EFFICIENT PROJECTION SPACE UPDATES FOR THE APPROXIMATION OF ITERATIVE SOLUTIONS TO LINEAR SYSTEMS WITH SUCCESSIVE RIGHT HAND SIDES

BY

NICHOLAS CHRISTENSEN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
with a concentration in Computational Science and Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Professor Paul Fischer

ABSTRACT

Accurate initial guesses to the solution can dramatically speed convergence of iterative solvers. In the case of successive right-hand sides, it has been shown that accurate initial solutions may be obtained by projecting the newest right hand side vector onto a column space of recent prior solutions. We propose a technique to efficiently update the column space of prior solutions. We find this technique can modestly improve solver performance, though its potential is likely limited by the problem step size and the accuracy of the solver.

ACKNOWLEDGMENTS

I offer sincere gratitude to Professor Paul Fischer for his invaluable insights and advice. Under his tutelage and direction I have become a much better mathematician, programmer, and scientist. Without him this thesis would not have been possible.

I also thank members of the Spectral Element Analysis Lab for their encouragement and assistance, particularly in helping me understand Nek5000's finer details.

Finally, I thank James Lottes, a note from whom served as the kernel for this thesis.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Background

Iterative methods are among the most powerful means of solving linear systems, particularly when a linear system is large, sparse, and non-diagonalizable. Such systems may occur, for example, when solving partial differential equations (PDEs) numerically, an important task for many in science, engineering, and applied mathematics.

Iterative methods evolve an initial guess to the solution towards the numerical solution, stopping after reaching some desired accuracy. The number of iterations required for an iterative method to converge to the solution significantly decreases if the initial guess is already close (in a minimized residual sense) to the numerical solution. In the case of solving (implicitly) smoothly time-evolving PDEs - towards which we orient the following discussion - such an initial guess is available.

Time-evolving PDEs are often modeled by solving the differential equation at discrete points in time, separated by a time interval Δt. Over very short periods of time the system changes very little. Consequently, the solution at some time t resembles closely the solution at a nearby previous time $t - \Delta t$. If the solution at this previous time is used as the initial guess for the solution at time t, then the iterative solver needs only to calculate the difference between the solution at the previous time step and the solution at the current time. We not restricted to only solving for the change in the solution between timesteps, however. As prior research has borne out,[1] we can save the solution at multiple previous time steps and generate an initial guess to the solution using a linear combination (i. e. weighted sum) of these solutions to obtain a better approximation.[1]

[1]While we approach this problem in the context of numerically solving partial differential equations, this technique is applicable to any smoothly evolving problem with successive right hand sides.[1]

For accuracy and stability, it is better to generate the initial guess from a set of basis vectors orthogonal in the A inner product than to use the prior solutions directly. This presents a potential problem. As the system evolves in time, new solutions are generated necessitating updating the A-orthogonal basis. The cost of doing this using a Gram-Schmidt procedure grows quadratically with the number of saved solutions. Here we present a method that uses Givens rotations or Householder reflections in addition to a Gram-Schmidt procedure to accomplish the same task with a cost that scales linearly with the number columns. We find, however, that the performance benefits are modest and Gram-Schmidt algorithms may provide adequate performance.

1.2 Prior work

1.2.1 Projection onto prior solutions

Though most effort into improving iterative solvers has focused on developing lighter and more rapidly converging preconditioners, several methods of improving initial solution approximations have emerged.[2] The method sketched above and expounded on slightly more in the next chapter was introduced by Fischer.[1,3] It has continued to find applications[4–8] which in part motivates our efforts here. Löhner obtained good results using a simplified version of Fischer's approach that simply projects onto the column space of a few prior solutions without orthogonalization.[9] The number of usable prior solutions is presumably limited by ill-conditioning, but performance comparison may be warranted in future work.

