + \frac{1}{2}\vec{x}^\dagger A\vec{x} \\
&= \frac{1}{2}\vec{p}^\dagger A\vec{p} + c + \vec{x}^\dagger \vec{b} - \vec{b}^\dagger \vec{x} - \vec{b}^\dagger \vec{p} \\
&= \frac{1}{2}\vec{p}^\dagger A\vec{p} - \vec{b}^\dagger \vec{p} + c \\
&= f(\vec{p}).
\end{aligned}$$

In the new form of $f(\vec{p})$, one can directly see that if A is positive definite, $\vec{x}$ must minimize the function:

$$f(\vec{p}) = f(\vec{x}) + \frac{1}{2} \underbrace{(\vec{p} - \vec{x})^\dagger A(\vec{p} - \vec{x})}_{> 0 \text{ if } A \text{ pos. def.}}.$$

Therefore $\vec{x}$ is the global unique minimum.

□

TODO: figure of a pos/neg definite quadratic form.

Before deriving the conjugate gradient method, we look at a related method called the ***method of steepest descent***. We are interested in a method that iteratively solves equation (7.1) starting at a ***initial guess*** $\vec{x}_0$ until the series is interrupted, because the approximate solution $\vec{x}_i$ might be close to the real solution by a certain tolerance or the solution was found exactly,

$$\vec{x}_0 \longrightarrow \vec{x}_1 \longrightarrow \cdots \longrightarrow \vec{x}_i \longrightarrow \vec{x}_{i+1} \longrightarrow \cdots$$

For each step, we can define the ***error*** and ***residual*** of the current step i.

Definition 7.2 (Error and Residual)**.** *Define the* ***error*** $\vec{e}_i$ *and the* ***residual*** $\vec{r}_i$ *as*

$$\vec{e}_i = \vec{x}_i - \vec{x}, \tag{7.3a}$$
$$\vec{r}_i = \vec{b} - A\vec{x}_i. \tag{7.3b}$$

The residual is the vector of discrepancies and the same as $\vec{r}_i = -f'(\vec{x}_i) = -A\vec{e}_i$, the negative derivative of the quadratic form. The derivative point ni direction of the maximum increase, thus the residual points in direction of the steepest descent seen from the position of point $\vec{x}_i$.

Definition 7.3 (Method of Steepest Descent)**.** *The iteration step equation of the* ***method of steepest descent*** *in defined as*

$$\vec{x}_{i+1} = \vec{x}_i + \alpha_i \vec{r}_i, \tag{7.4}$$

where the $\alpha_i \in \mathbb{C}$ *are the amounts to go in direction* $\vec{r}_i$. *The* α_i *are determined by minimizing the parabola with respect to* α_i, $\frac{d}{d\alpha_i} f(\vec{x}_{i+1}) \stackrel{!}{=} 0$.

TODO: figure of steepest descent zigzac.

Remark (Convergence). As seen in figure [TODO], the method of steepest descent converges very slowly to the actual solution, when starting at a unfavorable starting point $\vec{x}_0$. The speed of convergence also heavily depends on the condition number of matrix A. We see that the iteration goes in the same direction multiple times. How about, when we only go *once* in each direction i, but by the perfect amount α_i? Then we would be done after at most n steps.

This gives motivation for a enhanced method. Let's define a new ***step equation*** as

$$\vec{x}_{i+1} = \vec{x}_i + \alpha_i \vec{p}_i, \tag{7.5}$$

with ***directions*** $\vec{p}_i$ and ***amounts*** α_i that have to be determined. But this time, we will impose the condition to go in every direction only once at most. This will lead us to the ***method of conjugate gradient***.

Using the step equation (7.5), we can update the error and residuals,

$$\begin{aligned}
\vec{e}_{i+1} &= \vec{x}_{i+1} - \vec{x} && (7.6a)\\
&= \vec{e}_i + \alpha_i \vec{p}_i && (7.6b)\\
&= \vec{e}_0 + \sum_{j=0}^{i} \alpha_j \vec{p}_j, && (7.6c)
\end{aligned}$$

$$\begin{aligned}
\vec{r}_{i+1} &= \vec{b} - A\vec{x}_{i+1} && (7.7a)\\
&= \vec{r}_i - \alpha_i A\vec{p}_i && (7.7b)\\
&= -A\vec{e}_{i+1}. && (7.7c)
\end{aligned}$$

The $\{\vec{p}_i\}$ need to form a basis of $\mathbb{C}^n$, because the method should succeed with any arbitrary initial guess $\vec{x}_0$. Since we move in the vector space $\mathbb{C}^n$ from an arbitrary point $\vec{x}_0$ to the solution $\vec{x}$, the n direction vectors need cover all possible directions in the space, therfore need to be linear independent.

To be done after at most n steps, we need that the n-th error is zero, $\vec{e}_n = 0$. Since the directions form a basis, we can write $\vec{e}_0$ as a linear combination of the $\{\vec{p}_i\}$,

$$\vec{e}_0 = \sum_{j=0}^{n-1} \delta_j \vec{p}_j.$$

Using this we can rewrite $\vec{e}_n$,

$$\begin{aligned}
\vec{e}_n &= \vec{e}_o + \sum_{j=0}^{n-1} \alpha_j \vec{p}_j\\
&= \sum_{j=0}^{n-1} \delta_j \vec{p}_j + \sum_{j=0}^{n-1} \alpha_j \vec{p}_j\\
&= \sum_{j=0}^{n-1} (\delta_j + \alpha_j) \vec{p}_j.
\end{aligned}$$

In order for this to be zero, all coefficients need to be zero, thus $\delta_j = -\alpha_j$. Then the i-th error can be written in a different way

$$\begin{aligned}
\vec{e}_i &= \vec{e}_0 + \sum_{j=0}^{i-1} \alpha_j \vec{p}_j\\
&= \sum_{j=0}^{n-1} \delta_j \vec{p}_j - \sum_{j=0}^{i-1} \delta_j \vec{p}_j\\
&= \sum_{j=i}^{n-1} \delta_j \vec{p}_j. && (7.8)
\end{aligned}$$

In the last row, we can see that after every step in the iteration, we shave off the contribution of one direction $\vec{p}_i$ to the initial error $\vec{e}_0$ (or phrased differently: $\vec{e}_{i+1}$ has no contribution from direction $\vec{p}_i$). But we still need to find these directions. We could for example impose that the $(i+1)$-th error should be orthogonal to the i-th direction, because we never want to go in that direction again,

$$\begin{aligned} 0 &\stackrel{!}{=} \vec{p}_i^{\dagger} \vec{e}_{i+1} \\ &= \vec{p}_i^{\dagger} (\vec{e}_i + \alpha_i \vec{p}_i). \end{aligned}$$

This gives us a expression for the amount α_i,

$$\alpha_i = -\frac{\vec{p}_i^{\dagger} \vec{e}_i}{\vec{p}_i^{\dagger} \vec{p}_i}.$$

The problem with this expression is that we don't know the value of $\vec{e}_i$ - if we would, we could just subtract it from the current $\vec{x}_i$ and obtain $\vec{x}$ exactly. So, we do not know $\vec{e}_i$, but what we actually know is something similar, namely $-A\vec{e}_i$, with is the residual. So if we manage to sandwich an A in the expression above, we are save. It turns out that imposing A-orthogonality instead of regular orthogonality between $\vec{e}_{i+1}$ and $\vec{p}_i$ achieves what we're up to by the exact same steps[2],

$$\begin{aligned} 0 &\stackrel{!}{=} \vec{p}_i^{\dagger} A \vec{e}_{i+1} \\ &= \vec{p}_i^{\dagger} A (\vec{e}_i + \alpha_i \vec{p}_i) \end{aligned}$$

Solving for α_i gives the (almost) final expression for the amounts,

$$\implies \alpha_i = -\frac{\vec{p}_i^{\dagger} A \vec{e}_i}{\vec{p}_i^{\dagger} A \vec{p}_i} = \frac{\vec{p}_i^{\dagger} \vec{r}_i}{\vec{p}_i^{\dagger} A \vec{p}_i}. \tag{7.9}$$

Notice that the denominator is never zero, because A is positive definite. Let us continue with the expression for A-orthoginality, but insert the derived expression (7.8) for $\vec{e}_{i+1}$ this time,

$$\begin{aligned} 0 &\stackrel{!}{=} \vec{p}_i^{\dagger} A \vec{e}_{i+1} \\ &= \vec{p}_i^{\dagger} A \left[\sum_{j=i+1}^{n-1} \delta_j \vec{p}_j \right] \\ &= \sum_{j=i+1}^{n-1} \underbrace{\delta_j}_{\neq 0} \vec{p}_i^{\dagger} A \vec{p}_j. \end{aligned}$$

This implies that for $j > i$ and $i \in \{0, \dots, n-1\}$, we have

$$\vec{p}_i^{\dagger} A \vec{p}_j = 0.$$

But since A is Hermitian, we can Hermitian conjugate the whole expression above and obtain

$$0 = \left(\vec{p}_i^{\dagger} A \vec{p}_j \right)^{\dagger} = \vec{p}_j^{\dagger} A \vec{p}_i.$$

So the expression holds for $i > j$ as well, which implies that the $\{\vec{p}_i\}$ are A-***orthogonal***,

[2]This is equivalent to imposing $0 \stackrel{!}{=} \vec{r}_{i+1}^{\dagger} \vec{p}_i$ which is done in most literature, but in the opinion of the author this is less intuitive.

$$\vec{p}_i^\dagger A\vec{p}_j = 0 \quad \forall i \neq j.$$

So the problem has reduced to finding a set of A-orthogonal vectors in an iterative way. Luckily there is a well know method to find orthogonal vectors from a set of linear independent vectors: ***Gram-Schmidt orthogonalisation***. The procedure can be altered to find A-orthogonal vectors instead.

Definition 7.4 (Gram-Schmidt Orthogonalisation)**.** *Let $\{\vec{u}_0, \dots, \vec{u}_{n-1}\} \subset \mathbb{C}^n$ be a set of n linear independent vectors. The iterative Gram-Schmidt procedure is*

$$\begin{aligned}\vec{p}_0 &= \vec{u}_0 \\ \vec{p}_i &= \vec{u}_i + \sum_{k=0}^{i-1} \beta_{ik}\vec{p}_k,\end{aligned} \tag{7.10}$$

where the $\beta_{ik} \in \mathbb{C}$ are (to be determined) coefficients. In the regluar procedure, the β_{ik} are just normalized projections of $\vec{u}_i$ to $\vec{p}_k$ that are subtracted from $\vec{u}_i$, leading to a vector $\vec{p}_i$ that is orthogonal to all previously calculated $\vec{p}_k$.

In our problem, we need a set of vectors that are A-orthogonal. By imposing this condition we find a different expression for the β_{ik},

$$\begin{aligned}0 &\overset{!}{=} \vec{p}_i^\dagger A\vec{p}_j \\ &= \vec{u}_i^\dagger A\vec{p}_j + \sum_{k=0}^{i-1} \beta_{ik}\vec{p}_k^\dagger A\vec{p}_j \\ &= \vec{u}_i^\dagger A\vec{p}_j + \beta_{ij}\vec{p}_j^\dagger A\vec{p}_j,\end{aligned}$$

where in the last step, we assumed $i > j$ (else we would not find a expression for β_{ij}) and therefore only the j-th term in the sum remains, because of the A-orthonormality of the directions. Solving this for β_{ij} gives

$$\beta ij = -\frac{\vec{u}_i^\dagger A\vec{p}_j}{\vec{p}_j^\dagger A\vec{p}_j}. \tag{7.11}$$

In principle we are done here, we only need a set of linearly independent vectors $\{\vec{u}_i\}$. Since the conjugate gradient method is iterative and often dealing with huge problem sizes n, we need to store all previous directions $\vec{p}_k$ in order to calculate the current direction (see equation (7.10)). This becomes a problem in limited memory situations. We want that the current step only depends on the previous one. By imposing this condition, we need the sum in equation (7.10) to collapse; the β_{ik} should only be non-zero for $k = i - 1$. If we manage to satisfy this, the orthogonalisation procedure would simplify to

$$\begin{aligned}\beta_i &\coloneqq \beta_{i,i-1}, \\ \vec{p}_i &= \vec{u}_i + \beta_i\vec{p}_{i-1},\end{aligned}$$

where in the second equation, the current $\vec{p}_i$ only depends on the previous $\vec{p}_{i-1}$. For this to hold, all other β_{ij} need to be zero. For such a β_{ij} the numerator needs to be zero. Let therefore $j < i - 1$

$$\vec{u}_i^\dagger A\vec{p}_j \overset{!}{=} 0.$$

To find a different expression for the left hand side, consider

$$
\begin{aligned}
\vec{u}_i^\dagger \vec{r}_{j+1} &= \vec{u}_i^\dagger \left(\vec{r}_j + \alpha_j A\vec{p}_j\right) \\
&= \vec{u}_i^\dagger \vec{r}_j + \alpha_j \vec{u}_i^\dagger A\vec{p}_j, \\
\implies \vec{u}_i^\dagger A\vec{p}_j &= \frac{1}{\alpha_j}\left[\vec{u}_i^\dagger \vec{r}_{j+1} - \vec{u}_i^\dagger \vec{r}_j\right],
\end{aligned} \tag{7.12}
$$

where we inserted the recursive relation of the resuduals (7.7b) and the yellow part is the expression we want to be zero for $j < i-1$. We therefore find a condition for the linear independent set $\{\vec{u}_i\}$, namely that the scalar product of $\vec{u}_i$ with $\vec{r}_{j+1}$ and $\vec{r}_j$ must be the same. But we can apply the same equation over and over again and obtain

$$
\vec{u}_i^\dagger \vec{r}_{j+1} = \vec{u}_i^\dagger \vec{r}_j = \dots = \vec{u}_i^\dagger \vec{r}_0, \qquad\qquad j < i-1
$$

We have to find $\{\vec{u}_i\}$ that satisfy the above equation. It is sufficient to find a set of $\{\vec{u}_i\}$ that are orthogonal to all the residuals and the equation would be obeyed.

Lemma 7.2. *The residuals are orthogonal, thus for all $i \neq j$, it holds*

$$
\vec{r}_i^\dagger \vec{r}_j = 0.
$$

Proof. The proof consists of 2 steps.

1) Let $i < j$,

$$
\begin{aligned}
\vec{p}_i^\dagger \vec{r}_j &= -\vec{p}_i^\dagger A\vec{e}_j \\
&= -\sum_{k=j}^{n-1} \delta_j \vec{p}_i A\vec{p}_k \\
&= 0,
\end{aligned}
$$

where the yellow expression is zero, because $i < j \leq k$.

2) Let $i < j$. By step 1), we have

$$
\begin{aligned}
0 &= \vec{p}_i^\dagger \vec{r}_j \\
&= \vec{r}_i^\dagger \vec{r}_j + \sum_{k=0}^{i-1} \beta_{ik} \vec{p}_k^\dagger \vec{r}_j \\
&= \vec{r}_i^\dagger \vec{r}_j.
\end{aligned}
$$

The yellow expression is again zero by step 1). Using the symmetry of the scalar product, the above equation also holds for i and j interchanged ($i > j$), therfore holds for all $i \neq j$.

□

From now on we set $\vec{u}_i = \vec{r}_i$. What remains to find is the final expression for the β_i.

$$
\begin{aligned}
\beta_i := \beta_{i,i-1} &= -\frac{\vec{u}_i^\dagger A\vec{p}_{i-1}}{\vec{p}_{i-1}^\dagger A\vec{p}_{i-1}} \\
&= -\frac{1}{\vec{p}_{i-1}^\dagger A\vec{p}_{i-1}} \frac{1}{\alpha_{i-1}} \left[\vec{r}_i^\dagger \vec{r}_i - \vec{r}_i^\dagger \vec{r}_{i-1}\right]
\end{aligned}
$$

$$
\begin{aligned}
&= -\frac{\vec{r}_i^\dagger \vec{r}_i}{\alpha_{i-1}\vec{p}_{i-1}^\dagger A\vec{p}_{i-1}} \\
&= -\frac{\vec{r}_i^\dagger \vec{r}_i}{\vec{p}_{i-1}^\dagger \vec{r}_{i-1}},
\end{aligned}
$$

where in the first row we used the definition (7.11), in the second row we have used equation (7.12) and the yellow expression is zero by the orthogonality of the residuals lemma 7.2. In the last line we used the expression for the α_j equation (7.9)

To obtain the final form of the α_i and the β_i, we can use a leftover of the proof of lemma 7.2, namely

$$
\begin{aligned}
\vec{p}_i^\dagger \vec{r}_i &= \vec{r}_i^\dagger \vec{r}_i + \beta_i \underbrace{\vec{p}_{i-1}^\dagger \vec{r}_i}_{= 0 \text{ by lemma 7.2 step 1)}} \\
&= \vec{r}_i^\dagger \vec{r}_i.
\end{aligned}
$$

Using this we find the final form of the α_i and the β_i as well as the ***method of conjugate gradient***.

