Orthogonalization variants in the Arnoldi Algorithm

Henri Willems

March 31, 2025

In my Master thesis, I investigated restarted Krylov subspace methods for the evaluation of matrix functions multiplied with a vector. I had written an implementation of this using Python and Numpy, which worked reasonably well on my laptop. But I was wondering how difficult it would be to optimize this calculation. In particular, I was thinking about a JAX implementation with its just-in-time compilation using the XLA compiler. However, after some short experimentation, I realized that I would have to solve some tricky issues with early stopping for this iterative method.

After some time, I came across the repository `github.com/pnkraemer/matfree` which contains clean and well-documented JAX code for matrix-free linear algebra methods. It also has an implementation of a jit-able and autodiff-able (!) Arnoldi method for the evaluation of matrix functions as described in the paper [3]. Looking into the code in detail, I noticed something peculiar with the implementation of the Arnoldi algorithm for constructing the Arnoldi decomposition of the input matrix. Looking, for example, at the Wikipedia entry for the Arnoldi algorithm we see that the orthogonal matrix of Krylov space basis vectors is usually constructed by the modified Gram-Schmidt orthogonalization process. So I was surprised to see that the basis in `matfree` was not using this algorithm. At first glance, I did not even recognize the classical Gram-Schmidt algorithm, since there was no explicit iteration in the code.

Here I want to present some of the things I found out about different formulations of the Gram-Schmidt process. I will briefly describe the reason for using the classical Gram-Schmidt algorithm in HPC implementations of the Arnoldi method. Then I will show some numerical experiments to illustrate the impact on run-time and stability of the Arnoldi algorithm.

1 Arnoldi Algorithm

In the Arnoldi algorithm for a matrix $A \in \mathbb{R}^{n\times n}$ and a vector $b \in \mathbb{R}^n$, we require in each step m an orthogonal basis $Q \in \mathbb{R}^{n\times m}$ of the Krylov subspace $\mathcal{K}_m(A, b)$ and a Hessenberg matrix $H \in \mathbb{R}^{m\times m}$ such that

$$AQ = QH + h_{m+1,m} q_{m+1} e_m^T,$$

where $Q = [q_1, \ldots, q_m]$ and $H = [h_{ij}]$. The usual way to construct the basis Q is by the modified Gram-Schmidt orthogonalization process (MGS).

Algorithm 1 Modified Gram-Schmidt

$q_1 = b/\|b\|_2$
for $j = 1, \ldots, m-1$ **do**
    $w = Aq_j$
    for $i = 1, \ldots, j$ **do**
        $h_{i,j} = \langle q_i, w\rangle$
        $w = w - h_{ij} q_i$
    end for
    $h_{j+1,j} = \|w\|_2$
    $q_{j+1} = w/h_{j+1,j}$
end for

This algorithm is usually advised to be used since it is more stable than the classical Gram-Schmidt algorithm.

Algorithm 2 Classical Gram-Schmidt

$q_1 = b/\|b\|_2$
for $j = 1, \ldots, m-1$ **do**
    $w = Aq_j - \sum_{i=1}^{j} h_{i,j} q_i$, where $h_{i,j} = \langle q_i, w\rangle$
    $h_{j+1,j} = \|w\|_2$
    $q_{j+1} = w/h_{j+1,j}$
end for

The MGS continuously updates the vector w in the inner i loop to mitigate rounding errors. However, this forces the implementation of MGS to be sequential. In contrast, the CGS algorithm can be implemented with

matrix-vector products by replacing line 3 by

$$\begin{aligned} w &= Aq_j \\ h_{1:j+1,j} &= Q^T w \\ w &= w - Qh_{1:j+1,j}. \end{aligned}$$

It's well known that if you want to speed up computations in Python, you should avoid for loops and use Numpy/ Scipy functions instead. This is because those libraries can call out to BLAS kernels which are implementations optimized for your specific hardware, for example, on my laptop Numpy uses OpenBLAS 0.3.28. Thus, we'd expect CGS with matrix-vector products to be much faster than MGS and this is indeed the case. Looking ahead at the numerical example, generating the first 50 Arnoldi vectors takes three times as long with MGS compared to CGS.

An alternative idea for speeding up MGS is to simply allow for loss of orthogonality. Restricting the size of the i loop is known as the Incomplete Orthogonalization Process (IOP).