Other approximation methods are known. One related set of methods forms the solution approximation by projecting onto a (potentially augmented) set of orthogonal Krylov bases. We refer to Saad and to Chan and Wan for overview of these methods, although this area of research has advanced considerably since their analyses were published.[10,11] Another active area of research examines use of Proper Orthogonal Decomposition for generating accurate initial guesses.[2,12–14] In other work, Grinberg and Karniadakis studied a high-order spectral extrapolation method[2] and Sayed and and Sadkane applied the Petrov-Galerkin method to obtain a solution approximation.[15]

Techniques have been developed for the case of a changing A matrix in addition to a changing right hand side. We do not consider this case here and instead content ourselves that for a slowly changing matrix, the oldest solution may be removed from the projection space before the cumulative deviation becomes significant.[2] Work by de Sturler, Parks, and Kilmer on Krylov subspace recycling pertains to this problem.[16,17]

1.2.2 Efficient QR updates

Efficient ways of updating a QR factorization have been studied since the 1970s. Gill, Golub, Murray, and Saunders examined updates for the full orthogonal QR factorization.[18] Daniel, Gragg, Kaufman, and Stewart (DGKS) later extended these techniques to the reduced orthogonal QR factorization.[19] Implementation of DGKS's algorithms were notably absent from LINPACK,[20] but a Fortran implementation was eventually published by Buckley in 1981.[21,22] Reichel and Gragg subsequently improved on DGKS's algorithm in a later Fortran implementation.[23] More recently, Hammarling, Higham, and Lucas developed algorithms for certain block updates to QR factorizations and generalized the updating algorithms to handle the underdetermined case.[24,25] Updating algorithms are now frequently covered in numerical methods textbooks[26–29] and libraries for the task of updating orthogonal QR factorizations are available in several programming languages.[30–33]

The oblique QR factorization, of interest for the current problem, has received less attention. Afriat wrote on the topic of oblique projections in 1957[34] and in subsequent decades strong use for oblique projections was found in signal processing.[35,36] Work on general oblique QR factorization traces to Thomas in the 1990s.[37] More recently, excellent work has been done on stability and error analysis of oblique QR factorization algorithms.[38–41] Pioneering work by Rebollo-Neira[35] and Stewart[38] on updating oblique QR factorizations using the Gram-Schmidt procedure is relevant to our discussion here.

[2] But see also the discussion in section 2.1.1.

1.3 Outline

The remainder of this work is organized as follows. Section 1 of chapter 2 presents the more straight-forward approach to updating the projection space using Gram-Schmidt orthogonalization. Section 2 of the chapter explores the mathematical theory of the efficient update method accompanied by several algorithms for implementations based on Givens rotations and Householder reflections. Chapter 3 presents the results of numerical experiments with the algorithms in Nek5000, a computational fluid dynamics program. We close in chapter 4 with some final analysis and discussion.

CHAPTER 2

SOLUTION APPROXIMATION BY PROJECTION

2.1 A-orthogonal basis

Let A be an $n \times n$ symmetric positive definite matrix and consider the system of equations $A\mathbf{x}^i = \mathbf{b}^i$ where the superscript denotes the solution vector and right hand side vectors at the ith discrete time step.[1] In this system, $\mathbf{b}^i$ is a vector of knowns and $\mathbf{x}^i$ is a vector of unknowns for which the system is solved.

Suppose we solve $A\mathbf{x}^i = \mathbf{b}^i$ using an iterative method (e. g. CG, GMRES). The iterative solver initiates with a guess to the solution vector $\widehat{\mathbf{x}}^i$. If the system is well-conditioned and the iterative method is stable, then after each round of the iterative method, $\widehat{\mathbf{x}}^i$ is updated with the goal of making the norm of the residual, $\|A\widehat{\mathbf{x}}^i - \mathbf{b}^i\|_2$, smaller than at the previous iteration. Over the course of iteration $\widehat{\mathbf{x}}^i$ converges toward $\mathbf{x}^i$, the numerical solution to the system.[42]

We cease applying the iterative method after the norm of the residual is reduced below some pre-determined tolerance. The number of iterations needed to decrease the residual below this point depends on the convergence rate of the iterative method and the initial residual. The latter is the focus of the discussion here. To reduce the number of iterations to convergence, we desire an initial $\widehat{\mathbf{x}}^i$ close to the solution $\mathbf{x}^i$ so $\|A\widehat{\mathbf{x}}^i - \mathbf{b}^i\|_2$ is small compared to $\|\mathbf{b}^i\|_2$.