Definition 7.5 (Method of conjugate gradient)**.** *The iteration step equation of the* ***method of conjugate gradient*** *in defined as*

$$
\vec{x}_{i+1} = \vec{x}_i + \alpha_i \vec{p}_i,
$$

with

$$
\begin{aligned}
\vec{r}_{i+1} &= \vec{r}_i - \alpha_i A\vec{p}_i, \\
\vec{p}_{i+1} &= \vec{r}_{i+1} + \beta_{i+1}\vec{p}_i,
\end{aligned}
\qquad
\begin{aligned}
\alpha_i &= \frac{\vec{r}_i^\dagger \vec{r}_i}{\vec{p}_i^\dagger A\vec{p}_i}, && (7.13) \\
\beta_{i+1} &= -\frac{\vec{r}_{i+1}^\dagger \vec{r}_{i+1}}{\vec{r}_i^\dagger \vec{r}_i}, && (7.14)
\end{aligned}
$$

and initial starting vectors

$$
\begin{aligned}
\vec{x}_0 &= \textit{arbitrary starting point}, \\
\vec{p}_0 &= \vec{r}_0 = \vec{b} - A\vec{x}_0.
\end{aligned}
$$

There are some remarks to note about the method of conjugate gradient.

Remark. The β_{i+1} of the current iteration depends on the norm of the current residual as well as the last one. This means that we can store the result of the last iteration and reuse it in the current, the norm may not be calculated twice.

Remark. In the souce code of openQxD (see [3]) the matrix A is the Dirac matrix applied twice $A = D^\dagger D$. This means that the denominator of α_i is a regular inner product as well; $\vec{p}_i^\dagger A\vec{p}_i = \vec{p}_i^\dagger D^\dagger D\vec{p}_i = (D\vec{p}_i)^\dagger (D\vec{p}_i) = \|D\vec{p}_i\|^2$

Remark. Therefore in each iteration, we have:

- 2 times the norm of a vector,
- 2 matrix-vector multiplications,
- 3 times axpy.[3]

Remark (Floating point errors)*.* Since the method contains recursive steps, floating point roundoff accumulation is an issue. This causes the residuals to loose their A-orthogonality. It can be resolved by calculating the residual from time to time using its (computationally more expensive) definition $\vec{r}_i = \vec{b} - A\vec{x}_i$, which involves one matrix vector multiplication. One can for example do this every m-th step. The same problem applies to the directions $\vec{p}_i$ that loose their A-orthogonality.

Remark (Problem size). The method of conjugate gradient is suitable for problems of very huge size n. The algorithm is done after n steps, but there might be problems such that even n steps are out of reach for an exact solution.

Remark (Complexity). The time complexity of the conjugate gradient method is $O(m\sqrt{\kappa})$, where m is the number of non-zero entries in A and κ is its ***condition number***. The space complexity is $O(m)$.

Remark (Starting). The ***starting vector*** $\vec{x}_0$ can be chosen at wish. If there is already a rough estimate of the solution one can take that vector. But usually just $\vec{x}_0 = 0$ is chosen. Since the minimum is global, there is no issue in chosing a starting point. The method will always converge towards the real solution.

Remark (Stopping). If the problem size does not allow to run n steps, one can stop when the norm of the residual falls below a certain ***threshold*** value. Usually this threshold is a fraction of the initial residual $\|\vec{r}_i\| < \epsilon\|\vec{r}_0\|$ [14].

Remark (Initialization). The very first step of the method is equivalent to a step in the method of steepest descent, see equation (7.4).

Remark (Speed of convergence). TODO: cg is quicker if there are duplicated eigenvalues. number of iterations for exact solution is at most the number of distinct eigenvalues.

Remark (Preconditioning). The linear system of equations can be transformed using a matrix M to

$$M^{-1}A\vec{x} = M^{-1}\vec{b}.$$

It is assumed M is such that is is easy to insert and it approximates A in some way, resulting in $M^{-1}A$ to be better conditioned than was A. An examples of a particular preconditioner M would be a diagonal matrix, with diagonal entries of D. It is indeed easy to invert and it approximates A quite well if A has non-zero diagonal entries and most off-diangonal entries are zero.

Remark (Conjugate Gradient on the normal equations (CGNE)). The algorithm can be used even if A is not symmetric nor Hermitian nor positive definite. The linear system of equations to be solved is then

$$A^\dagger A\vec{x} = A^\dagger\vec{b}.$$

If A is square and invertible, solving the above equation is equivalent to solving $A\vec{x} = \vec{b}$. Conjugate gradient can be applied, because $A^\dagger A$ is Hermitian and positive ($\vec{x}^\dagger A^\dagger A\vec{x} = \|A\vec{x}\| \geq 0$). Notice that $A^\dagger A$ is less sparse than A, and often $A^\dagger A$ is bady conditioned.

8 Real number formats

8.1 IEEE Standard for Floating-Point Arithmetic

Floating point numbers are omnipresent in the scientific applications. In the conjugate gradient kernel of [3], there are large scalar products over vectors of very high dimensionality over multiple ranks. The components of these vectors are single precision floating point numbers (I call them binary32 from here on). The precision was degraded from binary64 to binary32 already and a speedup of a factor of 2 was achieved. This motivates to explore even smaller floating point formats with encoding lengths of 16 bits. Since scalar products as well as matrix-vector products are memory-bound operations, going to a smaller bit-length will increase the throughput of the calculation. Therefore, a 16 bits floating point format with a smaller exponent could lead to a double of performance if the new operation is still memory-bound.

[3] This stands for $a\vec{x} + \vec{y}$, scalar times vector plus vector, "a x plus y" (to resemble the BLAS level 1 routine call of the same name).

Definition 8.1 (IEEE 754 Floating point format)**.** *The* ***IEEE 754 floating point format*** *[11] is defined using the* ***number of exponent bits*** *e and the* ***number of mantissa bits*** *m respectivetly. A binary floating point number is illustrated in Figure 2.*

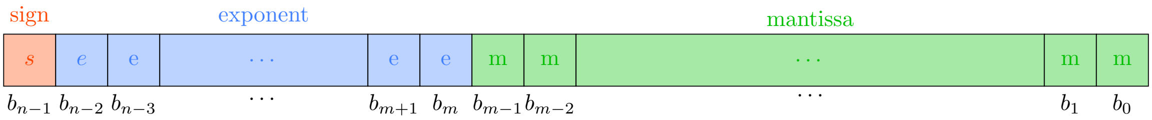

Figure 2: Binary representation of a IEEE 754 n-bit precision floating-point number. The orange bit represents the sign bit, the blue bits represent the fixed-length e exponent bits and the green bits represent the fixed-length m mantissa bits. Notice that $n = 1 + e + m$.

The resulting floating point number is then calculated as

$$f = (-1)^s \cdot M \cdot 2^E,$$

where $E = E' - B$ denotes the biased exponent, B is the exponent bias, M the mantissa and s the sign bit. The (unbiased) exponent E' is calculated as follows

$$E' = \sum_{i=0}^{(e-1)} b_{m+i} 2^i, \tag{8.1}$$

where B is the exponent bias.

Definition 8.2 (Exponent bias)**.**

$$B = 2^{(e-1)} - 1,$$

The calculation of the mantissa is a bit more involved, since it depends on the number being normal or subnormal.

Definition 8.3 (Subnormal numbers)**.** *The IEEE 754 standard introduces so called* ***subnormal numbers****. If all the exponent bits are* 0*, meaning the unbiased exponent $E' = 0$, and the mantissa bits are not all* 0*, then the number is called subnormal. The exponent being zero causes the implcit bit to flip to* 0*, instead of* 1*.*

Remark. Subnormal numbers have a variable-length mantissa and exponent, because some of the mantissa bits are used as additional exponent bits, making the numbers less precise the lower they get (see the smooth cutoff in Figure 4).

Therefore the mantissa of a regular (non-subnormal) number is (when the exponent $0 < E < B$, this implies that the implicit bit is 1)

$$M = \underset{\text{implicit bit}}{1} + \sum_{i=1}^{m} b_{m-i} 2^{-i},$$

whereas the mantissa of a subnormal number (when the exponent $E = 0$) is

$$M = \underset{\text{implicit bit}}{0} + \sum_{i=1}^{m} b_{m-i} 2^{-i},$$

The 1 or 0 in the front of the summand is the leading ***implicit bit***, sometimes also called the $(m+1)$***-th mantissa bit*** that tells us whether the number is subnormal or not.

Floating point formats				
name	s	e	m	comment
binary64	1	11	52	double precision, IEEE 754 [12]
binary32	1	8	23	single precision, IEEE 754 [12]
binary16	1	5	10	half precision, IEEE 754 [12]
bfloat16	1	8	7	Googles Brain Float [15]
tensorfloat32	1	8	10	NVIDIAs TensorFloat-32 [8] [4]
binary24	1	7	16	AMDs fp24 [2]
binary128	1	15	112	IEEE 754 [12]
binary256	1	19	236	IEEE 754 [12]

Table 1: Commonly used floating point formats, where s is the number of sign bits, e the number of exponent bits and m the number of mantissa bits.

Remark. The mantissa range of a regular floating point number is $M \in [1, 2)$, whereas the matissa range of a subnormal floating point number is $M \in (0, 1)$. The number zero is not considered subnormal.

Usual floating point formats are summarised in Table 1.

The format of interest is the binary16 half precision IEEE 754 floating point format. The highest representable number is when the exponent is highest. This is not the case when all e exponent bits are 1, because then - according to the specification [11] - the number is either $\pm\infty$ or not a number (NaN), depending on the mantissa. The maximal unbiased exponent is therefore the next smaller number,

$$E'_{max} = \underbrace{1...1}_{e-1 \text{ times}} 0.$$

Using equation (8.1), we find

$$\begin{aligned} E'_{max} &= \sum_{i=1}^{(e-1)} 2^i \\ &= 2^e - 2. \end{aligned}$$

The mantissa on the other hand is maximal when all mantissa bits are 1 (including the implicit bit),

$$\begin{aligned} M_{max} &= 1 + \sum_{i=1}^{m} 2^{-i} \\ &= 2 - 2^{-m}. \end{aligned}$$

Using these two formulas we can define the

Definition 8.4 (***highest representable number***)**.** *The* ***highest representable number*** *in any floating point format is*

$$\begin{aligned} f_{max} &= (-1)^0 \cdot M_{max} \cdot 2^{(E'_{max} - B)} \\ &= (2 - 2^{-m}) \cdot 2^{(2^e - 2^{e-1} - 1)} \\ &= (2 - 2^{-m}) \cdot 2^{(2^{e-1} - 1)}. \end{aligned}$$

[4] Allocates 32 bits, but only 19 bits are actually used.

Floating point format limits				
name	f_{max}	f_{min}	f_{smin}	sign. digits [5]
binary64	1.8×10^{308}	2.2×10^{-308}	4.9×10^{-324}	≤ 15.9
binary32	3.4×10^{38}	1.2×10^{-38}	1.4×10^{-45}	≤ 7.2
binary16	6.6×10^{4}	6.1×10^{-5}	6.0×10^{-8}	≤ 3.3
bfloat16	3.4×10^{38}	1.2×10^{-38}	9.2×10^{-41}	≤ 2.4
tensorfloat32	3.4×10^{38}	1.2×10^{-38}	1.1×10^{-41}	≤ 7.2
binary24	1.8×10^{19}	2.2×10^{-19}	3.3×10^{-24}	≤ 5.1
binary128	1.2×10^{4932}	3.4×10^{-4932}	6.5×10^{-4966}	≤ 34
binary256	$1.6 \times 10^{78,913}$	$1 \times 10^{-78,912}$	$1 \times 10^{-78,983}$	≤ 71.3

Table 2: Summary of highest representable numbers, minimal subnormal and non-subnormal representable numbers above 0 in any IEEE 754 floating point format together with their approximated precision.

The minimal number above 0 can be found similarly, using minimal unbiased exponent (when all exponent bits are 0, except the last one, therefore $E'_{min} = 1$) and the minimal mantissa ($M_{min} = 1$).

Definition 8.5 (***minimal (non-subnormal) representable number above*** 0)**.** *The* ***minimal (non-subnormal) representable number above*** 0 *in any floating point format is*

$$\begin{aligned} f_{min} &= (-1)^0 \cdot M_{min} \cdot 2^{(E'_{min} - B)} \\ &= 2^{(2-2^{e-1})}. \end{aligned}$$

The minimal subnormal number can the found, when the unbiased exponent consists of only zeros ($E'_{smin} = 0$) and for the mantissa, only the rightmost bit is one ($M_{smin} = 2^{1-m}$).

Definition 8.6 (***minimal subnormal representable number above*** 0)**.** *The* ***minimal subnormal representable number above*** 0 *in any floating point format is*

$$\begin{aligned} f_{min} &= (-1)^0 \cdot M_{smin} \cdot 2^{(E'_{smin} - B)} \\ &= 2^{1-m} \cdot 2^{(1-2^{e-1})} \\ &= 2^{(2-m-2^{e-1})}. \end{aligned}$$

See Table 2 for these limiting numbers in the different floating point formats.

8.2 Posits

The posit datatype is designed to be a replacement for the IEEE floating point format, fixing its various quirks. Some of the more entertaining are:

- The appearance of NaNs. They are considered unnatural, because a specific bit pattern describing a number that is not a number is a contradiction.
- The NaNs and the fact that floats have two different representations for the number zero (0 and -0) lead to very complicated and slow comparison units.
- Floats may under- or overflow, because the standard employs the round to nearest even rounding rule ($\pm\infty$ and 0 are considered even).
- Floats are non-associative and non-distributive[6]leading to rounding errors that have to be taken into account, specially in scientific computing.

[5]Number of significant digits in decimal; $-\log_{10}(\texttt{MACHINE\_EPSILON}) = \log_{10}(2^{m+1})$.

- The standard gives no guarantee of bit-identical results across systems.

The goal is to utilise the number of bits more efficiently and remove these inconsistencies. The key idea is to place half of all numbers between 0 and 1 and the other half are the reciprocals (the reciprocal of 0 being $\pm\infty$). The number can then be drawn on a projective real number circle [7]. The structure of a binary posit number is illustrated in Figure 3.

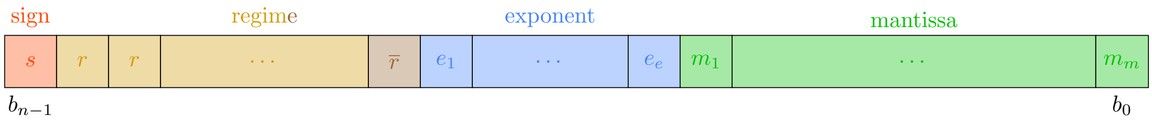

Figure 3: Binary representation of a n-bit posit number. As with regular floats the orange bit represents the sign bit, the yellow bit(s) represent the variable length regime bit(s) terminated by the brown bit that is the opposite regime bit, the blue bit(s) represent the variable-length exponent bit(s) and the green bit(s) represent the variable-length mantissa bit(s).

The actual value of the number is calculated as follows. The yellow and brown bits determine the regime of the number. They either start with a row of all 0 or all 1 terminated by the opposite bit indicating the end of the row. The number of bits in the row are counted as m and if they are all 0 they get a minus sign, the regime being $k = -m$. If they are all 1 the regime is calculated as $k = m - 1$. After the regime is decocded, the remaining bits contain the exponent with at most es bits depending on how much bits remain. If no bits remain the exponent is 0. The exponent and the mantissa are both of variable length. Both can have 0 bits, in this case the number consists of only regime bits. This is the reason why posits have a larger number range than floats. The exponent is enconded as unsigned integer, so there is no bias and no bit pattern denoting special numbers such as subnormals or NaNs. Therefore n-bit posits have more numbers than n-bit floats, because they have no NaNs. After the exponent - if there are still bits remaining - the fraction follows, else the fraction is just 1.0 Since there are no subnormals the implicit bit is always 1. There are two special numbers that do not follow the above encoding scheme; zero which has the bit pattern of all 0 and $\pm\infty$ with a 1 followed by all 0. These two numbers are reciprocals of each other. A general posit number can therefore be written as

$$p = (-1)^s \cdot useed^k \cdot M \cdot 2^E,$$

where s is the sign bit, $useed$ is defined to be $useed = 2^{2^{es}}$, with es the number of predefined exponent bits, M is the mantissa and E the exponent.

The mantissa is calculated as

$$M = 1 + \sum_{i=1}^{m} m_i 2^{m-i},$$

where m is the variable number of mantissa bits and the implicit bit in front of the sum is always 1. The exponent is

$$E = \sum_{i=1}^{e} e_i 2^{e-i},$$

where e is the variable number of exponent bits satisfying $e \leq es$.