Of course, it's not much use to speed up a computation if the result is wrong. The CGS algorithm is known to lose orthogonality of the basis vectors and is "numerically unstable" as the Wikipedia entry states. A simple way to prevent loss of orthogonality in the CGS algorithm is to reorthogonalize the basis vectors. In our pseudocode for CGS, this would mean assigning $w = Aq_j$ and then repeating line 3,

$$w = w - -\sum_{i=1}^{j} h_{i,j} q_i, \text{ where } h_{i,j} = \langle q_i, w \rangle.$$

A result by [1] states that one reorthogonalization repetition is sufficient to ensure orthogonality of the basis vectors to numerical precision, if A is numerically non-singular. For example JAX (version 0.5.3) uses this "twice is enough" approach in its implementation of the GMRES algorithm. This doubles the operation count compared to regular CGS and MGS, but as we will see later in practice, it is still much faster than MGS.

Another version of the Gram-Schmidt process steps back to the original goal of orthogonalizing w against the Krylov space $K_m(A, v)$. In other words, this means that w should be contained in the complement of the Krylov space. This can be achieved by projecting onto that space with a matrix P such that

$$P = I_m - Q_m (Q_m^T Q_m)^{-1} Q_m^T.$$

The `matfree` Python implementation of the Arnoldi algorithm gives the option to use

$$w = w - Q@Q.T@w,$$

as a reorthogonalization step in the CGS algorithm, where $(Q_m^T Q_m)^{-1}$ is collapsed to the identity in the projection matrix.

More elaborate projection-based reorthogonalizations can be found in the context of massively parallel computing, e.g., in [4]. In this setting, many processors are supposed to be used in parallel to speed up large computations, for example, on an HPC. One of the main factors is minimizing the amount of times the processors need to share information with each other, such as when calculating a scalar from a matrix-vector product involving a vector that is split across many processors. Thus, one might try to group all matrix-vector products to be executed near each other in these so-called communication avoiding methods.

2 Numerical Example

For demonstrating the behavior of the different orthogonalization procedures, we use a structured sparse *Grcar* matrix and a vector with elements drawn from the normal distribution. The matrix has a Hessenberg structure with -1 on the subdiagonal and 1 on the diagonal and the first three superdiagonals. In the Numerical Linear Algebra lecture by Prof. Liesen at TU Berlin, this matrix is used to show that CGR and MGR might lose orthogonality also for well-conditioned matrices. We conduct the experiment using Numpy and Scipy on a laptop CPU for a large matrix size of $n = 5,000$. The code can be found at `github.com/marthakn/ArnoldiOrthogonalization`.

First, we look at the loss of orthogonality of the Arnoldi basis vectors depending on the orthogonalization procedure. This is measured by $\|I_m - Q_m^T Q_m\|_F$ which would ideally be zero.

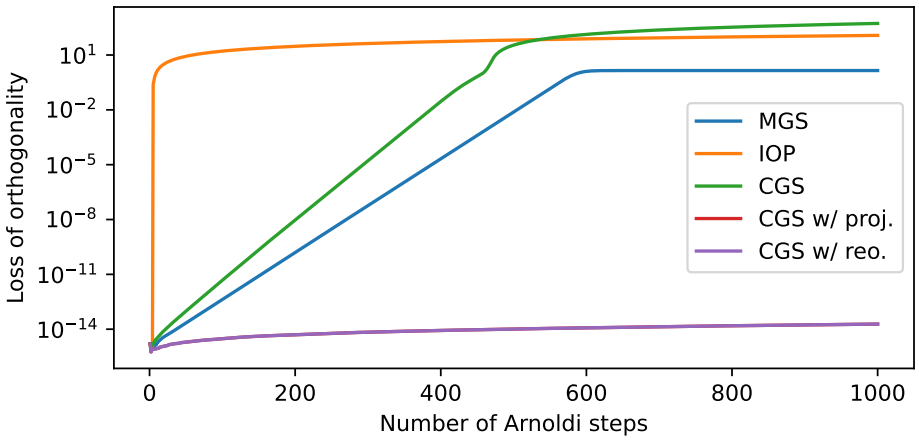

Figure 1: Loss of orthogonality $\|I_m - Q_m^T Q_m\|_F$ of the Arnoldi basis vectors.

For IOP, we choose to only orthogonalize against the two preceding vectors. Thus, the runtime of IOP tells us how long the Lanczos algorithm would take if we used it on a similarly sized symmetric matrix. However, for this matrix with this setting, IOP immediately loses orthogonality. The Figure also shows that MGS is more stable than CGS, but the loss of orthogonality is only delayed by a few iterations. However, reorthogonalization is very effective in maintaining the orthogonality of the basis vectors. Noteably, it doesn't make a difference here whether we repeat CGS (CGS w/ reo.) or use the projection-based reorthogonalization of `matfree` (CGS w/ proj.).