If the system changes little between time steps, then the solution vector for the current time step, $\mathbf{x}^i$, resembles the solution vector at a recent time step such as $\mathbf{x}^{i-1}$. With a sufficiently small step size, the *amount of change* itself also likely changes little between time steps. Using an additional recent data point, say the solution $\mathbf{x}^{i-2}$, we can obtain information about

[1] Partial differential equations are frequently formulated as linear systems with symmetric positive definite matrices and we restrict our treatment here to these systems.

the expected change in the solution and adjust our solution approximation accordingly. As the change in the change in the solution and so forth may differ little between time steps, we may persist with this scheme until the higher order changes are no longer smooth. In this fashion, we may use the solutions at several previous time steps to make higher order improvements to the initial guess. Hence, a reasonable initial $\widehat{\mathbf{x}}^i$ may be constructed from some linear combination of the solution vectors of previous time steps.

$$\widehat{\mathbf{x}}^i = \sum_{j=1}^{k} w_j \mathbf{x}^{i-j}. \tag{2.1}$$

If we maintain a set of the solution vectors and right hand side vectors at the previous k time steps,[2] then at the ith time step we have a system $AX = B$ where, for example, $X = [\mathbf{x}^{i-k}\ \mathbf{x}^{i-k+1}\ \ldots\ \mathbf{x}^{i-2}\ \mathbf{x}^{i-1}]$ and $B = [\mathbf{b}^{i-k}\ \mathbf{b}^{i-k+1}\ \ldots\ \mathbf{b}^{i-2}\ \mathbf{b}^{i-1}]$.[3] The above expression for $\widehat{\mathbf{x}}^i$ can then be restated in matrix form as

$$\widehat{\mathbf{x}}^i = X\mathbf{w} \tag{2.2}$$

where $\mathbf{w}$ is a $k \times 1$ vector that weights the contribution of each column of X.

The above formulation of $\widehat{\mathbf{x}}^i$ is problematic for two reasons. First, computing $\mathbf{w}$ requires solving the least squares problem

$$AX\mathbf{w} = B\mathbf{w} \approx \mathbf{b}^i \tag{2.3}$$

at each time step. This procedure scales like nk^2 procedure and it may become costly if n or k are large. Second, because computers have finite precision and because right hand side vectors of the system change little between consecutive time steps, the columns of B are likely to be nearly linearly dependent. This makes the least squares problem above quite ill-conditioned. Consequently, a direct linear combination of many previous solutions is likely not a much more accurate approximation to $\mathbf{x}^i$ than a linear combination of only a few. This

[2]While we use an unbroken sequence of vectors, this is not a requirement. Any set of successive right hand side and solution vectors suffices, though one's choice of saved vectors can affect the utility of the projection.

[3]We are not committed to this ordering of vectors and will find a different ordering can be advantageous.

limits the accuracy of the approximation.

With some modifications to the problem, both of these issues are surmountable. The former problem is the study of following sections. The latter problem is largely addressed by forming $\widehat{\mathbf{x}}^i$ as a linear combination of columns of a matrix Q_x that has the same column space as X but is orthogonal in the A inner product (i. e. if $\mathbf{q}_i$ and $\mathbf{q}_j$ are columns of Q_x then $\langle \mathbf{q}_i, A\mathbf{q}_j \rangle = \mathbf{q}_i^T A \mathbf{q}_j = \delta_{ij}$).[4] This gives us

$$\widehat{\mathbf{x}}^i = Q_x \mathbf{r} \tag{2.4}$$

where $\mathbf{r}$ is a vector of weights.

Substituting this expression for $\widehat{\mathbf{x}}^i$ into $A\widehat{\mathbf{x}}^i = \mathbf{b}^i$, we have $AQ_x\mathbf{r} = \mathbf{b}^i$. Multiplying both sides by Q_x^T, we then obtain $Q_x^T A Q_x \mathbf{r} = Q_x^T \mathbf{b}^i$. Because Q_x is orthogonal in the A inner product (A-orthogonal), $Q_x^T A Q_x = I$ and we obtain

$$\mathbf{r} = Q_x^T \mathbf{b}^i. \tag{2.5}$$

Consequently by (2.4) and (2.5),

$$\widehat{\mathbf{x}}^i = Q_x Q_x^T \mathbf{b}^i. \tag{2.6}$$

Because $Q_x Q_x^T \mathbf{b}^i = Q_x Q_x^T A \mathbf{x}^i$, we note (2.6) constitutes an oblique projection of $\mathbf{x}^i$ (the desired solution vector) onto the column space of Q_x (the space of the k prior solutions).[5] $Q_x Q_x^T$ is symmetric which implies Q_x is normal. Therefore, the conditioning of (2.6) depends only on the ratio in modulus of the maximum and minimum eigenvalues of $Q_x Q_x^T$.[6] This is an improvement over finding $\widehat{\mathbf{x}}^i$ using

$$\widehat{\mathbf{x}}^i = X\mathbf{w} = X(B^T B)^{-1} B^T \mathbf{b}^i. \tag{2.7}$$