Using these two equations, we are now able to calculate the highest representable number and the minimal representable number above 0 in posit format.

Definition 8.7 (***highest representable number***)**.** *The* ***highest representable number*** *in any posit format is*

[6] There was even a system using IEEE 754 that had non-commutative floating point operations[4].

Posit format limits				
name	*es*	p_{max}	p_{min}	sign. digits [7]
posit64	3	2.0×10^{149}	4.9×10^{-150}	≤ 17.7
posit32	2	1.3×10^{36}	7.5×10^{-37}	≤ 8.1
posit16	1	2.7×10^{8}	3.7×10^{-9}	≤ 3.6
posit8	0	64	1.6×10^{-2}	≤ 1.5

Table 3: Summary of highest representable numbers, minimal representable numbers above 0 in any posit format together with their approximated precision.

$$
\begin{aligned}
p_{max} &= (-1)^0 \cdot useed^{n-2} \\
&= 2^{2^{es}(n-2)}.
\end{aligned}
$$

Definition 8.8 (***minimal representable number above*** 0)**.** *The **minimal representable number above** 0 in any posit format is the reciprocal of the highest representable number* p_{max}

$$
\begin{aligned}
p_{min} &= \frac{1}{p_{max}} \\
&= 2^{2^{es}(2-n)}.
\end{aligned}
$$

See Table 3 for these limiting numbers in the different posit formats.

Posits employ a feature called the quire, which is the generalized answer to the fused multiply–add operation that recently found its way into [12] in 2008, where the rounding is deferred to the very end of the operation.

8.3 Floating point numbers in openQxD

To explore how the conjugate gradient kernel in openQxD would perform when using smaller bit lengths, one can look at the exponentials of the numbers in the matrix and vectors, see Figure 5. The plot shows all exponents appearing together with their overall occurence in percent. The number zero was taken from the plot, because it has biased exponent $E = -127$. The occurences for zero are given in the legend.

The highest exponent in all 4 runs was $E = 4$, whereas the lowest exponent deceased when the number of lattice points increased. The range of exponents that is representable in binary16 spans from -24 to $+16$ and is indicated by the solid orange line and the solid pink line. Between -24 and -14 is the regime of subnormal numbers in binary16, with the lowest regular (non-subnormal) exponent indicated by the solid blue line. When using half precision instead of single precision, all numbers with exponents below -24, will be converted to zero, whereas exponents above $+16$ will be casted to $\pm\infty$ depending on the sign of the number. It can be seen, that when calculating the norm of these numbers, only numbers between the dashed blue line and the dashed pink line will participate. If there is a number above the dashed pink line in the unsave region this number will - after squaring - be casted to ∞ and therefore the norm will be ∞ as well[8]. In this case the variable representing the norm $x = \|\vec{v}\|$ should be of higher precision than binary16. The plot shows that the Dirac matrix `Dop()` is confined in a narrow exponent regime and a representation in 16-bit floats would suffice. Notice the sparsity the Dirac matrix.

8.4 The conjugate gradient kernel in openQxD

The conjugate gradient kernel `cgne()` in `modules/linsolv/cgne.c` in [3] implements the algorithm, see Listing 1.

[7] Number of significant digits in decimal; $-\log_{10}$(`MACHINE_EPSILON`). Notice that posits have ***tapered accuracy***; numbers near 1 have much more precision than numbers at the borders of the regime. The precision of floats decreses as well with very large and small numbers, but posit precision decreases faster, see Figure 4.

[8] A method to circumvent this is to scale the vector entries during the calculation and scale the result back, exploiting homogeneity of the norm, $\|\vec{v}\| = \frac{1}{s}\|s\vec{v}\|$ for $s \in \mathbb{R}_{>0}$.

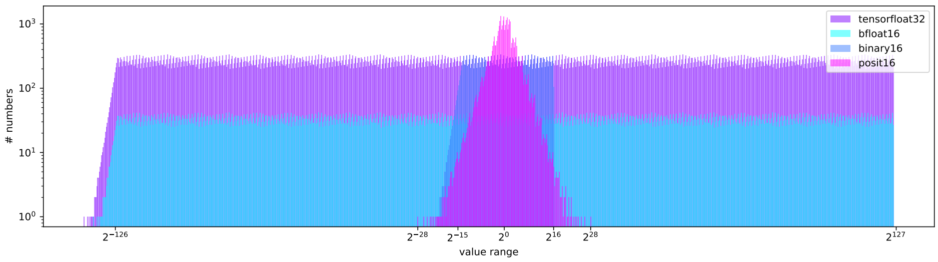

Figure 4: Density or distribution of numbers for tensorfloat32, binary16, posit16 and bfloat16. The number of bins was chosen to be 1024 of logarithmic width. The IEEE conformant floats tensorfloat32, binary16 and bfloat16 exhibit a simmilar shape, namely the distribution of numbers is exponential decreasing for higher and smaller numbers. The high numbers undergo a rough cutoff at the highest representable number. Numbers above that value will be cast to infinity. Compared to this, the small numbers show a smooth cutoff, because of the existence subnormal numbers. The range of posit16 is bigger than the range of binary16, but specially in the very small numbers this difference in range is neglectible. Some features of posits can be observed: First, their distribution is symmetric around 1, because posits have no subnormals. Second, more numbers are closer to 1 than in case of floats; the closer to 1, the better the number resolution. Closest to 1, the number resolution becomes better than binary16 resolution. Third, posits have no fixed-length mantissa nor exponent. That's the reason why the height of the posit shape depends on the number regime, which happens for floats only in the subnormal regime, where the exponent and mantissa are indeed of variable length. For all formats, the amount of numbers decreases exponentially when going away from 1, but posits decrease faster. This suggests that when calculating in the number regime close to 1 posits might be the better choice, but when numbers span the whole number range equally, floats might be superior. But in that case one has to take care about over- and underflows. Notice that the height of the shape is determined by the number of mantissa bits, therefore giving the precision, whereas the width is determined by the number of exponent bits, therefore giving the number range. For example tensorfloat32 and binary16 have a very different number range, but exhibit the same percision for numbers in their intersection, meaning that binary16 is a subset of tensorfloat32. On the other hand comparing tensorfloat32 and bfloat16 they have approximately the same number range, but different precisions in them, meaning that bfloat16 is as well a subset of tensorfloat32, which itself is a subset of binary32. Notice that when plotting binary32 and posit32 in such a plot, they would look very similar to binary16 versus posit16.

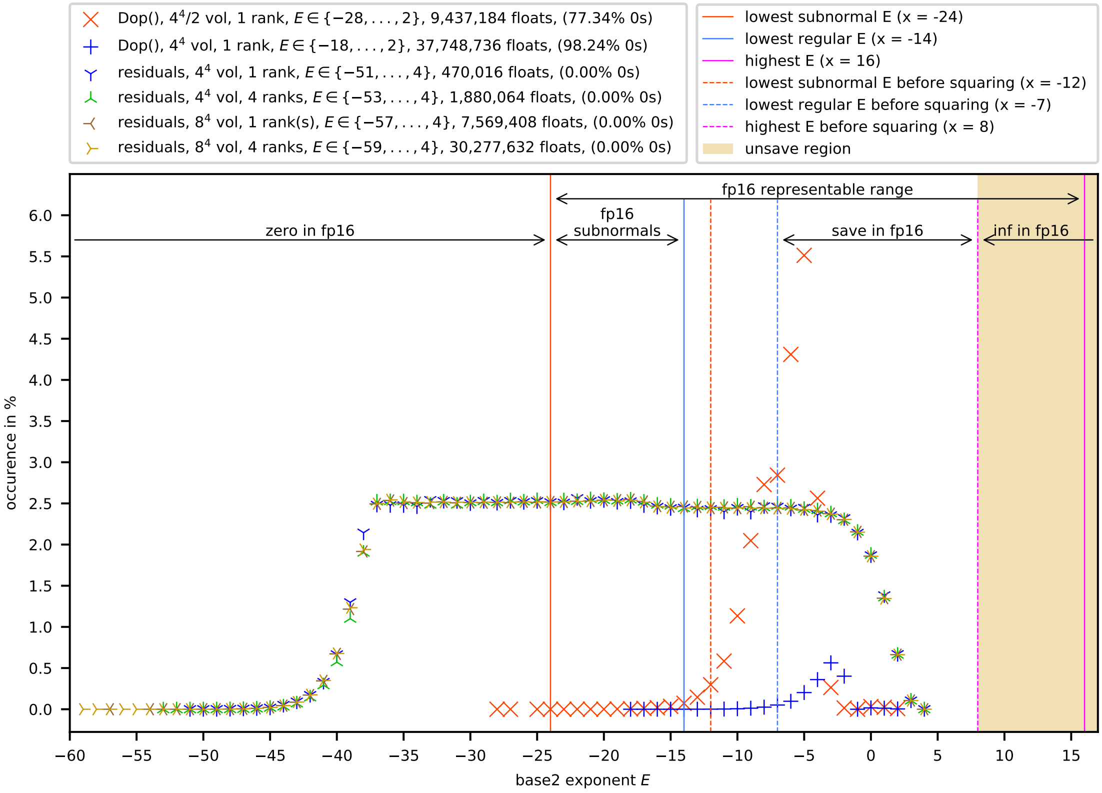

Figure 5: Exponent distribution of binary32 single precision floats in the residual vectors of all steps in a conjugate gradient run in openQxD as well as entries of the Dirac operator. 4 runs were made, with a lattice size of 4^4 and 8^4 on one single rank and 4 ranks respectively. The number is normalised to $(-1)^s \cdot M \cdot 2^E$, where $M \in [1, 2)$.

```
429 double cgne(int vol,int icom,void (*Dop)(spinor *s,spinor *r),
430             void (*Dop_dble)(spinor_dble *s,spinor_dble *r),
431             spinor **ws,spinor_dble **wsd,int nmx,double res,
432             spinor_dble *eta,spinor_dble *psi,int *status)
433 {
```

Listing 1: The conjugate gradient kernel in `modules/linsolv/cgne.c` line 429ff.

```c
490 if ((rn<=tol)||(rn<=(PRECISION_LIMIT*xn))||(ncg>=100)||
491     ((*status)>=nmx))
492   break;
```

Listing 2: break condition in `modules/linsolv/cgne.c` line 490ff, `rn` is the norm of the current residual, `xn` is the norm of the current solution vector, both in binary32.

The function expects the Dirac matrix `Dop()` in binary32, `Dop_dble()` in binary64 format and the source vector `eta` ($\vec{b}$) in binary64 only. In the initialisation the starting vector `psi` ($\vec{x}_0$) is set to zero. The algorithm stops when the desired maximal relative residue `res` ($= \frac{\|\mathtt{eta} - D^\dagger D\mathtt{psi}\|}{\|\mathtt{eta}\|}$) is reached, where `psi` is the calculated approximate solution of the Dirac equation $D^\dagger D\mathtt{psi} = \mathtt{eta}$ in binary64. For this, the tolerance `tol` is calculated using $\mathtt{tol} = \|\mathtt{eta}\| * \mathtt{res}$. The parameter `nmx` is the maximal number of iterations that may be applied and `status` reports the total number of iterations that were required, or a negative value if the algorithm failed. `icom` is a control parameter and `ws` and `wsd` are wordspace allocations. The volume of the lattice should be given in `vol`.

Since the Dirac matrix is given in two precisions, the algorithm in the code bails out of the main conjugate gradient loop, when some particular conditions where met, see Listing 2.

This may happen in 4 cases:

1. if the recusively calculated residual is below the tolerace,
2. if the precision of binary32 is reached[9],
3. after a hardcoded number of 100 steps,
4. if the maximal number of steps is reached.

Point 2 is the most interesting condition, because lets imagine that this condition is met, but the algorithm does not break out of the main loop. Therefore the norm of the current residual compared to the norm of the current solution vector differ in their orders of magnitude by the precision limit of the datatype (binary32 in this case). This means that the solution vector $\vec{x_i}$ contains large numbers compared to the residual vector $\vec{r_i}$. Therefore the changing in residual from iteration to iteration is small compared to numbers in $\vec{x_i}$ as well. Since $\vec{r_i}$ contains small numbers, the amounts α_i are small as well. This causes $\vec{x_{i+1}} = \vec{x_i} + \alpha_i \vec{d_i}$ to not change anymore, because adding very large and very small numbers in floating point arithmetic will return the larger number unchanged if the two numbers differ in magnitude by the precision limit of the datatype. The algorithm stalls in that case and breaking out of the main loop is the emergency brake.

So when one of the above conditions are met, the algorithm performs a ***reset step***. A reset step consists of calculating the residual not in the recursive way, instead calculating it in it's definition $\vec{r_i} = \vec{b} - A\vec{x_i}$ in double precision. This involves 2 invocations of each `Dop_dble()` as well as `Dop()` which is very expensive. The algorithm is resetting in the sense that the solution vector is set back to $\vec{x_i} = 0$, but before resetting, the solution vector in binary32 is added to the real solution vector `psi` in binary64 which was initialied to zero at the start of the algorithm as well. It looks like a restart of the whole calculation, but the direction for the next iteration $\vec{d_i} = \vec{r_i}$ is set to the just calculated, very accurate residual. Therefore the the algorithm now continues in a new direction A-orthogonal to all previous directions and progression is kept. The step is meant to remove the accumulated roundoff errors due to the recursive calcuation of the residuals and directions. The first step following a reset step is a step in the direction of steepest descent just like the very first step of the algorithm. The less precise the datatype, the more reset steps need to be taken, because the precision limit is reached earlier.

8.5 Simulating other datatypes

Some operations such as norms and scalar products are memory-bandwidth-bound, which means the on-chip memory bandwidth determines how much time is spent computing the output. Storing

[9] The constant `PRECISION_LIMIT` is defined to be `100*MACHINE_EPSILON`, where the `MACHINE_EPSILON` is the difference between 1 and the lowest value above 1 depending on the datatype. In case of binary32 the `MACHINE_EPSILON` takes a value of $1.192{,}092{,}9 \times 10^{-7}$.

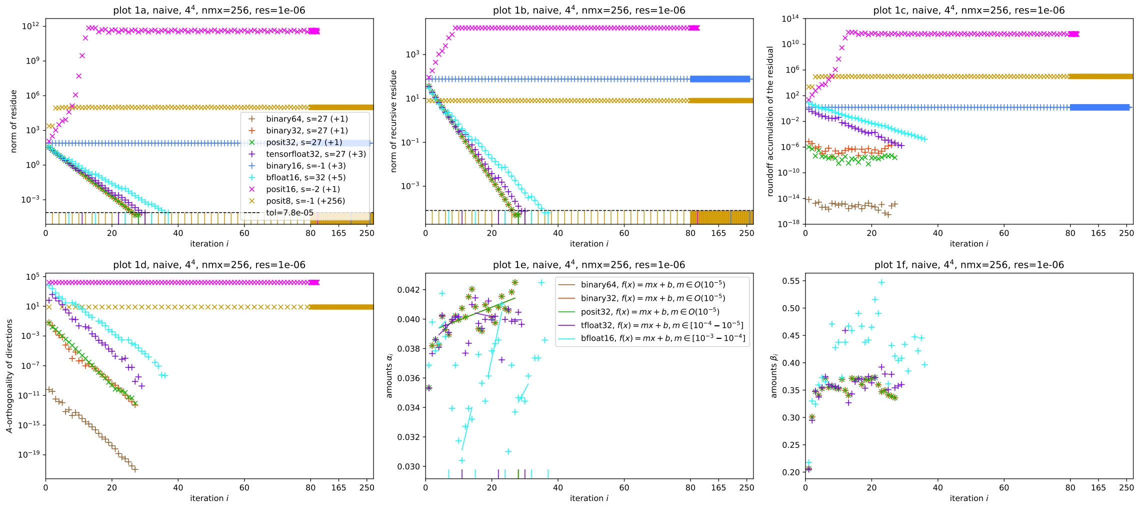

Figure 6: Convergence analysis of a conjugate gradient run, where binary32 was replaced by one of the simulated datatypes. The number `s` describes the number of normal steps needed (the value of `status`), whereas the numbers in the brackets indicate the number of reset steps. All reset steps are indicated by ticks at the dashed black line denoting the tolerance limit. The iterations will always go up to `nmx=256`, but the range 80-256 is compressed since the most interesting behavior happens before step 80 for most of the simulated datatypes. The 6 plots show the naive replacement of the binary32 datatype with the simulated one. This means that every single variable containing a binary32 was replaced with a variable of the simulated datatype. Plot *1a* shows the exact residue (7.7a) calculated in every iteration using the Dirac matrix and the source vector both in binary64, whereas plot *1b* shows the norm of the recursively calculated residue (7.7b) (casted from the simulated datatype to binary64). The relative residue suffers roundoff accumulation because of the recusive calculation; this is the differnce between plots *1a* and *1b*, which is plotted in plot *1c*. Plot *1d* shows the A-orthogonality of the current direction to the last direction, namely the value of $\vec{p}_i^\dagger A \vec{p}_{i+1}$. The last 2 plots, *1e* and *1f*, show the values of the amounts α_i and β_i (see equations (7.13) and (7.14)) in every iteration, but only of the datatypes that converged (`status>0`). The lines in plot *1e* are linearly fitted to the data points ($f(x) = mx + b$). The number range of the slope m is given in the plot legend.

input data in a format with lower bit-length reduces the amount of data to be transferred, thus improving the speed of calculation.