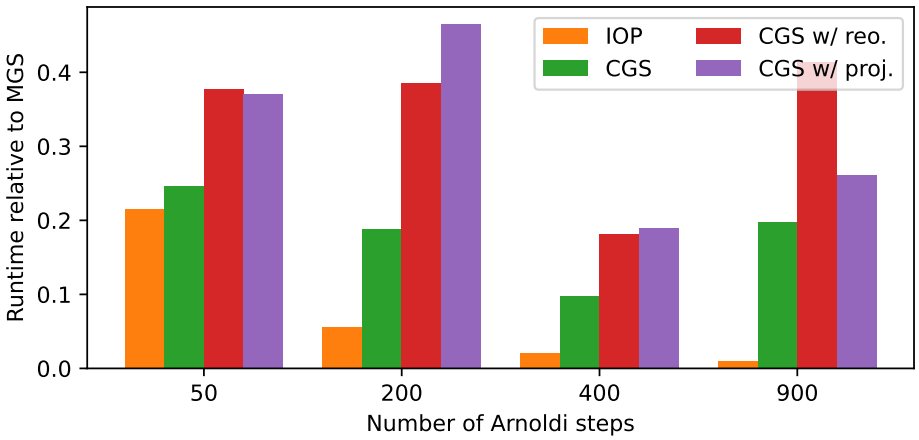

Figure 2: Runtime generating m Arnoldi vectors. Plotted is the time relative to what MGS takes.

Next, we have a look at the runtime differences. The Figure above plots the relative runtime of generating m Arnoldi compared to using MGS. The runtimes are chosen as the median of five tries. Of course, a Numpy/ Scipy implementation will not have production level performance, but with the fast computational BLAS kernels, it can point us towards the performance of the methods when hardware accelerators are available. As we expect, MGS is consistently the slowest method. Ignoring the inaccurate IOP, the fastest method is CGS but the really stable reorthogonalized versions are not much slower. They both perform two additional matrix-vector products per iteration, so we would expect them to have very comparable runtimes. Therefore, it is somewhat surprising that they differ this much at all.

As a last variation, we could try to see if one of these methods is more suitable for usage with a hardware accelerator. For this, we use a free Google Colab notebook with a v4 TPU and reimplement the methods from above in JAX. The JAX implementation is based on the code from `github.com/pnkraemer/matfree`. Then we use the JAX JIT compiler to automatically take advantage of the TPU. This is just a proof-of-concept and not meant to be truly representative of properly optimized performance, but we see the results in the table with minimal effort. Using the accelerator

Table 1: Orthogonalization methods for 900 Arnoldi vectors. Runtime is in seconds using TPU, loss of orthogonality is $\|I_m - Q_m^T Q_m\|_F$, speedup is median $t_{\text{CPU}}/t_{\text{TPU}}$ of five runs.

Variant	Runtime	Loss of orthogonality	Speedup
MGS	11.2	1.4	3.8
CGS	0.4	403.7	193
CGS w/ proj.	0.7	2×10^{-14}	94

with activated double precision in JAX does not change the stability of the result, as expected. However, we can mainly see that the reorthogonalization approach is much more able to take advantage of the TPU than the MGS approach.

3 Further reading

More thorough comparisons of orthogonalization procedures can be found in [2], [4].

References

[1] L. Giraud, J. Langou, M. Rozložník, and J. van den Eshof, "Rounding error analysis of the classical Gram-Schmidt orthogonalization process," *Numerische Mathematik*, vol. 101, no. 1, pp. 87–100, Jul. 1, 2005. DOI: 10.1007/s00211-005-0615-4.

[2] V. Hernández, J. E. Román, A. Tomás, and V. Vidal, "SLEPc Technical Report STR-1: Orthogonalization routines in SLEPc," SLEPc Technical Report, Jun. 2007. [Online]. Available: https://slepc.upv.es/documentation/reports/str1.pdf.

[3] N. Krämer, P. Moreno-Muñoz, H. Roy, and S. Hauberg, "Gradients of functions of large matrices," *Advances in Neural Information Processing Systems*, vol. 37, pp. 49 484–49 518, Dec. 16, 2024. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2024/hash/58b286aea34a91a3d33e58af0586fa40-Abstract-Conference.html.

[4] T. Tafolla, S. Gaudreault, and M. Tokman. "Low-synchronization Arnoldi methods for the matrix exponential with application to exponential integrators." arXiv: 2410.14917 [math], pre-published.