The matrix $X(B^T B)^{-1} B^T$ has the same eigenvalues as $Q_x Q_x^T$ but is not normal so here

[4] The Dirac delta function is defined such that $\delta_{ij} = 1$ if $i = j$ and $\delta_{ij} = 0$ if $i \neq j$.
[5] $(Q_x Q_x^T A)(Q_x Q_x^T A) = Q_x (Q_x^T A Q_x) Q_x^T A = Q_x Q_x^T A$ so $Q_x Q_x^T A$ is a projector.
[6] This result follows from the singular value decomposition of $Q_x Q_x^T$.

$\left|\frac{\lambda_{max}}{\lambda_{min}}\right|$ is only a lower bound for its condition number.[7]

So far we have focused on modifying the initial guess to the solution vector to reduce the initial residual while leaving the system we are solving, $A\mathbf{x} = \mathbf{b}$, intact. However, an equivalent and occasionally useful way of reducing the initial residual is to instead set the initial guess to $\mathbf{0}$ and to solve a modified problem $A\Delta\mathbf{x}^i = \Delta\mathbf{b}^i$ where

$$\Delta\mathbf{x}^i = \mathbf{x}^i - \widehat{\mathbf{x}}^i = \mathbf{x}^i - Q_x\mathbf{r}. \tag{2.8}$$

and

$$\Delta\mathbf{b}^i = A(\Delta\mathbf{x}^i) = A\mathbf{x}^i - AQ_x\mathbf{r} = \mathbf{b}^i - Q_b\mathbf{r}. \tag{2.9}$$

Afterward, we may calculate the desired solution vector as $\mathbf{x}^i = \widehat{\mathbf{x}}^i + \Delta\mathbf{x}^i$. In the case of solving either system, we require the same Q_x (and Q_b) so our discussion here is relevant to both formulations.

2.1.1 Error of the initial guess

Following Chan and Wan,[11] we estimate the scaling of the approximation error as a function of a Δt and k. Assuming the right hand side vector $\mathbf{b}$ varies smoothly in time, then there exist extrapolation factors χ_j such that

$$\mathbf{b}^i = \sum_{j=1}^{k} \chi_j \mathbf{b}^{i-j} + O(\Delta t^k)\,\mathbf{e}, \tag{2.10}$$

[7] The proof is as follows. The condition number $\kappa(A) = \|A\|\|A^{-1}\| = \max_{\|\mathbf{x}\|=1}\|A\mathbf{x}\| \times \max_{\|\mathbf{y}\|=1}\|A^{-1}\mathbf{y}\|$. However, $\max_{\|\mathbf{x}\|=1}\|A\mathbf{x}\| \geq \|A\mathbf{x}\| = |\lambda_i|\|\mathbf{x}\| = |\lambda_i|$ and likewise $\max_{\|\mathbf{y}\|=1}\|A^{-1}\mathbf{y}\| \geq \|A^{-1}\mathbf{y}\| = \left|\frac{1}{\lambda_j}\right|\|\mathbf{y}\| = \left|\frac{1}{\lambda_j}\right|$. If we choose $\lambda_i = \lambda_{max}$ and $\lambda_j = \lambda_{min}$, then we see $\kappa(A) = \|A\|\|A^{-1}\| \geq \left|\frac{\lambda_{max}}{\lambda_{min}}\right|$.

where $\mathbf{e}$ is an error vector. From (2.6), the projected solution at the ith time step is $\widehat{\mathbf{x}}^i = Q_x Q_x^T \mathbf{b}^i$. By substituting the expression in (2.10) for $\mathbf{b}^i$,

$$\begin{aligned}
\widehat{\mathbf{x}}^i &= Q_x Q_x^T \mathbf{b}^i \\
&= \sum_{j=1}^{k} \chi_j Q_x Q_x^T \mathbf{b}^{i-j} + O(\Delta t^k)\, Q_x Q_x^T \mathbf{e} \\
&= \sum_{j=1}^{k} \chi_j \widehat{\mathbf{x}}^{i-j} + O(\Delta t^k)\, \mathbf{e}'
\end{aligned} \tag{2.11}$$

.