The complete conjugate gradient kernel was simulated in different datatypes, floats as well as posits. In order to produce the plots, the dirac matrix `Dop_dble()` and the source vector `eta` were extracted in binary64 format from the original code running a simulation of a 4^4 lattice, Schrödinger functional (SF) boundary conditions (`type 1`), no C* boundary conditions (`cstar 0`) and 1 rank. The first 2000 trajectories were considered of thermalization. The matrix was extracted in trajectory 2001. A python script mimicking the exact behavior of the `cgne()` kernel from the source code[10], was implemented to cope with arbitrary datatypes. The simulated datatypes were binary64, binary32, tensorfloat32, binary16, bfloat16, posit32, posit16, and posit8. The Dirac matrix had approximately 2% non-zero value. The results are plotted in figures 6, 7, 8 and 9.

8.5.1 Discussion of figures 6 - 9

Figures 6, 7, 8 and 9 contain all relevant data. It is expected in general that the plots show datatypes of the same bit-length in clusters and exhibit a hierarchy in precision and exponent range; more precision and larger exponent range should end up in faster convergence. Thus we expect the following hierarchy (where smaller means convergence in fewer steps)

$$\text{binary64} < \text{posit32} \leq \text{binary32} \leq \text{tensorfloat32} \leq (1) \leq \text{posit16} \leq \text{binary16} \leq (2) < \text{posit8}, \quad (8.2)$$

[10] See line 429ff in `modules/linsolv/cgne.c` in [3].

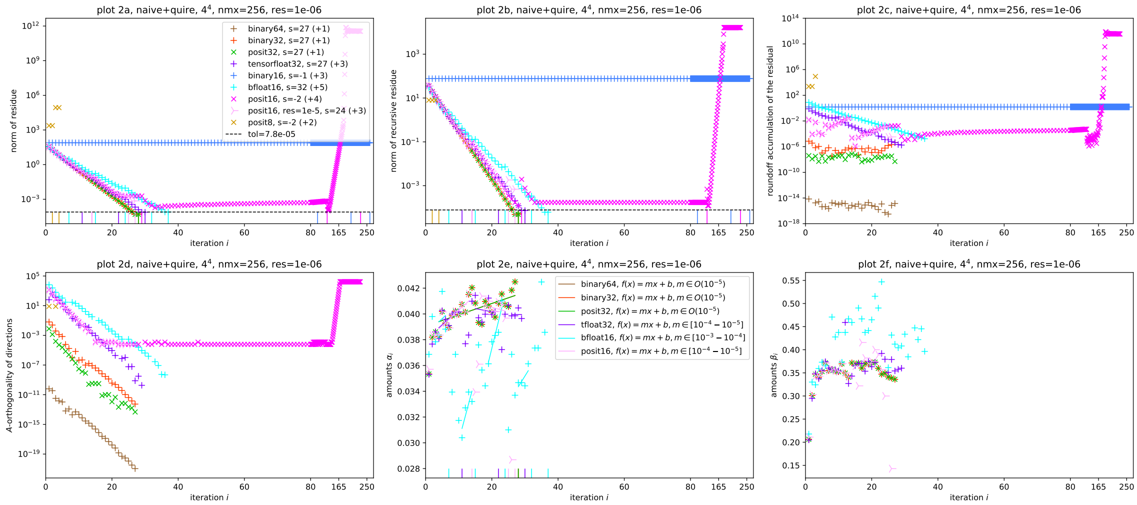

Figure 7: In these plots, the posits were utilizing quires as their collective variables, the remaining setup was the same as for firgure 6, therefore the floating point datatypes show exactly the same values, only posits changed their behavior.

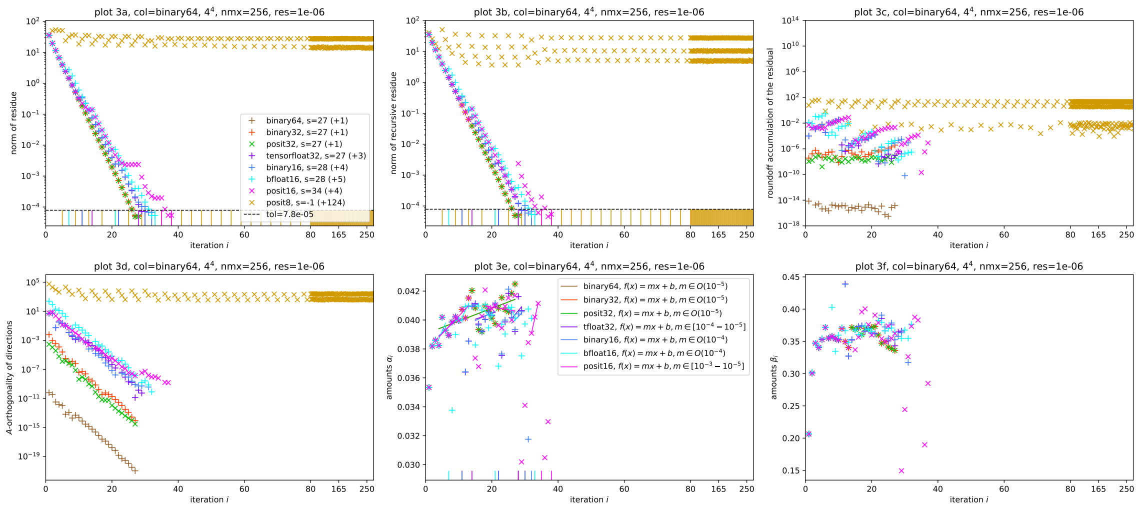

Figure 8: The 6 plots introduce a slightly smarter replacement. All collective variables such as norms where calculated in binary64, such that a datatype with a small number range such as binary16 may not over- or underflow when calculating the norm of a vector full of said datatype. This replacement resembles the quire for posits. Using this replacement, even heavily reduced datatypes like binary16 and posit16 converged and threw a result of equal quality as the one simulated with binary64.

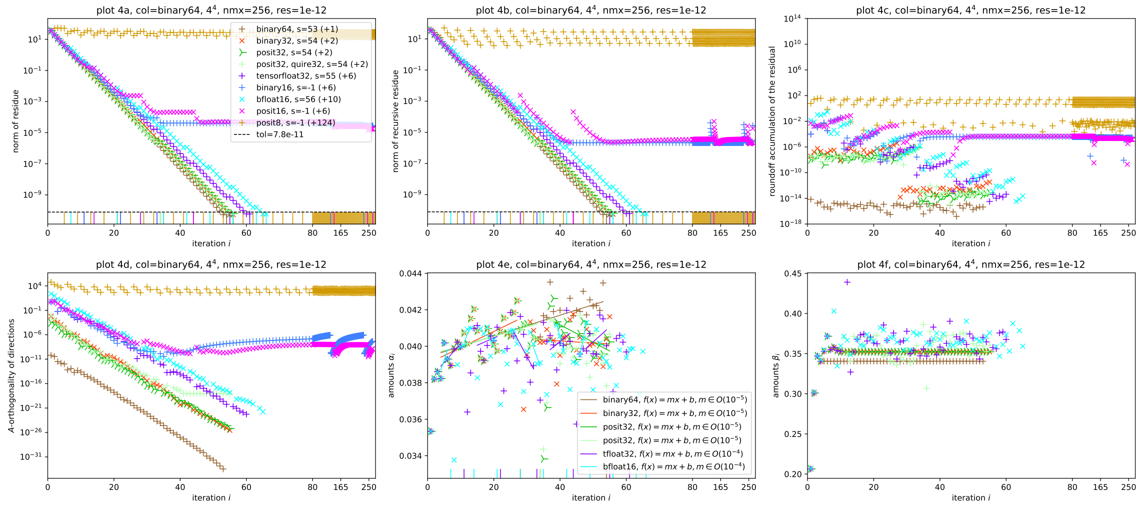

Figure 9: The configuration in this series of plots is equal to Figure 8, besides the value of `res` - the desired relative residue of the calculated solution - is set to 10^{-12} instead of 10^{-6}. Notice that 10^{-12} is outside the representable number range of the datatypes that did not converge; binary16, posit16 and posit8.

where bfloat16 could be either at position (1) or (2), depending on what is more important; precision or number range.

In Figure 6 where the datatype is naively replaced by the simulated datatype, it can be concluded that only datatypes with large enough number ranges converged. binary64, binary32 and posit32 converged each after `status=27` steps with one reset step. The less precise tensorfloat32 took `status=27` (+3) and the even less precise bfloat16 needed `status=32` (+5) steps. Such a hierarchical result was expected since they have the same exponent range and thus approximately the same number range, but differ only in precision (see Table 1). Notice that the less precise the datatype, the more reset steps are needed. This happens because the precision limit of the simulated datatype is reached faster, if the datatype has less precision.

The roundoff accumulation error of posit32 is slightly better than the one of binary32, although defeated by 8 orders of magnitude of binary64 because of its much more precision. It is notable to remark that the roundoff accumulation does not increase substantially from step to step, what would be expected from a recursive calculation. The reason for the small difference between binary32 and posit32 could be that the involved real numbers are closer to representable numbers in posit32 than in binary32. Posits have a larger number density around 1 compared to floats of the same bit-length, and therefore more precision in that regime (see Figure 4 for the example of binary16 versus posit16). Posits also have more numbers, because they have no NaNs. Roundoff accumulation is specially dependent on the precision of the datatype, which makes sense; the lower the precision, the higher the roundoff accumulation. The difference in A-orthogonality is neglectible for posit32 compared to binary32, but again clearly surpassed by binary64.

binary16 did not converge (`status=-1`) after the maximal number of `nmx=256` steps. Its footprint is absent in plot *1d*, because it consisted only of NaNs and infinities, causing $\alpha_i = 0$ and $\beta_i = 1$. This implied that $\vec{r}_i = \vec{r}_{i+1}$ and $\vec{p}_{i+1} = \vec{r}_{i+1}$ and therfore $\vec{x}_{i+1} = \vec{x}_i$ and the algorithm stalled. This explains the residues not changing in plots *1a* and *1b*. The reason for the first infinity was an overflow when calculating the norm of $\vec{b}$ in the very first iteration. This suggests that the limited number range of binary16 might not be enough (at least for a naive replacement), comparing to bfloat16 with the same bit-length, but larger number range that was able to converge, although very slowly.

The behavior of posit8 is very similar to binary16, but without the overflow, because posit do not overflow by definition. Instead the biggest representable number is returned or in case of an underflow the smallest representable number is returned [5]. The algorithm stalled at a value of the norm of the recusive residual of $\|\vec{r}_i\| = 8$. The biggest 8-bit posit number with exponent bits $es = 0$ is $2^6 = 64$, so the norm squared cannot be bigger than 64 and the norm itself cannot be bigger than $8 = \sqrt{64}$ (see plot *1b*). This happend in the first step, whereas the actual residual in binary64 was $\backsim 10^3$. The amounts $|\alpha_i| \ll 1$ in iterative steps are therfore very small causing $\vec{x}_{i+1} \approx \vec{x}_i$. Significant changes in $\vec{x}_i$ will not happen and convergence is unlikely. Also notice that posit8 had 256 reset steps, which means that after every step there was a reset step. The steps where caused by the very high precision limit of posit8. The value of `PRECISION_LIMIT` is 100∗`MACHINE_EPSILON`, which has a value of 3.125 for posit8.

The story of posit16 is very similar, just that the maximal representable value with $es = 1$ is 268,435,456 and the square root of this is 16,384 which is reached after 8 steps (see plot *1b*). The actual residual in the 8-th step was $\backsim 10^7$, the algorithm diverged and then stalled. Iterative steps are therfore mostly too small and convergence is unlikely.

We observe that number range is more important than precision, when naively replacing the datatype, but the higher the precision, the faster the convergence and the less reset steps needed.

In Figure 7 the replacement utilised the possibility to use quires for the posit runs. Therefore, the numbers for the float datatypes are exactly equal to the ones in Figure 6, because floats have no such feature. They are not discussed again.

Comparing plots *1c* and *2c* and looking at posit32, one can see that the roundoff accumulation in the residual due to its recursive calculation is slightly better than without using the quire. This makes sense, because quires introduce deferred rounding. This is exploited specially in the calculation of norms and matrix-vector products. It also results in a somewhat better maintaining of A-orthogonality for the direction vectors.

However, the data points of posit16 bear little resemblance to its previous or later runs. It comes much closer to the target residual tolerance than in the last simulation, but it is still not reached. The tolerance is within the number range of posit16, even so it did not converge. The reason for this is that the smallest representable number in posit16 is 2^{-28}. The quire for posit16 has the same number range, despite the 128 bits in length. Every norm squared of a non-zero vector must be larger to equal to this number, because posit do not underflow. Therefore the norm is always larger or equal to $\sqrt{2^{-28}} = 2^{-14} \approx 6.1 \cdot 10^{-5}$. The tolerance of $7.8 \cdot 10^{-5}$ - even though larger than that number - is perhaps still too close. Comparing the lightpink values, that are posit16 as well, but the relative residual `res` is set to 10^{-5} instead (the tolerance being one order of magnitude larger), they converged after only `status=24` steps. This suggests that the reason for the strange behavior lies in the relative residual that was chosen too close to the lowest number above zero of the number regime.

Using the same arguments and analysis, posit8 had no chance to give a meaningful result.

In Figure 8, a smarter replacement was done. All variables that have a colletive role suffer from overflow. For example the norm of a vector $\vec{v} \in \mathbb{R}^n$ is

$$\|\vec{v}\| = \sqrt{\sum_{i=1}^{n} v_i^2}.$$

The number below the square root may be much bigger before squaring than after. If we calculate the norm in posit8, the result will be $\|\vec{y}\| \leq 8$. More importantly, when using a datatype that overflows such as binary16, the value after squaring might be perfectly fine, but the value under the square root could be outside the range of representable numbers, $\sqrt{\infty} = \infty$ and $\sqrt{0} = 0$. This is cured if the collective variable is of a datatype with larger number range than the underlying datatype that is summed over. In Figure 8 all collective variables were of type binary64.

The data of binary64 exhibits no significant alterations. Again comparing binary32 and posit32 with their previous data points, we see that the roundoff accumulation of binary32 is a little better and posit32 is approximately the same as with the quire, suggesting that when using posits utilizing the quire is probably sufficient.

Looking at tensorfloat32, it has the same exponent range as binary32, but less precision and it has the same number of mantissa bits as binary16, but at a higher exponent range. Compared to binary16, both datatypes have the same amount of numbers to be distributed in their respective number range. It is expected to perform worse or equal to binary32, but better or equal to binary16 and bfloat16. Therefore it's expected to converge in $27 \leq$ `status` ≤ 28 steps, see equation (8.2). This is indeed the case with `status=27` steps. We see that the larger number range compared to binary16 has little to do with speed of convergence. This is because the number regime is within the binary16 regime, except for collective variables. This explans as well why tensorfloat32 performed precisely as in the naive replacement, Figure 6, but the roundoff accumulation is better because of the more precise collective variables.

The bfloat16 with even less precision but comparable number range of tensorfloat32 converged in `status=28` steps as well, but needed two more reset steps, tightening the previous conclusion about speed of convergence.

The most interesting data points are the ones of binary16 and posit16 that both were able to converge in `status=28` and `status=34` steps respectively. They performed quite similar, even though it would be expected that posit16 would perform a slightly better because of the bigger number range and bigger number density in relevant number regimes (see Figure 4). In plot *3c* the increase of roundoff accumulation can be observed for binary16 and posit16 in steps where the real residue changes (where the algorithm makes progress, see for example: steps 1 to 10). Notice that, when the real residue stalls and the recusive residue still (wrongly) decreases, the roundoff accumulation will saturate until the order of magnitude of the two numbers becomes too large such that their difference is dominated by the larger number. This can the seen in the data points of posit16 in plot *3a*. It suggests that the precision limit was chosen too low for the datatype. Notice that the precision limit is defined to be 100 times the `MACHINE_EPSILON` of the datatype. The `MACHINE_EPSILON` for the posit datatypes is quite misleading, because it gives us (by definition) the precision of numbers around 1. This is the regime where posits are most precise, their precision falling off very rapidly when leaving it. Thus for posit16 in the regime 10^{-1} the `MACHINE_EPSILON` is correct (seen at iteration 14), whereas in the regime 10^{-3} it is chosen to small and we can see a staircase-shape around the reset steps at iterations 28 and 35. Such a stalling of the real residue should be avoided at any cost, because the algorithm stalls as well in that case. The `MACHINE_EPSILON` is defined to be the difference between 1 and the lowest number above 1. For floats this definition makes more sense, because their precision does not fall off that fast, but for posits which are most presice around 1 this gives a too precise value, not refecting the real precision of posits in their whole number range correctly. Instead, the machine epsilon should be a function of the number regime, increasing when going far away from 1. This is the reason for the staircase-shaped curve of posit16 in plot *3a*. The phenomenon is even more prominent for posit16 in plot *4a* of Figure 9. The posit32 does not have this problem, because its `MACHINE_EPSILON` is sufficient for the number regime used in the algorithm. When demanding lower relative residuals, staircase-shapes should be expected for posit32 as well.