Equation 2.11 states we can extrapolate $\widehat{\mathbf{x}}^i$, the solution approximation at the ith timestep, from a set of approximations at prior timesteps while incurring an error that scales as $O(\Delta t^k)$. However, in *projecting* $\widehat{\mathbf{x}}^i$ we do better because we use the *actual* previous solutions. Therefore, the error in the projection approximation scales *at least* like $O(\Delta t^k)$ and potentially scales better. Note this result still holds if the matrix A also evolves smoothly in time. Our decision to largely ignore the case of changing A consequently does not severely curtail the applicability of the present discussion.

2.2 Gram-Schmidt projection space update

Equation 2.6 improves the accuracy of computing $\widehat{\mathbf{x}}^i$, but leaves us with the problem of computing Q_x at every time step. We discuss in this section the straightforward approach to computing Q_x using Gram-Schmidt orthogonalization and discuss a more efficient approach in the following section.

As its name suggests, Q_x can be generated from an (oblique) QR factorization of X. Because we require Q_x be A-orthogonal, the QR algorithms available for this are largely limited to versions of Gram-Schmidt orthogonalization. No algorithms analogous to Householder transformations or Givens rotations exist for QR factorization in an oblique inner product.[40] As the Givens and Householder algorithms depend on the property that products of orthogonal matrices are also orthogonal, direct use of these methods in the oblique

case may not be possible.[43]

Algorithms for oblique Gram-Schmidt orthogonalization are available in the literature (e. g. [19,37,41,44,45]). The Gram-Schmidt algorithms we present below are tailored to our use case. In particular, the algorithms reflect our need to simultaneously orthogonalize X and B where B already exists and does not need to be recalculated as the product of AX. We use reverse ordered iterations to reduce data movement.[8]

If at each time step we have X, the set of k previous solution vectors, and B, the set of k previous right hand side vectors, then the column-oriented classical Gram-Schmidt procedure (CGS) orthogonalizes X in the A norm (i. e. orthogonalizes X and B) using the process shown in Algorithm 2.1. The vectors $\mathbf{q}_i$ are columns of Q_x and the vectors $\mathbf{s}_i$ are columns of $Q_b = AQ_x$.

Algorithm 2.1 A-Orthogonal Classical Gram-Schmidt

```
for i = k to 1 do
    q_i ← x_i
    s_i ← b_i
    for j = i + 1 to k do
        r_ji ← (q_j^T b_i + b_j^T s_i)/2
    end for
    for j = i + 1 to k do
        q_i ← q_i − r_ji q_j
        s_i ← s_i − r_ji s_j
    end for
    r_ii ← sqrt(s_i^T q_i)
    if r_ii > tolerance then
        q_i ← q_i / r_ii
        s_i ← s_i / r_ii
    else
        remove q_i and s_i
    end if
end for
```

Calculating r_{ji} as the average of $\mathbf{q}_j^T\mathbf{b}_i$ and $\mathbf{b}_j^T\mathbf{x}_i$ may seem peculiar. It is mathematically valid because

$$\mathbf{b}_j^T\mathbf{x}_i = (A\mathbf{x}_j)^T\mathbf{x}_i = \mathbf{x}_j^T A^T\mathbf{x}_i = \mathbf{x}_j^T A\mathbf{x}_i = \mathbf{x}_j^T\mathbf{b}_i = r_{ji}. \tag{2.12}$$

[8]We assume we are not tracking the index of the oldest column. By orthogonalizing in this order we avoid shifting columns when $k < k_{max}$.

The motivation calculating r_{ji} in this way is that $\mathbf{x}$ may not be solved for exactly within the iterative method. Instead, $\mathbf{x}$ is found to some tolerance, with the effect that $A\mathbf{x} = \mathbf{b}$ is only true to within some tolerance. Consequently, neither $\mathbf{x}_j^T\mathbf{b}_i$ nor $\mathbf{b}_j^T\mathbf{x}_i$ is inherently a better way of calculating r_{ji} and the best determination we can make is to average both.

A value of r_{ii} close to zero (within some set tolerance) indicates the ith columns of X and B are nearly linearly dependent with the columns that have already been orthogonalized. The contributions of such a column to $\widehat{\mathbf{x}}$ are redundant, so we can safely remove these columns from X and B (and in fact must do so to prevent the algorithm from breaking down when dividing by r_{ii}). The removal procedure is to shift the position of columns with indices greater than i to the position of their index less one. Following this, k is updated to reflect the removal of the column. Choice of tolerance is up to the user, though a functional approach is to set it to some small number relative to $\sqrt{\mathbf{x}^T\mathbf{b}}$.

The largest costs of this algorithm is the inner loops from $j = i+1$ to k. Because i changes from k to 1, the total number of iterations in the inner loop is $0 + 1 + 2 + \ldots + k - 1 = \sum_{l=0}^{k-1} l = \frac{k(k-1)}{2}$. As the vectors have length n, the total work of the Gram-Schmidt algorithm is $O(\frac{nk^2}{2})$. The algorithm also requires storing X, B, Q_x, and Q_b for a total storage cost of $4nk$.

A known problem with CGS is the loss of significant figures in subtraction, which can destroy the orthogonality of Q_x and Q_b. This typically motivates use of the modified Gram-Schmidt procedure (MGS) instead.[46] The row oriented modified Gram-Schmidt procedure - shown in Algorithm 2.2 - does not prevent subtractive cancellation. However, it does better enforce orthogonality by orthogonalizing against the error introduced by cancellation.

The required work of MGS is similar to that of CGS. However, in distributed memory systems, MGS may be costly due to communication costs. Computing the inner products requires global communication if the vectors are distributed across multiple processes. In CGS, communication can be agglomerated into two global sum calls per outer loop iteration with a resulting $O(2k)$ communication cost. (We place the inner products in a separate for-loop for this reason.) MGS requires re-computing the inner product at each iteration of the inner loop and consequently has an $O(k^2)$ communication cost.

The loss of orthogonality of CGS is canonically fixed by applying a second round of CGS to

Algorithm 2.2 *A*-Orthogonal Modified Gram-Schmidt

for $i = k$ **to** 1 **do**
  $r_{ii} \leftarrow \sqrt{\mathbf{x}_i^T \mathbf{b}_i}$
  if $r_{ii} >$ tolerance **then**
    $\mathbf{q}_i \leftarrow \mathbf{x}_i / r_{ii}$
    $\mathbf{s}_i \leftarrow \mathbf{b}_i / r_{ii}$
    for $j = i - 1$ **to** 1 **do**
      $r_{ij} \leftarrow (\mathbf{q}_i^T \mathbf{b}_j + \mathbf{s}_i^T \mathbf{x}_j)/2$
      $\mathbf{q}_j \leftarrow \mathbf{x}_j - r_{ij}\mathbf{q}_i$
      $\mathbf{s}_j \leftarrow \mathbf{b}_j - r_{ij}\mathbf{s}_i$
    end for
  else
    remove $\mathbf{q}_i$ and $\mathbf{s}_i$
  end if
end for

the output of a first round of CGS.[19] Stewart has found this generally applies in the oblique case as well.[47] While two rounds of CGS (CGS2) doubles the computational cost, it requires only $O(4k)$ global sums. If communication is sufficiently expensive and k is sufficiently large, then CGS2 may have better performance despite this additional work. If the values r_{ji} must be saved then values in each round are summed to obtain the final r_{ji} values.

At each time step we append the latest solution and right hand side vectors to X and B respectively. The order of iteration over the columns in the Gram-Schmidt algorithms presented here ensures the contributions of the oldest solution vectors are exclusively in the first columns. The number of columns we maintain in X and B is usually constrained.[9] When this k_{max} is reached it is reasonable to discard the oldest column as it likely contributes least (or close to least) to the formation of $\widehat{\mathbf{x}}^i$.[1] This opens space in the last column, allowing us to append the newest vectors as usual.

It is possible to avoid shifting entirely by tracking the location of the oldest column and overwriting it directly with the new column. If k is large, the overall performance benefit of doing this may be small compared with the cost of the Gram-Schmidt procedure. For small k this optimization may be more useful.

[9]The constraint may be due to available system memory or - as is more often the case - the improvement to $\widehat{\mathbf{x}}$ may not offset the computational cost of an additional column.