Comparing binary16 with bfloat16 and tensorfloat32, we see again that exponent range is less relevant than precision. Presicion determines the amount of reset steps.

Figure 9 shows all the simulated datatypes using a collective datatype of binary64 just as in Figure 8, but with a relative resudial of 10^{-12} instead. This might be a more realistic scenario. The last row resembles the predicted hierarchy (8.2) particularly well. Notice that 10^{-12} is outside the representable number range of binary16, posit16 and posit8. This means that these datatypes have no chance to reach the target tolerance, therefore we expected them not to converge. This is indeed the case. We also see that binary16 and posit16 both are not able to go below 10^{-5}, meaning the tolerance in the third row was chosen very close to the minimum possible, but still converging tolerance (see also discussion of posit16 in Figure 7). Both datatypes make no further significant progress after step 45. It can also be seen that even the recusive residue stalls or increases - an indicator that the datatype has reached its limits.

The comparison between binary32 and posit32 is again of insight. Their difference is subtile. We see that both needed the same amount of steps. Roundoff accumulation and A-orthogonality are again slightly better, making posit32 the overall better 32-bit datatype for the problem. The reason for this goes down to the higher precision of posits in the relevant number regime. Looking at the lightgreen values, that are posit32 as well, but utilizing the quire instead of binary64 as collective variable, we observe the same amount of steps to convergence, but roundoff accumulation is slighly worse. It might be an unfair comparison, because binary64 as collective variable has more precision, surpassing even the deferred rounding employed by the 512-bit quire for posit32. In plot *4d* the posit32 with quire will not go below some fixed value. The reason for this is the lowest posit32 value with exponent bits `es=2` is 8^{-30} and the norm of a posit32-vector with at least one non-zero component must be bigger or equal to the square root of this; $1.15 \cdot 10^{-18}$. This suggests that when choosing `res` to be smaller than 10^{-18}, we expect posit32 not to converge anymore in analogy to posit16 in the second row.

Since binary16 was able to converge in Figure 8, this suggests that the number regime is within binary16 giving posit32 more precision in that regime over binary32

Finally, compare the 3 datatypes with the same exponent range, but different precisions; binary32, tensorfloat32 and bfloat16. The less precision, the slower the convergence. The price to go from 23 to 10 mantissa bits results in 1 more conjugate gradient step as well as 4 more reset steps. When going further down to 7 mantissa bits again 1 more regular step and 4 more reset steps where needed to finally bring bfloat16 to convergence after `status=56` regular conjugate gradient plus 10 reset steps. Bearing in mind that it uses only 16 bits, this is a remarkable result. It performed way better than its 16-bit competitors.

We also see in plot *4a* that all datatypes start to converge by the same speed (all slopes are equal). The actual residual of the datatype with the lowest precision, namely bfloat16 with 7 mantissa bits, resets first, followed by binary16 and tensorfloat32 which have both 10 mantissa bits. The next one is posit16, because it has more precision than binary16 in the relevant regime, followed by binary32 with 23 mantissa bits and later by posit32, where the same argument as before holds. The curve of binary64 would also reset at some point, but that is outside the scale.

Specially plot *4a* suggests that we can start to calculate in a datatype with 16 bits of length until we fall below a constant, to be determined value (that depends on the datatype), then continuing the calculation in a datatype with 32 bitlength until that number regime is exhausted as well, again switching to a 64 bit datatype to finish the caclulation.

8.5.2 8^4 lattice

In order to make sure that the previous analysis is consistent and the physics involved were relevant, the same data was extracted from a 8^4 lattice and some of the plots were remade from the new data, see Figure 10. Only the datatypes binary64, binary32 and binary16 were simulated. In priciple, the data tells the same story. The main difference to figures 8 and 9 is that more steps were needed to converge, because the Dirac matrix is much larger than before, although only 0.04% of all components were non-zero, compared to 2% in the 4^4 lattice of the previous analysis. In plots *2a* to *2f*, were the relative residue was chosen to be 10^{-12}, we again see the saturation of binary16 marking the lower limit of the datatype. After every reset step, a jump in roundoff accumulation can be seen, because the residual in the reset step is calculated in higher precision. It is interesting that the roundoff accumulation in the final steps of binary16 come very close to those of binary32 (see plot *1c*). A reason for this could be the clustering of reset steps just before convergence, giving very accurate results with little roundoff, even for less presice datatype. We also see that the speed of convergence does not significantly depend on the precision of the datatype, only the amount of reset step does, thus the less steep slope of binary16. When the lower limit of the datatype is reached, the slope becomes zero and the residual shows no striking reduction anymore. This is were the datatype should be switched to one with a larger number range.

8.5.3 Conclusion

The desicion between floats and posits is not trivial. It highly depends on how fast the machine can perform FLOPS and POPS. For example division in floating point arithmetic is very expensive (it may exceed 24 CPU cycles, many compiler optimizations evade them), whereas in posit arithmetic it is said to be cheap, because obtaining the inverse of a number is easy.

Another example could be that comparisons between floats are more expensive than for posits. Two posits are equal if their bit representations are equal. Comparing two floats is much more expensive, mainly because of the many NaNs and since 0 and -0 are equal but not bit-identical.

On the other hand, there is currently no hardware available, that has dedicated posit units and posits are not studied as intensive as floats. Floats are widespread, well understood and implemented in common hardware.

If one desides to replace binary32 with posits, the most elegant solution would be to naively replace the datatype and utilize quires in collective operations. To use binary64 collective variables is not recommended, because this would introduce many type conversions between the floating point and the posit format which is assumed to be expensive. The drawback ot this method is that posit16 may only converge if the relative residue is chosen high enough (see plot *2a* in Figure 7).

If the desicion goes for floats, which might be the more realistic scenario, then the most elegant solution would be to use collective variables in binary64. Type conversions between different IEEE floating point types are not considered to be expensive. The tensorfloat32 compared to binary32

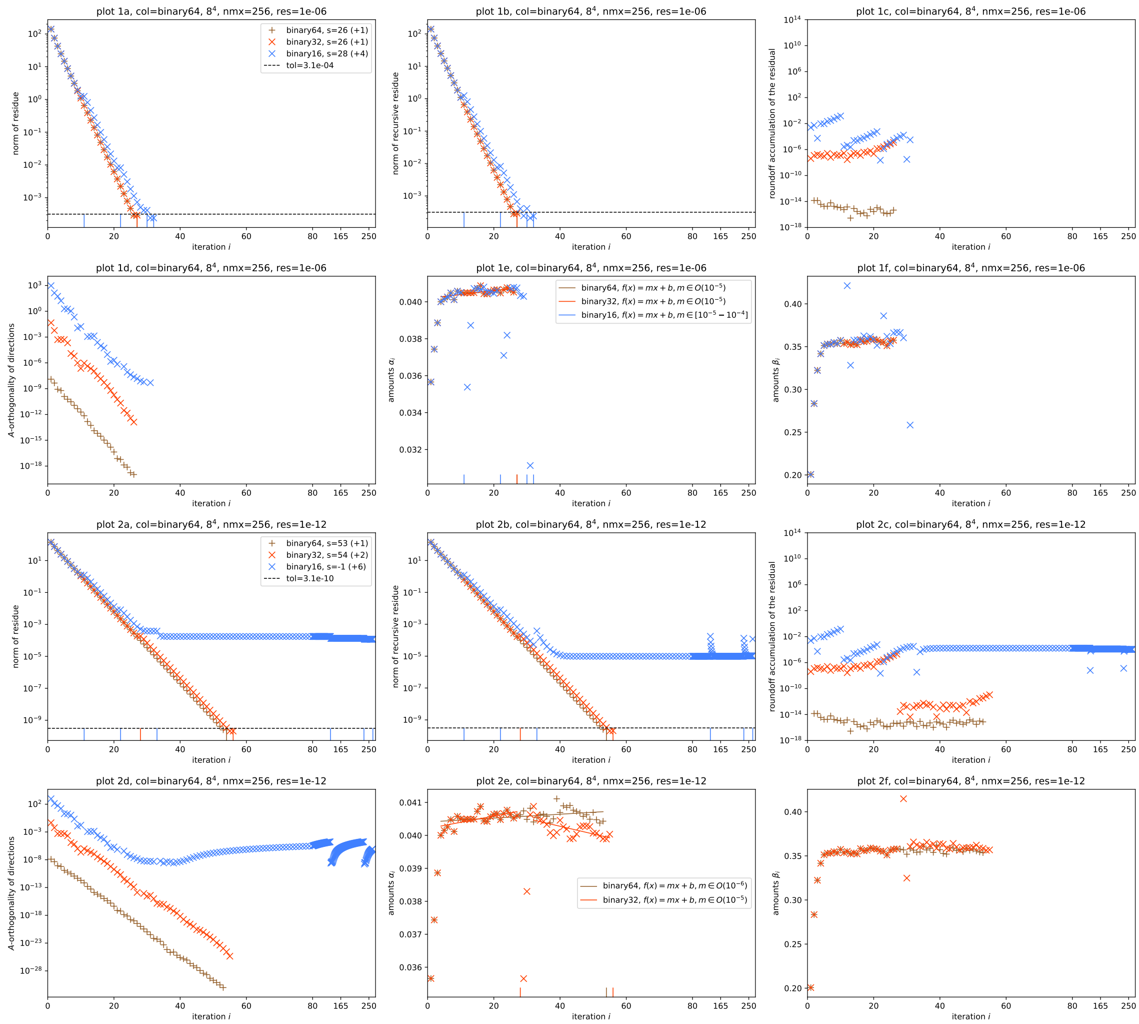

Figure 10: In anology to figures 8 and 9. This time an 8^4 lattice was used and only the floating point datatypes that are available in hardware nowadays were simulated. The *first and second row* use binary64 as collective variable and 10^{-6} was the desired relative residual. The *third and fourth row* have the exact same setup, but with a relative residual of 10^{-12} instead.

and bfloat16 answers the question how important precision is in the calculation. All of them have the same number of exponent bits and therefore approximately the same number range, but very different precisions. We see that all of them were able to converge in any experiment, but with binary64 as collective variable, the results were closest to each other (see Figure 8 plot *3a*). The only real difference was in the amount of reset steps. If the datatype is lower in bitlength, the memory-boundedness suggests that the calculation performs faster, but the tradeoff is the amount of (computationally expensive) reset steps that increases with less precision. However, the datatype for collective operations should be precise and should have a large number range. Since the amount of variables needed in that datatype does not scale with the lattce size, it is perfectly right to use a datatype with large bitlength. Comparing the convergence of bfloat16 in the naive case (Figure 6 plot *1a*) with the case binary64 collective variables (Figure 8 plot *3a*), it can be seen that the algorithm converged 21 steps faster, only because the collective datatype was chosen to be binary64. On the other hand, comparing the performance of binary16 in the two plots, we see that the number range of the collective datatype brought binary16 from no convergence to convergence within `status=35` steps - only marginally slower than binary32. These arguments make binary64 the best choice for variables with a collective role.

Proposal 8.1: Mixed Precision

The above analysis suggests that the calculation of the solution can be (at least partly) conducted in an even less precise datatype than binary32. One could for example choose 3 datatypes with different precision. The algorithm can be started using the least precise one. If the tolerance hits a certain value at the boundaries of the datatype, the algorithm switches to the next higher one. The calculation is continued in that datatype until the tolerance reaches the limits of the new datatype. Again the datatype is switched to the next higher one[a]. This calculation in mixed precision is not dependent on the algorithm itself and can therefore be applied to every iterative solver algorithm. Algorithm 1 shows an example implementation of such a mixed precision calculation. The array d consists of all available datatypes participating in the calculation in ascending order, meaning the least precise datatype comes first. The function *solve*() performs the underlying algorithm (for example conjugate gradient) in the datatype given by its arguments. It expects at least a starting vector $\vec{x_0}$ and a tolerance and returns the status [b], the calculated solution and the residual up to the given tolerance.

Algorithm 1: Pseudo-code for an iterative algorithm in mixed precision.

input: desired norm of relative residual rn
input: array of datatypes in $\{d\}_{k=0}^{N}$
input: iterative algorithm *solve*()
$\vec{x_0}, \vec{r_0}, \ldots \leftarrow$ initial guess, $\ldots$;
$\vec{x}, \vec{r} \leftarrow \vec{x_0}, \vec{r_0}$;
status $\leftarrow 0$;
for $k \leftarrow 0, 1$ **to** N **do**
    convert all variables to datatype $d[k]$;
    $tol \leftarrow \frac{1}{\|\vec{r_0}\|} max(rn, \texttt{MACHINE\_EPSILON of } d[k])$;
    substatus, $\vec{x}, \vec{r}, \ldots \leftarrow solve(tol, \vec{x}, \ldots)$;
    if *substatus* > 0 **then**
        status $\leftarrow$ status + substatus;
    if $\|\vec{r}\| < rn$ **then**
        return status, $\vec{x}$; `// success`
end
status $\leftarrow -3$;
return status, $\vec{x_0}$; `// the algorithm failed`

[a] One obvious choice could be $d = \{\text{binary64}, \text{binary32}, \text{binary16}\}$. When the algorithm is started in binary16 and a tolerance of $\approx 10^{-4}$ is reached, the algorithm continues in binary32, the limit of which is at a tolerance of $\approx 10^{-35}$. A continuing calculation would then be conducted in binary64.

[b] See section 8.4

Proposal 8.2: Approximating the amounts α_i

Looking at plot *4e* of Figure 9, where the amounts α_i are plotted for every iteration, we see that after every reset step the amounts need $2-3$ steps to reach a value that is not changing very much for future iterations. This is becomes apparent when looking at the fitting lines. The values of the α_i are in the range 10^{-1} and the slopes m of the fitting lines are in the range 10^{-4} - 10^{-5}, suggesting that the value of α_i is not changing from iteration to iteration when only looking at $2-3$ significant decimal digits.
A possibility to reduce computational cost in each iteration could be to approximate the values of future α_i to be constant. The less precise the datatype, the larger the change in α_i. The large error in α_i of bfloat16 in all plots suggests that the algorithm is not sensible to errors in α_i. Therefore, it can be expected that the results should not change significantly with a approximated value of α_i.

- Advantage: The resoduals can be calculated using $\vec{b} - A\vec{x}$, not recusively. This implies less roundoff accumulation.
- Advantage: Only one matrix-vector multiplication per iteration.
- Disadvantage: Since the α_i are just approximated, the number of needed iterations may increase.
- Disadvantage: The Dirac operator D must be given in the form of $A = D^\dagger D$ as *one* operator, else the algorithm still consists of 2 matrix-vector multiplications per iteration. Also, $D^\dagger D$ is less sparse than D.

The results of simulations with approximated values for the α_i can be observed in plot series 11 and 12. The value was approximated based on previous values. The first 5 steps where skipped (thus the algorithm performed natively). In step number 5, the last 3 values of α_i where averaged. In the following steps the constant value calculated in step 5 was reused. After every reset step, the value of α_i had to be recalculated using the above procedure. Therefore a datatype such as bfloat16 that has reset steps after approximately every 7th regular step, will benefit in only 2 steps per reset step. This is very little difference to native runs compared to datatypes with high precision.
The calculation became more sensible to the number range of the datatype. This can be seen in all plots when looking at binary16 that was not able to converge anymore, although by a very small amount. tensorfloat32 on the other hand performed very similar to the regular rounds, it was expected that it needs slightly more iterations. When going with this strategy, it is therefore advisable to perform more regular cg-steps when coming closer to the boundaries of the datatype. One possible solution would be to choose a higher machine epsilon close to the boundaries, forcing the algorithm to perform more reset steps, in turn causing more regular cg-steps and recalculations of α_i.
Notice that with larger lattice sizes, the approximation of the amounts has less error (see plots *1e* and *2e* in figures 11 and 12) and the algorithm is thus more stable.

9 SAP preconditioned GCR algorithm

The next solver appearing in openQxD is called `SAP_GCR`. It makes use of a multiplicative Schwarz Alternating Procedure (SAP) as preconditioner for a flexible Generalized Conjugate Residual (GCR) run.