2.2.1 Reducing storage cost

Assume we factor X into the product of Q_x and R (the latter being a $k \times k$ upper-triangular matrix of weights). We then have $X = Q_x R$. A similar decomposition can be made for B as $B = AX = (AQ_x)\, R = Q_b R$. Eliminating R from both sides leaves us with a modified system $AQ_x = Q_b$.

Because Q_x has the same column space as X and Q_b has the same column space as B, it is sufficient to store only Q_x and Q_b, using their columns in place of the columns of X and B in the Gram-Schmidt algorithms. Rather than appending to X and B, the latest solution and right hand side vectors are instead appended to Q_x and Q_b. This may be an important improvement when solving very large systems, but we will also make use of it in the following section. Algorithm 2.3 and Algorithm 2.4 show the reduced storage versions of CGS and MGS respectively.

Algorithm 2.3 Reduced Memory A-Orthogonal Classical Gram-Schmidt

1: **for** $i = k$ **to** 1 **do**
2: **for** $j = i + 1$ **to** k **do**
3: $r_{ji} \leftarrow (\mathbf{q}_j^T \mathbf{s}_i + \mathbf{s}_j^T \mathbf{q}_i)/2$
4: **end for**
5: **for** $j = i + 1$ **to** k **do**
6: $\mathbf{q}_i \leftarrow \mathbf{q}_i - r_{ji}\mathbf{q}_j$
7: $\mathbf{s}_i \leftarrow \mathbf{s}_i - r_{ji}\mathbf{s}_j$
8: **end for**
9: $r_{ii} \leftarrow \sqrt{\mathbf{s}_i^T \mathbf{q}_i}$
10: **if** $r_{ii} >$ tolerance **then**
11: $\mathbf{q}_i \leftarrow \mathbf{q}_i / r_{ii}$
12: $\mathbf{s}_i \leftarrow \mathbf{s}_i / r_{ii}$
13: **else**
14: remove $\mathbf{q}_i$ and $\mathbf{s}_i$
15: **end if**
16: **end for**

Note that even though most columns in Q_x and Q_b are already A-orthogonal, orthogonalizing against the newest columns destroys this prior orthogonality. Straight application of the Gram-Schmidt procedure consequently requires (re-)orthogonalizing the entire set of columns, even if only a single column is changed.

Despite this improvement in memory use, the $O(nk^2)$ cost of Gram-Schmidt may become

Algorithm 2.4 Reduced Memory A-Orthogonal Modified Gram-Schmidt

for $i = k$ **to** 1 **do**
  $r_{ii} \leftarrow \sqrt{\mathbf{q}_i^T \mathbf{s}_i}$
  if $r_{ii} >$ tolerance **then**
    $\mathbf{q}_i \leftarrow \mathbf{q}_i / r_{ii}$
    $\mathbf{s}_i \leftarrow \mathbf{s}_i / r_{ii}$
    for $j = i - 1$ **to** 1 **do**
      $r_{ij} \leftarrow (q_i^T \mathbf{s}_j + \mathbf{s}_i^T \mathbf{q}_j)/2$
      $\mathbf{q}_j \leftarrow \mathbf{q}_j - r_{ij} \mathbf{q}_i$
      $\mathbf{s}_j \leftarrow \mathbf{s}_j - r_{ij} \mathbf{s}_i$
    end for
  else
    remove $\mathbf{q}_i$ and $\mathbf{s}_i$
  end if
end for

prohibitively expensive as n and k increase. A means of decreasing the cost of orthogonalization is the therefore desirable and is the subject of the next section.

2.3 Efficient projection space update

The Gram-Schmidt procedures outlined above do not make use of the fact that the majority of the vectors in Q_x and Q_b are already A-orthogonal. Instead, after adding new solution and right hand side vectors, Q_x and Q_b are completely reorthogonalized at every time step. This section presents a technique for reorthogonalizing a set of vectors in the A-norm with $O(nk)$ work by using this prior orthogonality. The resulting algorithm uses Givens rotations or Householder transformations in addition to Gram-Schmidt.[10]

2.3.1 Derivation

We begin by defining, in the same fashion as (2.8), a vector $\delta \mathbf{x}^i$ as the difference between $\mathbf{x}^i$ - the numerical solution at the ith time step - and $\widetilde{\mathbf{x}}^i$ - our approximation to $\mathbf{x}^i$ in the column space of Q_x.

$$\delta \mathbf{x}^i = \mathbf{x}^i - \widetilde{\mathbf{x}}^i = \mathbf{x}^i - Q_x \mathbf{r}. \tag{2.13}$$

[10] A personal communication from James Lottes assisted in the presentation of this section.