TODO: motivation: parallel processing, chrial regime (spontaneous breaking of chiral symmetry), simulation containing sea-quarks limited to small lattices and large quark masses.

9.1 Even-Odd Preconditioning

Preconditioning in general, when employed in lattice QCD, is expected to have significant impact on the number of iterations of a solver. One way of preconditioning $D\psi = \eta$ on a lattice is

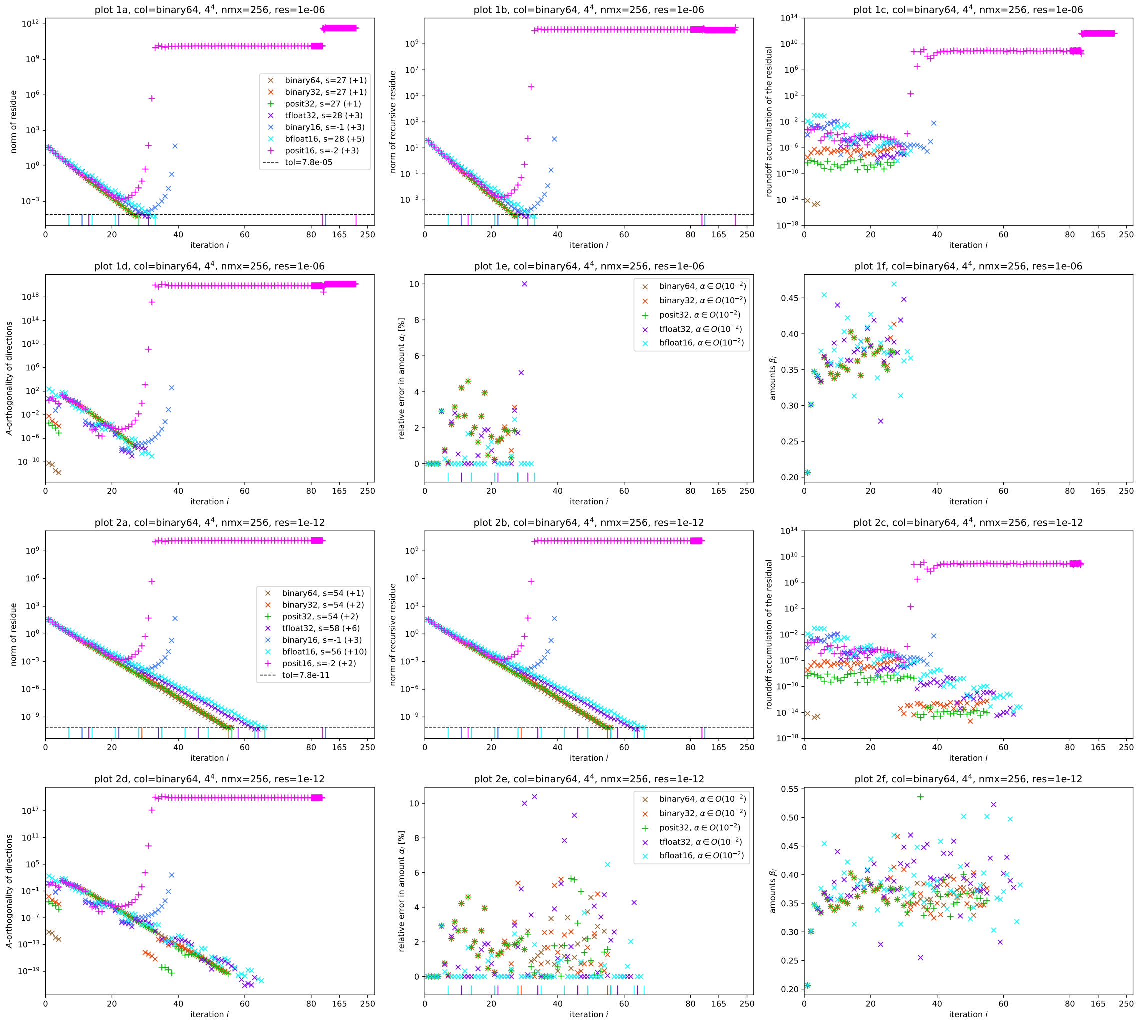

Figure 11: Plots *1a* to *1f* contain the convergence analysis of a conjugate gradient run with a 4^4 lattice, relative residual 10^{-6} and approximated values of α_i. In plots *2a* to *2f* the residual was chosen to be 10^{-12}. Plots *1e* and *2e* contain the relative error in the approximated α_i compared to the real α_i.

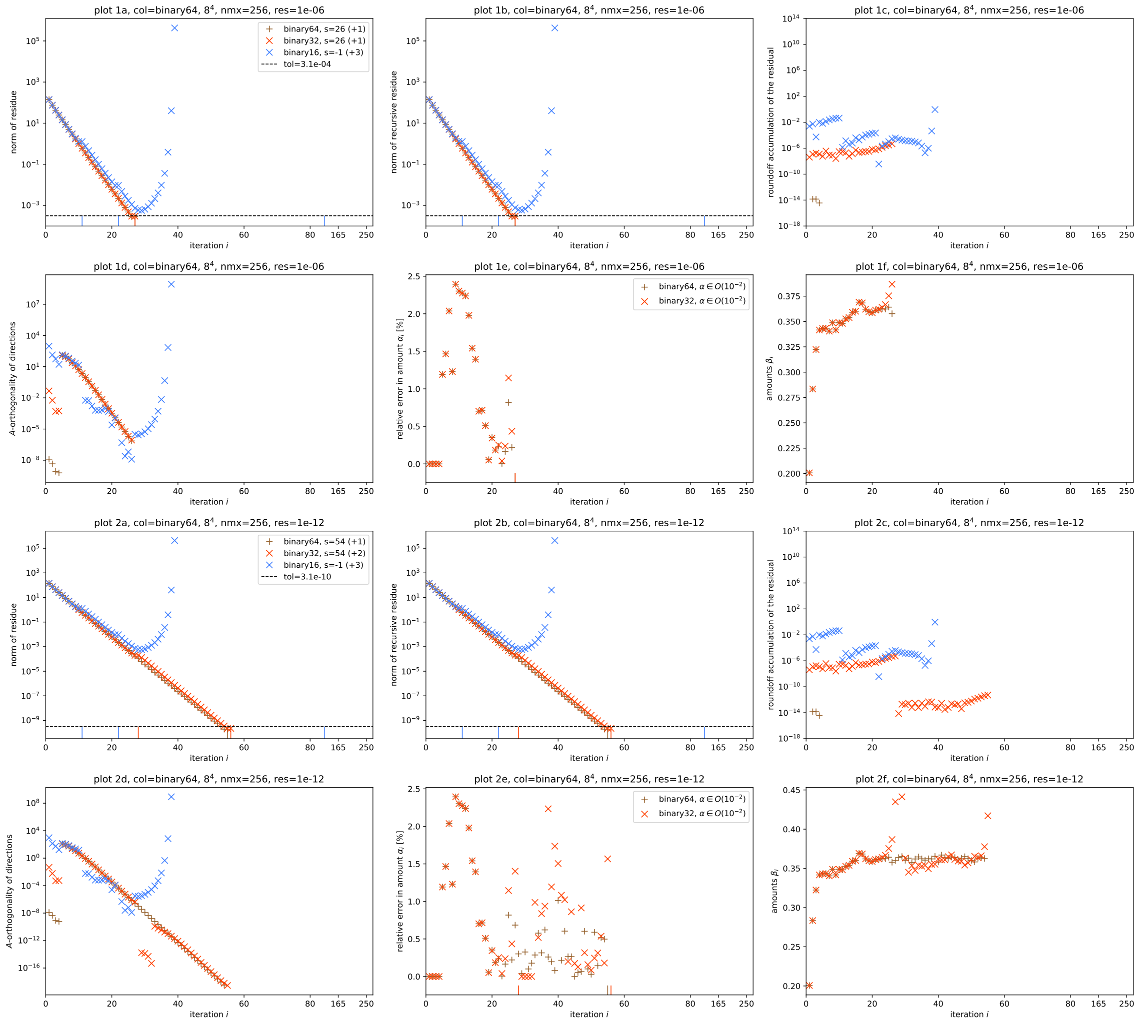

Figure 12: The same setup as figure 11, but with a 8^4 lattice.

$$LDR\psi' = L\eta,$$

with $\psi = R^{-1}\psi'$ and L, R chosen wisely such that LDR is well conditioned. If $L = \mathbb{I}$, it is called ***right preconditioning***, if $R = \mathbb{I}$ it is called ***left preconditioning***. If the Dirac-matrix involves only nearest-neighbor interactions it is possible to split the lattice into even and odd sites[11] [12]. If the sites are ordered such that the even sites come first[13],

$$D = \begin{pmatrix} D_{ee} & D_{eo} \\ D_{oe} & D_{oo} \end{pmatrix}, \qquad \psi = \begin{pmatrix} \psi_e \\ \psi_o \end{pmatrix}$$

D_{ee} (D_{oo}) consists of the interactions of the even (odd) sites among themselves, whereas D_{eo} and D_{oe} consider the interactions of even with odd sites. ψ_e and ψ_o contain the values for even and odd lattice sites of the spinor.

Using specific forms of L and R, D can be brought in a block-diagonal form, namely

$$L = \begin{pmatrix} 1 & -D_{eo}D_{oo}^{-1}D_{oe} \\ 0 & 1 \end{pmatrix} \qquad \text{and} \qquad R = \begin{pmatrix} 1 & 0 \\ -D_{oo}^{-1}D_{oe} & 1 \end{pmatrix}.$$

After a bit of algebra,

$$LDR = \begin{pmatrix} \hat{D} & 0 \\ 0 & D_{oo} \end{pmatrix}, \qquad \text{with} \qquad \hat{D} = D_{ee} - D_{eo}D_{oo}^{-1}D_{oe}.$$

This specific preconditioning reduces the amount of iterative steps needed by a factor of 2 approximately, because D_{oo} and $\hat{D}$ are matrices of half the dimension of D. The inversion of D_{oo} is simple, because with only nearest-neighbor-interactions the odd sites do not interact among themselves, only with even sites. Thus D_{oo} exhibits block-diagonal form (all blocks are 6×6, why?). Using

$$D\psi = \eta \implies \begin{pmatrix} D_{ee} & D_{eo} \\ D_{oe} & D_{oo} \end{pmatrix} \begin{pmatrix} \psi_e \\ \psi_o \end{pmatrix} = \begin{pmatrix} D_{ee}\psi_e + D_{eo}\psi_o \\ D_{oe}\psi_e + D_{oo}\psi_o \end{pmatrix} = \begin{pmatrix} \eta_e \\ \eta_o \end{pmatrix}$$

we can write the preconditioned form, where only the reduced system with even lattice sites has to be solved to determine ψ_e

$$\begin{aligned} \hat{D}\psi_e &= D_{ee}\psi_e - D_{eo}D_{oo}^{-1}D_{oe}\psi_e \\ &= (\eta_e - D_{eo}\psi_o) - D_{eo}D_{oo}^{-1}(\eta_o - D_{oo}\psi_o) \\ &= \eta_e - D_{eo}D_{oo}^{-1}\eta_o, \end{aligned}$$

because ψ_o follows from the solution ψ_e via

$$\psi_o = D_{oo}^{-1}(\eta_o - D_{oe}\psi_e).$$

9.2 Schwarz Alternating Procedure

Domain decomposition is a way to partition the large system into (possibly many) smaller subproblems with regularly updated boundary conditions coming from solutions of neighboring subproblems. They fit very well into the notion of parallel processing, because the subproblem can be chosen to

[11] It is therefore very similar to a domain decomposition method, see later.

[12] Even lattice points are the ones where the sum of the global cartesian coordinates ($x_0 + x_1 + x_2 + x_3$) in units of the lattice spacing a is even.

[13] This is indeed the case in openQxD (see `main/README.global`) in [3].

be contained in one single rank. The full lattice is split into sublattices called ***local lattice***. Each rank has its own local lattice, the size of which is determined at compilation time. The full lattice consists of the ensemble of all local lattices arranged in a grid. It is therefore advisable to choose the size of decomposed subdomains as a divisor of the local lattice size such that one or more blocks fit into one rank. These subproblems can then be solved using a iterative solving method.

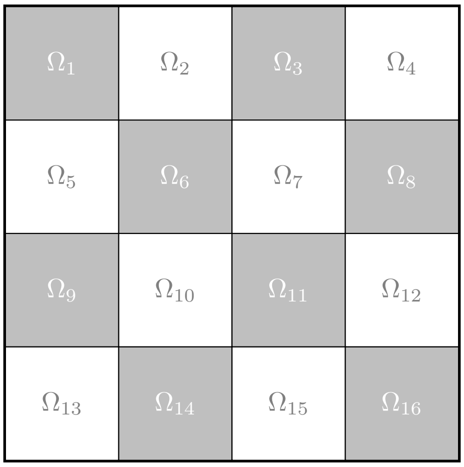

Figure 13: $d = 2$ dimensional example of a decomposition of a lattice Ω into domains named Ω_i.

The idea behind SAP is to loop through all blocks Ω_i and solve the smaller subproblem using boundary conditions given from the most recent global solution (see figure 13). If the original problem only includes nearest-neighbor interactions, the solution of a block Ω_i depends only on that block and its exterior boundary points, which are the adjacent points on the neighboring blocks with opposite color. For example, the solution of the subproblem involving Ω_6, depends only on the solutions of Ω_2, Ω_5, Ω_7 and Ω_{10}[14]. Therefore all gray (white) subproblems can be solved simultaneously, with the most recent boundary conditions obtained from the white (gray) domains. Solving all gray, followed by all white subproblems is called a ***Schwarz cycle*** and is considered one iteration in the SAP. Each subproblem can be solved with a desired solver separately, again applying some preconditioning[15].

9.3 SAP as a Preconditioner

The multiplicative Schwarz Alternating Procedure is such a domain decomposition method coming from the theory of partial differential equations. It can be applied in the form of a right preconditioner M^{-1} making the preconditioned system

$$M^{-1}A\vec{x} = M^{-1}\vec{b} \tag{9.1}$$

to be solved in very few steps, if M^{-1} is a good approximation for A^{-1}. The preconditioning matrix M^{-1} although is never explicitly available during the calculation, such as it is the case in even-odd preconditioning which can also be applied in advance. In order to solve the preconditioned equation (9.1) using a iterative Krylov subspace method, the algorithm must be able to apply M^{-1} and $M^{-1}A$ to an arbitrary vector $\vec{v}$. If it is possible to implement such operations on multiple ranks in an efficient way and if the preconditioner makes $M^{-1}A$ well conditioned[16], we reached the goal. Obviously an application of M^{-1} should be possible without involving A^{-1}. The actions of operators M^{-1} and $M^{-1}A$ on a vector $\vec{v}$ are assembled using a multiplicative Schwarz Alternating Procedure, where the blocks are treated by some fixed number of Minimal Residual (MR) steps[17].

[14] It depends on all other subproblems as well, but only indirectly.

[15] Using even-odd preconditioning is perfectly fine with D replaced by the restricted Dirac operator D_i acting only on the points in Ω_i.

[16] Would it be dingy to expect such a thing from a preconditioner?

The blocks need not to be solved to a certain precision, because the procedure is only used as a preconditioner approximating the solution. This is a motivation for proposal 9.1.

In openQxD the `SAP_GCR` solver is implemented as follows: The large problem is solved using a flexible GCR solver, that in each of its `nmx` steps uses a different preconditioner. The preconditioner is given by `ncy` steps of the Schwarz Alternating Procedure applied to the current solution vector. Each SAP cycle involves approximately solving all gray followed by all white blocks on the whole lattice each with `nmr` steps of the MR method using even-odd preconditioning (`isolv=1`) or not (`isolv=0`).

Proposal 9.1: MR in reduced precision

Since MR is memory-bound, it can be conducted in mixed or reduced precision.

Proposal 9.2: Performing the MR steps on the GPU

The preconditioning procedure involves `mnr` Minimal Residual (MR) steps to be taken on each block in each Schwarz cycle to approximate a solution to the block problem. Since blocks of the same color are independent on each other and the Dirac operator acting only on a specific block involves no communication whatsoever, we can conclude that the procedure of solving a subproblem is a problem *local* to the block and self-contained in the sense that it can be solved independently and without MPI communication. This could be a very handy starting point when going towards GPU-utilisation. Once the souce vector and the restricted Dirac operator are transfered to the GPU, the problem can be solved on the GPU without involving any communication with other ranks or GPUs. This can also be beneficial, because of the following argument: The local lattice of one single rank, can be subdivided into multiple blocks as well (imagine figure 13 being the local lattice). The actual implementation solves the gray (white) blocks in a local lattice sequentially[a]. Since all the gray (white) problems within the local lattice can be solved simulaneously, the code does not exploit the full concurrency potential of the procedure. Solving the subproblems on the GPU, one could launch MR solvers on all gray blocks simultaneously followed by all white blocks. Keeping in mind proposal 9.1, the MR solver can be called in mixed or even reduced precision.

[a] By iterating over the blocks, see `sap()` at line 717ff in `modules/sap/sap.c` in [3].

9.4 Generalized Conjugate Residual algorithm

TODO: Why GCR? it allows inexact preconditioning without compromising the correctness of the solution.

We wish to solve (7.1) if A is not Hermitian. Comparing to the conjugate gradient algorithm, we minimize the residual $\vec{r}$ of the solution $\vec{x}$, using the ***quadratic form***,

$$\begin{aligned} f(\vec{x}) &= \frac{1}{2}\left(\vec{b} - A\vec{x}\right)^{\dagger}\left(\vec{b} - A\vec{x}\right) + c \\ &= \frac{1}{2}\left\|\vec{b} - A\vec{x}\right\|^{2} + c \\ &= \frac{1}{2}\|\vec{r}\|^{2} + c. \end{aligned}$$

where $c \in \mathbb{C}$. When taking the derivative of this function with repect to $\vec{x}$, we find that

$$f'(\vec{x}) = A^{\dagger}A\vec{x} - A^{\dagger}\vec{b}.$$

Lemma 9.1 (Uniqueness of the solution). *The solution $\vec{x}$ in equation (7.1) is unique and the global minimum of $f(\vec{x})$, if A is non-singular.*

[17] Determined by the value of `mnr` in the solver section of the input file.

Proof. Let us rewrite $f(\vec{p})$ at an arbitrary point $\vec{p} \in \mathbb{C}$ in terms of the solution vector $\vec{x}$,

$$\begin{aligned}
f(\vec{p}) &= \frac{1}{2}\left(\vec{b} - A\vec{p}\right)^\dagger \left(\vec{b} - A\vec{p}\right) + c + f(\vec{x}) - f(\vec{x}) \\
&= f(\vec{x}) + \frac{1}{2}\vec{p}^\dagger (A^\dagger A)\vec{p} - \frac{1}{2}(A\vec{p})^\dagger \vec{b} - \frac{1}{2}\vec{b}^\dagger (A\vec{p}) + \frac{1}{2}\vec{b}^\dagger \vec{b} \\
&= f(\vec{x}) + \frac{1}{2}(\vec{p} - \vec{x})^\dagger (A^\dagger A)(\vec{p} - \vec{x}) + \frac{1}{2}(A\vec{p})^\dagger (A\vec{x}) + \frac{1}{2}(A\vec{x})^\dagger (A\vec{p}) - \frac{1}{2}(A\vec{x})^\dagger (A\vec{x}) \\
&\quad - \frac{1}{2}(A\vec{p})^\dagger \vec{b} - \frac{1}{2}\vec{b}^\dagger (A\vec{p}) + \frac{1}{2}\vec{b}^\dagger \vec{b} \\
&= f(\vec{x}) + \frac{1}{2}(\vec{p} - \vec{x})^\dagger (A^\dagger A)(\vec{p} - \vec{x})
\end{aligned}$$

where to obtain the last line, $A\vec{x} = \vec{b}$ as used, thus the term simplified.
In the new form of $f(\vec{p})$, one can directly see that, $\vec{x}$ must minimize the function:

$$\begin{aligned}
f(\vec{p}) &= f(\vec{x}) + \frac{1}{2}(\vec{p} - \vec{x})^\dagger (A^\dagger A)(\vec{p} - \vec{x}) \\
&= f(\vec{x}) + \frac{1}{2}\underbrace{\|A(\vec{p} - \vec{x})\|^2}_{> 0 \text{ for } \vec{p} \neq \vec{x}}.
\end{aligned} \tag{9.2}$$

Therefore $\vec{x}$ is the global unique minimum if A is non-singular.

□

Remark. Notice the similarity of the above equation (9.2) to the analogue of the conjugate gradient algorithm (7.2). The only difference is the substitution of $A \longmapsto A^\dagger A$. It is therefore advisable in the derivation of an algorithm to require the directions $\vec{p}_i$ to be $A^\dagger A$-orthogonal instead of A-orthogonal.

In the same manner as in the derivation of the method of conjugate gradient, we impose a iterative ***step equation*** to be

$$\vec{x}_{i+1} = \vec{x}_i + \alpha_i \vec{p}_i,$$

again with ***directions*** $\vec{p}_i$ and ***amounts*** α_i that have to be determined. The recursively calculated ***residual*** has again the same formula

$$\vec{r}_{i+1} = \vec{r}_i - \alpha_i A\vec{p}_i.$$

Imposing $A^\dagger A$-orthogonality instead of regular A-orthogonality between error $\vec{e}_{i+1}$ and direction $\vec{p}_i$,

$$\begin{aligned}
0 &\stackrel{!}{=} \vec{e}_{i+1}^\dagger (A^\dagger A)\vec{p}_i \\
&= (\vec{e}_i + \alpha_i \vec{p}_i)^\dagger A^\dagger A\vec{p}_i
\end{aligned}$$

gives an expression for the amounts α_i. Notice the above equation is equivalent to imposing A-orthogonality $0 = \vec{r}_{i+1}^\dagger A\vec{p}_i$. However, we find (compare equation (7.9))

$$\alpha_i = \frac{\vec{r}_i^\dagger (A\vec{p}_i)}{\vec{p}_i^\dagger (A^\dagger A)\vec{p}_i} = \frac{\vec{r}_i^\dagger (A\vec{p}_i)}{\|A\vec{p}_i\|^2}.$$

The GCR algorithm does store all previous direction $\vec{p}_i$ as well as $A\vec{p}_i$ in contrast to conjugate gradient. Thus the derivation changes slightly. Let's continue with the determination of the directions using ***Gram-Schmidt orthogonalisation*** by imposing $A^\dagger A$-orthogonality instead of A-orthogonality and without imposing all previous β_{ij} to be zero (see definition 7.4). Likewise, we set $\vec{u}_i = \vec{r}_i$ and find

$$\vec{p}_0 = \vec{r}_0$$
$$\vec{p}_{i+1} = \vec{r}_{i+1} + \sum_{j=0}^{i} \beta_{ij}\vec{p}_j,$$

with

$$\beta ij = -\frac{\vec{r}_{i+1}^\dagger A^\dagger A\vec{p}_j}{\vec{p}_j^\dagger A^\dagger A\vec{p}_j} = -\frac{(A\vec{r}_{i+1})^\dagger (A\vec{p}_j)}{\|A\vec{p}_j\|^2}.$$

Using the above equations, we find the final form of the ***Generalized Conjugate Residuals Method***.

Definition 9.1 (Generalized Conjugate Residuals Method)**.** *The iteration step equation of the* ***Generalized Conjugate Residuals Method*** *in defined as*

$$\vec{x}_{i+1} = \vec{x}_i + \alpha_i\vec{p}_i, \tag{9.3}$$

with

$$\vec{r}_{i+1} = \vec{r}_i - \alpha_i A\vec{p}_i, \qquad \alpha_i = \frac{\vec{r}_i^\dagger (A\vec{p}_i)}{\|A\vec{p}_i\|^2}, \tag{9.4}$$

$$\vec{p}_{i+1} = \vec{r}_{i+1} + \sum_{j=0}^{i} \beta_{ij}\vec{p}_j, \qquad \beta ij = -\frac{(A\vec{r}_{i+1})^\dagger (A\vec{p}_j)}{\|A\vec{p}_j\|^2}, \tag{9.5}$$

and initial starting vectors

$$\vec{x}_0 = \textit{arbitrary starting point},$$
$$\vec{p}_0 = \vec{r}_0 = \vec{b} - A\vec{x}_0.$$

There are some remarks to note about the method of GCR.

Remark. After calculating $\vec{r}_{i+1}$ and $A\vec{r}_{i+1}$, we can recursively determine $A\vec{p}_{i+1}$ via

$$A\vec{p}_{i+1} = A\vec{r}_{i+1} + \sum_{j=0}^{i} \beta_{ij} A\vec{p}_j. \tag{9.6}$$

This limits the number of matrix-vector products to one per iteration.

Remark. All previous $\vec{p}_i$ and $A\vec{p}_i$ need to be stored in memory in order to construct the next $\vec{p}_{i+1}$ and $A\vec{p}_{i+1}$.

Remark. Comparing to the conjugate gradient algorithm, we imposed $A^\dagger A$-orthogonality of the directions $\vec{p}_i$ instead of A-orthogonality as well as A-orthogonality of $\vec{r}_{i+1}$ and $\vec{p}_i$ instead of regular orthogonality. A vanishing of all previous β_{ij} on the other hand was not imposed, leading to the sum in the step equation of $\vec{p}_{i+1}$.

9.5 GCR in openQxD

The actual implementation of the GCR algorithm in openQxD is quite different[18], but actually equivalent to definition 9.1 (see lemma 9.2). Ref. [9] explains the implementation of the algorithm in detail. The main GCR-loop looks as in Algorithm 2 (see Figure 3 in [9])

[18]See `fgcr()` in `modules/linsolv/fgcr.c` lines 212ff in [3].

Algorithm 2: Pseudo-code for the GCR recursion.

1 $\rho_0 = \eta$;
2 **for** $k \leftarrow 0, 1, 2$ **to** n_{kv} **do**
3 $\quad \phi_k = M_{sap}\rho_k$;
4 $\quad \chi_k = D\phi_k$;
5 $\quad$ **for** $l \leftarrow 0$ **to** $k-1$ **do**
6 $\quad\quad a_{lk} = (\chi_l, \chi_k)$;
7 $\quad\quad \chi_k = \chi_k - a_{lk}\chi_l$;
8 $\quad$ **end**
9 $\quad b_k = \|\chi_k\|$;
10 $\quad \chi_k = \frac{\chi_k}{b_k}$;
11 $\quad c_k = (\chi_k, \rho_k)$;
12 $\quad \rho_{k+1} = \rho_k - c_k\chi_k$;
13 **end**

In algorithm 2, M_{sap} is the SAP preconditioner, that might depend on the iteration number k as well, making the algorithm flexible. D is the Dirac-operator and ρ_k the residual in the k-th step. The algorithm does not include an update of the solution vector ψ_{k+1}, instead this is done after n_{kv} iterations all at once,

$$\psi_{k+1} = \sum_{l=0}^{k} \alpha'_l \rho_k. \tag{9.7}$$

Lemma 9.2. *The iterative algorithm from definition 9.1 is equivalent to algorithm 2 when setting the preconditioning operator $M_{sap} = \mathbb{I}$, the Dirac-matrix $D = A$, the source vector $\eta = \vec{b}$ and the solution vectors $\psi_k = \vec{x}_k$.*

Proof. Noticing that the residual $\rho_k = \vec{r}_k$ from line 12 in algorithm 2 and in definition 9.1 must be identical, we find that χ_k must be proportional to $A\vec{p}_k$. Before the normalization in line 10, we have $\chi_k = A\vec{p}_k$. The $b_k = \|\chi_k\|$ are set before normalisation of χ_k, therefore $b_k = \|\chi_k\| = \|A\vec{p}_k\|$. Using this we find $a_{lk} = (\chi_l, D\rho_k)$ and since $l < k$ the χ_l are normalied, thus $\chi_l = b_l A\vec{p}_l$ after line 10. Thus $a_{lk} = (A\vec{p}_l, D\rho_k)/b_l = -\beta_{k-1,l}\|A\vec{p}_l\|$. Finally, the c_k are defined after normalization of the χ_k, therefore they evaluate to $c_k = (\chi_k, \rho_k) = (A\vec{p}_k, \vec{r}_k)/b_k = \alpha_k\|A\vec{p}_k\|$. Using these substitutions we find the same formulas as in definition 9.1, except for the step equation.

The main difference between the step equations (9.3) and (9.7) is that in the former the solution $\vec{x}_{i+1}$ is spanned by the direction vectors $\vec{p}_i$, whereas in the latter it is spanned by the residuals $\rho_i = \vec{r}_i$. This is not a problem since both sets of vectors span the same space, but the amounts α'_l in equation (9.7) differ heavily from the amounts α_i in equation (9.4).

To determine the amounts α'_l in terms of α_i and β_{ij}, we notice equation (9.6),

$$A\vec{p}_i = A\vec{r}_i + \sum_{j=0}^{i-1} \beta_{i-1,j} A\vec{p}_j \iff b_i\chi_i = D\rho_i - \sum_{j=0}^{i-1} a_{ji}\chi_j \tag{9.8}$$

and the fact that

$$\rho_{k+1} = \eta - \sum_{l=0}^{k} c_l\chi_l. \tag{9.9}$$

But also

$$\rho_{k+1} = \eta - D\psi_{k+1}$$

$$
\begin{aligned}
&= \eta - \sum_{l=0}^{k} \alpha_l' D \rho_k \\
&= \eta - \sum_{l=0}^{k} \alpha_l' \left[b_k \chi_k + \sum_{j=0}^{k-1} a_{jk} \chi_j \right], \qquad (9.10)
\end{aligned}
$$

where in the last step equation (9.8) was inserted. The $\chi_i \propto A\vec{p}_i$ are linearly independent, thus the coefficients from (9.10) can be compared to (9.9), giving for $m = 0, 1, \dots, k$

$$
\begin{aligned}
\alpha_m' &= \frac{1}{b_m} \left[c_m + \sum_{l=m+1}^{k} \alpha_l' a_{ml} \right] \\
&= \alpha_m - \sum_{l=m+1}^{k} \alpha_l' \beta_{l-1,m}.
\end{aligned}
$$

□

Proposal 9.3: GCR in mixed precision

In the current version of openQxD **??**, the outer GCR solver is performed in pure binary64. A mixed precision variant would need the preconditioning M_{sap} to be done in mixed precision as well. Algorithm 1 would directly apply with $solve()$ replaced by `fgcr()` with the difference that `fgcr()` has to accept D, M_{sap}, $\vec{x}_0$ and $\vec{b}$ in the desired precision.

10 Deflated SAP preconditioned GCR algorithm

The low modes of the Dirac operator condensate TODO.

Small quark masses corresponding to real physics are believed to be the cause for the spontaneously breaking of chiral symmetry in lattice QCD [1]. Numerical lattice QCD has the problem that with large lattice volumes and small quark masses simulation techniques become inefficient in the ***chiral regime*** (where chiral symmetry is spontaneously broken). According to the Bank-Casher relation [1], this is because the number of eigenvalues of D below a fixed value grows with $O(V)$, where V is the total 4D lattice volume. On the other hand, the computational effort scales even worse with $O(V^2)$ [10]. This behavior goes under the name of ***V^2-problem***.

A solving algorithm that has a flat scaling in with respect to the quark masses can therefore lead to large speedups specially in that regime. By deflating the Dirac operator, it is possible to separate eigenmodes with very small eigenvalues from the others. Thus the space needs to be split in low and high modes without actually calculating the modes, else the problem would be solved already.

10.1 Deflation

Theorem 10.1 (Deflation)**.** *Let A be a linear, invertible operator acting on a vector space Λ, $\vec{b} \in \Lambda$ a arbitrary vector and P_L a projector*[19] *acting on Λ. Also, define the linear operator P_R such that $P_L A = A P_R$*[20]*. Consider*

$$
\vec{x}^\star := P_R \vec{x}_1^\star + (1 - P_R)\vec{x}_2^\star, \qquad (10.1)
$$

with $\vec{x}_1^\star$ and $\vec{x}_2^\star$ being solutions to the "smaller" (projected) systems

$$
P_L A \vec{x}_1 = P_L \vec{b} \qquad \textit{and} \qquad (1 - P_L) A \vec{x}_2 = (1 - P_L)\vec{b}
$$

respectively. Then

[19] P_L does not have to be orthogonal or hermitian.

[20] Such a linear operator P_R always exists - just set $P_R := A^{-1} P_L A$, since A is invertible.

1) P_R is a projector,

2) $\vec{x}^\star$ is a solution to $A\vec{x} = \vec{b}$.

Proof. Using that $P_L^2 = P_L$ is a projector,

$$\begin{aligned} P_R^2 &= (A^{-1}P_L A)^2 \\ &= A^{-1}P_L^2 A \\ &= A^{-1}P_L A \\ &= P_R. \end{aligned}$$

By direct calculation,

$$\begin{aligned} A\vec{x}^\star &= AP_R\vec{x}_1^\star + A(1-P_R)\vec{x}_2^\star \\ &= P_L A\vec{x}_1^\star + (1-P_L)A\vec{x}_2^\star \\ &= P_L\vec{b} + (1-P_L)\vec{b} \\ &= \vec{b}. \end{aligned}$$

□

Remark. Therefore, if we find clever projectors P_L and P_R without involving A^{-1}, we can solve $A\vec{x} = \vec{b}$ by solving the 2 smaller systems of equations and then projecting the solutions using P_R.

Remark. Notice that $P_L A$ as well as $(1 - P_L)A$ are not invertible, therefore there are infinitely many solutions $\vec{x}_1^\star$ and $\vec{x}_2^\star$[21]. Nonetheless the solution vector $\vec{x}^\star$ is still unique after the projection in equation (10.1).

Remark. Comparing deflation to left preconditioning, the difference is that in deflation P_L is a projector and $P_L A$ has condition number infinite whereas in case of preconditioning P_L is invertible and the condition number of $P_L A$ is expected to be smaller than of A.

Corollary 10.2. *Let A and $\vec{b}$ be as in theorem 10.1. Furthermore let $\{\vec{\omega}_i\}_{i=1}^N$ be a orthonormal basis of a linear subspace $\Omega \subset \Lambda$, called the **deflation subspace** and let the restriction of A to Ω, $\widetilde{A} := A\big|_\Omega$ called the **little operator**, be invertible. Define the action of P_L on an arbitrary vector $\vec{x} \in \Lambda$ as*

$$P_L\vec{x} := \vec{x} - \sum_{i,j=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j, \vec{x}\rangle$$

*and let $\vec{x}_1^\star$ be one of the solutions to the **deflated system** $\hat{A}\vec{x}_1 = P_L\vec{b}$, where $\hat{A} := P_L A$ is called the **deflated operator**. Consider*

$$\vec{x}^\star := P_R\vec{x}_1^\star + \sum_{i,j=1}^{N} \vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j, \vec{b}\rangle, \tag{10.2}$$

with P_R satifying $P_L A = AP_R$. Then $\vec{x}^\star$ is the unique solution to $A\vec{x} = \vec{b}$.

Proof. Lets first show that $P_L^2 = P_L$ is a projector,

$$P_L^2\vec{x} = P_L\left(\vec{x} - \sum_{i,j=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j, \vec{x}\rangle\right)$$

[21] Let P be a linear projector (not the identity-operator) and A a invertible linear operator. We want to solve $PA\vec{x} = P\vec{b}$. There exists at least one solution to this, namely the unique solution to $A\vec{x} = \vec{b}$. Since PA is not invertible, the only two possibilities are zero or infinite solutions and it can't be zero solutions.

$$
\begin{aligned}
&= \vec{x} - 2\sum_{i,j=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j,\vec{x}\rangle + \sum_{i,j=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\sum_{k,l=1}^{N}\langle\vec{\omega}_j,A\vec{\omega}_k\rangle(\widetilde{A}^{-1})_{kl}\langle\vec{\omega}_l,\vec{x}\rangle \\
&= \vec{x} - 2\sum_{i,j=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j,\vec{x}\rangle + \sum_{i,j,l=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_l,\vec{x}\rangle \underbrace{\sum_{k=1}^{N}\underbrace{\langle\vec{\omega}_j,A\vec{\omega}_k\rangle}_{=\widetilde{A}_{jk}}(\widetilde{A}^{-1})_{kl}}_{=\delta_{jl}} \\
&= \vec{x} - 2\sum_{i,j=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j,\vec{x}\rangle + \sum_{i,j=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j,\vec{x}\rangle \\
&= \vec{x} - \sum_{i,j=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j,\vec{x}\rangle \\
&= P_L\vec{x}.
\end{aligned}
$$

Now lets show that the second term in equation (10.2) is equal to $(1-P_R)\vec{x}_2^\star$ where $\vec{x}_2^\star$ solves $(1-P_L)A\vec{x}_2 = (1-P_L)\vec{b}$.

$$
\begin{aligned}
(1-P_R)\vec{x}_2^\star &= A^{-1}(1-P_L)A\vec{x}^\star \\
&= A^{-1}(1-P_L)\vec{b} \\
&= A^{-1}\sum_{i,j=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j,\vec{b}\rangle \\
&= \sum_{i,j=1}^{N} \vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j,\vec{b}\rangle
\end{aligned}
$$

which corresponds to the second term of $\vec{x}^\star$ in equation (10.2). Therefore by application of theorem 10.1, $\vec{x}^\star$ is the unique solution to $A\vec{x} = \vec{b}$.

□

Remark. Using the definition of P_L from corollary 10.2, the action of P_R on a arbirary vector $\vec{x}$ can be determined as

$$
\begin{aligned}
P_R\vec{x} &= A^{-1}P_LA\vec{x} \\
&= \vec{x} - \sum_{i,j=1}^{N} \vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j,A\vec{x}\rangle.
\end{aligned}
$$

Remark. An application of P_L to an arbitrary vector $\vec{x}$ involves solving the ***little equation*** $\widetilde{A}\vec{\beta} = \vec{\alpha}$ on Ω for a given $\vec{\alpha}\in\Omega$. To see this, lets look at the k-th component of $P_L\vec{x}$

$$
(P_L\vec{x})_k \coloneqq x_k - \sum_{i,j=1}^{N}(A\vec{\omega}_i)_k(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j,\vec{x}\rangle.
$$

Define the vector

$$
\vec{\alpha}_{\vec{x}} \coloneqq \begin{pmatrix}\langle\vec{\omega}_1,\vec{x}\rangle\\ \langle\vec{\omega}_2,\vec{x}\rangle\\ \vdots\\ \langle\vec{\omega}_N,\vec{x}\rangle\end{pmatrix}.
$$

Then

$$(P_L\vec{x})_k = x_k - \sum_{i=1}^{N}(A\vec{\omega}_i)_k(\widetilde{A}^{-1}\vec{\alpha}_{\vec{x}})_i.$$

By similar analysis, an application of P_R has the same cost with one additional application of A. Also the vectors $\{A\vec{\omega}_i\}_{i=1}^N$ and $\{\vec{\omega}_i\}_{i=1}^N$ have to be kept in system memory.

Remark. Assuming that the condition number of A is high and the ***spectrum*** of A, $\sigma(A)$, is separable in a way such that

$$\sigma(A) = \sigma_l(A) \cup \sigma_h(A) \qquad \text{with} \qquad \max_{\lambda\in\sigma_l(A)}|\lambda| \ll \min_{\lambda\in\sigma_h(A)}|\lambda|. \tag{10.3}$$

The subscripts stand for "low" and "high", corresponding to the low and high modes of the operator A. So, the property in equation (10.3) states that the bulk of the low and high eigenvalues are somehow clustered in two regions. Consider the linear subspaces $\Omega_l, \Omega_h \subset \Lambda$ such that the low and high eigenvectors corresponding to the low and high eingenvalues of A are contained in Ω_l and Ω_h respectively. Then the condition number of A restricted to the low (high) modes is much smaller than the condition number of A. Therefore, if we are able to find a orthonormal basis $\{\vec{\omega}_i\}_{i=0}^N$ of the subspace Ω_l containing the bulk of the low eigenmodes of A, we can apply deflation from corollary 10.2 to solve the little equation that has a significantly smaller condition number than A. Then solve the deflated system and using this solution construct a solution of the full system.

Lemma 10.3. *Let A, $\{\vec{\omega}_i\}_{i=1}^N$, Ω, P_L, P_R be as in corollary 10.2 and assume that the spectrum of A is separable (10.3). Define the deflation subspace to be the subspace corresponding to the low eigenmodes, $\Omega := \Omega_l$. Then $\kappa(\hat{A}) \ll \kappa(A)$*

Proof. Lets define the orthogonal projector $P^\perp$ to $\Omega^\perp$, the othogonal complement of the deflation subspace of Ω,

$$P^\perp\vec{x} := \vec{x} - \sum_{i=1}^{N}\langle\vec{\omega}_i,\vec{x}\rangle\vec{\omega}_i.$$

The deflated operator $\hat{A} := P_L A$ acts on the orthogonal complement,

$$\begin{aligned}
\hat{A}(1-P)\vec{x} &= P_L A(1-P)\vec{x} \\
&= P_L A\vec{x} - \sum_{k=1}^{N} P_L A\vec{\omega}_k\langle\vec{\omega}_k,\vec{x}\rangle \\
&= A\vec{x} - \sum_{i,j=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j, A\vec{x}\rangle - \sum_{k=1}^{N} A\vec{\omega}_k\langle\vec{\omega}_k,\vec{x}\rangle + \sum_{i,k=1}^{N} A\vec{\omega}_i \underbrace{\sum_{j=1}^{N}(\widetilde{A}^{-1})_{ij}\underbrace{\langle\vec{\omega}_j, A\vec{\omega}_k\rangle}_{=\widetilde{A}_{jk}}}_{\delta_{ik}}\langle\vec{\omega}_k,\vec{x}\rangle \\
&= A\vec{x} - \sum_{i,j=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j, A\vec{x}\rangle - \sum_{k=1}^{N} A\vec{\omega}_k\langle\vec{\omega}_k,\vec{x}\rangle + \sum_{k=1}^{N} A\vec{\omega}_k\langle\vec{\omega}_k,\vec{x}\rangle \\
&= A\vec{x} - \sum_{i,j=1}^{N} A\vec{\omega}_i(\widetilde{A}^{-1})_{ij}\langle\vec{\omega}_j, A\vec{x}\rangle \\
&= P_L A\vec{x} \\
&= \hat{A}\vec{x}.
\end{aligned}$$

Define the ***minimal and maximal eigenvalue*** of A,

$$\lambda_{min}(A) := \min_{\lambda \in \sigma(A)} |\lambda| \qquad \text{and} \qquad \lambda_{max}(A) := \max_{\lambda \in \sigma(A)} |\lambda|.$$

The condition number of $\hat{A}$ can now be upper bounded,

$$\kappa(\hat{A}) = \frac{\left|\lambda_{max}(\hat{A})\right|}{\left|\lambda_{min}(\hat{A})\right|} \ll \frac{|\lambda_{max}(A)|}{\left|\lambda_{min}(\hat{A})\right|} \leq \frac{|\lambda_{max}(A)|}{|\lambda_{min}(A)|} = \kappa(A),$$

where property (10.3) as used in the first inequality.

□

Remark. Lemma 10.3 tells us that the deflated system is significantly better conditioned than the full system and is therefore solved in fewer iterations.

Lemma 10.4. *P_L as defined in corollary 10.2 is a projection to the orthogonal complement of Ω, i.e. $\langle \vec{\omega}_k, P_L \vec{x} \rangle = 0$.*

Proof. Let $\vec{x}$ be an arbitrary vector, and $k \in \{1, \ldots, N\}$, then

$$\begin{aligned}
\langle \vec{\omega}_k, P_L \vec{x} \rangle &= \langle \vec{\omega}_k, \vec{x} \rangle - \sum_{i,j=1}^{N} \langle \vec{\omega}_k, A \vec{\omega}_i \rangle (\widetilde{A}^{-1})_{ij} \langle \vec{\omega}_j, \vec{x} \rangle \\
&= \langle \vec{\omega}_k, \vec{x} \rangle - \sum_{j=1}^{N} \langle \vec{\omega}_j, \vec{x} \rangle \sum_{i=1}^{N} \widetilde{A}_{ki} (\widetilde{A}^{-1})_{ij} \\
&= \langle \vec{\omega}_k, \vec{x} \rangle - \sum_{j=1}^{N} \langle \vec{\omega}_j, \vec{x} \rangle \delta_{kj} \\
&= 0.
\end{aligned}$$

□

TODO

11 Multishift Conjugate Gradient algorithm

TODO

Proposal 11.1: MSCG in mixed precision

TODO: Multishift Conjugate Gradient in mixed precision. Currently only in binary64.

12 Dirac operator

TODO

Definition 12.1 (Hadamard product)**.** *The* ***Hadamard product*** *of two vectors $\vec{x}$ and $\vec{y}$ is defined as*

$$\begin{aligned}
\odot \colon \mathbb{R}^n \times \mathbb{R}^n &\to \mathbb{R}^n \\
(\vec{x}, \vec{y}) &\mapsto \vec{x} \odot \vec{y},
\end{aligned}$$

with

$$(\vec{x} \odot \vec{y})_i := (\vec{x})_i (\vec{y})_i,$$

where $(\vec{v})_i$ denotes the i-th component of the vector $\vec{v}$.

13 Summary

TODO

14 Future

15 References

[1] T. Banks and A. Casher. Chiral symmetry breaking in confining theories. *Nuclear Physics B*, 169(1-2):103–125, 1980.

[2] I. Buck. Taking the plunge into gpu computing. *GPU Gems*, 2:509–519, 2005.

[3] I. Campos, P. Fritzsch, M. Hansen, M. Krstić Marinković, A. Patella, A. Ramos, and N. Tantalo. openq*d. https://gitlab.com/rcstar/openQxD, 2018. Accessed: 2021-01-06.

[4] W. Cody. Towards sensible floating-point arithmetic. Technical report, Argonne National Lab., IL (USA), 1980.

[5] P. W. Group et al. Posit standard documentation - release 3.2-draft. *Posit Standard Documentation*, 2018.

[6] J. L. Gustafson. Posit arithmetic. *Mathematica Notebook describing the posit number system*, 30, 2017.

[7] J. L. Gustafson and I. T. Yonemoto. Beating floating point at its own game: Posit arithmetic. *Supercomputing Frontiers and Innovations*, 4(2):71–86, 2017.

[8] R. Krashinsky, O. Giroux, S. Jones, N. Stam, and S. Ramaswamy. Nvidia ampere architecture in-depth. *NVIDIA blog: https://devblogs. nvidia. com/nvidia-ampere-architecture-in-depth*, 2020.

[9] M. Lüscher. Solution of the dirac equation in lattice qcd using a domain decomposition method. *Computer physics communications*, 156(3):209–220, 2004.

[10] M. Lüscher. Local coherence and deflation of the low quark modes in lattice qcd. *Journal of High Energy Physics*, 2007(07):081, 2007.

[11] I. of Electrical, E. E. C. S. S. Committee, and D. Stevenson. *IEEE standard for binary floating-point arithmetic*. IEEE, 1985.

[12] I. of Electrical, E. E. C. S. S. Committee, and D. Stevenson. *IEEE standard for binary floating-point arithmetic*. IEEE, 2008.

[13] R. G. T. REMOVE. Todo remove github repository: Source code of the implementation. http://github.com/chaoos/TODO, 2021. Accessed: 2021-01-01.

[14] J. R. Shewchuk et al. An introduction to the conjugate gradient method without the agonizing pain, 1994.

[15] S. Wang and P. Kanwar. Bfloat16: the secret to high performance on cloud tpus. *Google Cloud Blog*, 2019.

Appendices

A Code

All code used in this report is open source and can be found in the GitHub repository [13]

Acronyms

BLAS Basic Linear Algebra Subprograms. 10

CGNE Conjugate Gradient on the normal equations. 10

FLOPS Floating Point Operations Per Second. 24

GCR Generalized Conjugate Residual. 27, 32–34, 36

MPI Message Passing Interface. 44

MR Minimal Residual. 31, 32

MSCG Multishift Conjugate Gradient. 40

NaN not a number. 12–14, 21, 24

POPS Posit Operations Per Second. 24

QCD Quantum chromodynamics. 2

SAP Schwarz Alternating Procedure. 27, 31, 32, 35

SF Schrödinger functional. 19

Glossary

bfloat16 Googles Brain float [15] floating point number representation with encoding in length of 16 bits. 12, 13, 16, 19, 21–24, 26, 27

binary16 IEEE754 2008 [12] conformant floating point number representation with encoding in length of 16 bits. 12, 13, 15, 16, 19–24, 26, 27

binary32 IEEE754 2008 [12] conformant floating point number representation with encoding in length of 32 bits. 10, 12, 13, 16–19, 21–24, 26

binary64 IEEE754 2008 [12] conformant floating point number representation with encoding in length of 64 bits. 10, 12, 13, 18–26, 36

fused multiply–add A multiply-add operation $a + bc$ in one shot, where the rounding is deferred. 15

posit16 Posit Standard [5] conformant storage format for real number representation with encoding in length of 16 bits and an exponent size of `es=1`. 15, 16, 19–24

posit32 Posit Standard [5] conformant storage format for real number representation with encoding in length of 32 bits and an exponent size of `es=2`. 15, 16, 19, 21–24

posit64 Posit Standard [5] conformant storage format for real number representation with encoding in length of 64 bits and an exponent size of `es=3`. 15

posit8 Posit Standard [5] conformant storage format for real number representation with encoding in length of 8 bits and an exponent size of `es=0`. 15, 19, 21–23

quire Posit Standard [5] conformant special fixed-size data type that can be thought of as a dedicated register that permits dot products, sums, and other operations to be performed with rounding error deferred to the very end of the calculation [6]. 15, 20, 22–24

rank In MPI a process is identified by its rank, with is an integer between $[0, N-1]$, where N is the size of the MPI process group. 3

sparse matrix A matrix, where most of the entries are 0. 3

tensorfloat32 Nvidias TensorFloat-32 [8] floating point number representation with encoding in length of 32 bits, but only 19 bits are used. 12, 13, 16, 19, 21–24, 27