It immediately follows that

$$\delta \mathbf{b}^i = \mathbf{b}^i - Q_b \mathbf{r}. \tag{2.14}$$

We assume here that $k < k_{max}$ so there is room to append a column to Q_x and Q_b. This condition may be enforced by removing a column from Q_x and Q_b with the consequence that $\widetilde{\mathbf{x}}^i$ may not be equivalent to $\widehat{\mathbf{x}}^i$. This also implies $\Delta \mathbf{x}^i$ is not necessarily equivalent to $\delta \mathbf{x}^i$. We define $\mathbf{r}$ as before: $\mathbf{r} = Q_x^T \mathbf{b}^i$. The vector $\delta \mathbf{x}^i$ can be factored as a product of a scalar ρ and a vector $\mathbf{q}$ such that $\delta \mathbf{x}^i = \mathbf{q}\rho$. This allows us to express $\mathbf{x}^i$ as

$$\mathbf{x}^i = \widetilde{\mathbf{x}}^i + \delta \mathbf{x}^i = Q_x \mathbf{r} + \mathbf{q}\rho = \begin{bmatrix} Q_x & \mathbf{q} \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ \rho \end{bmatrix}. \tag{2.15}$$

We are interested in adding $\mathbf{x}^i$ to the column space of Q_x and begin by considering the addition of $\mathbf{x}^i$ to the column space of X. If we append $\mathbf{x}^i$ directly to X then we have a new set of solution vectors $[X \ \mathbf{x}^i]$. Substituting $Q_x R$ for X and $\mathbf{x}^i - Q_x \mathbf{r}$ for $\mathbf{x}^i$, we see

$$\begin{bmatrix} X & \mathbf{x}^i \end{bmatrix} = \begin{bmatrix} Q_x R & (Q_x + \mathbf{q}\rho) \end{bmatrix} = \begin{bmatrix} Q_x & \mathbf{q} \end{bmatrix} \begin{bmatrix} R & \mathbf{r} \\ \mathbf{0}^T & \rho \end{bmatrix}. \tag{2.16}$$

If we require $\|\mathbf{q}\|_A = 1$, then because $\|\delta \mathbf{x}^i\|_A = \|\mathbf{q}\rho\|_A = |\rho| \|\mathbf{q}\|_A = |\rho|$, we have $\rho = \|\delta \mathbf{x}^i\|_A = \sqrt{\langle \delta \mathbf{x}^i, \delta \mathbf{b}^i \rangle}$. Knowing ρ, we then determine $\mathbf{q} = \frac{\delta \mathbf{x}^i}{\rho}$.

A consequence of our definition of ρ and $\mathbf{q}$ is that $[Q_x \ \mathbf{q}]$ is A-orthogonal. It may appear here that our work is finished. We have a new matrix - $Q'_x = [Q_x \ \mathbf{q}]$ - that is orthogonal in the A inner product, the matrix contains $\mathbf{x}^i$ in its column space, and (assuming a reasonable implementation of $\widetilde{\mathbf{x}}^i = Q_x Q_x^T \mathbf{b}^i$) we computed it with $O(nk)$ work.

Unfortunately, this is not an adequate formulation of Q'_x. At the next time step $\widehat{i} = i + 1$, if $k = k_{max}$ we will desire to remove from Q'_x the components of $x^{\widehat{i}-k}$. This corresponds to throwing away the first column of Q'_x. However, throwing away this column has the side effect of removing from the column space components of vectors $\mathbf{x}^{\widehat{i}-1}$, $\mathbf{x}^{\widehat{i}-2}$, ... $\mathbf{x}^{\widehat{i}-(k-1)}$.[11] After removing the first column of Q'_x and $[X \ \mathbf{x}^i]$, $Span(Q'_x) \subset Span(\ [X \ \mathbf{x}^i]\ )$, so the best-fit

[11] We can see this is the case from the upper triangular format of $\begin{bmatrix} R & \mathbf{r} \\ \mathbf{0}^T & \rho \end{bmatrix}$.

projection onto $Span(Q'_x)$ is no longer also a best-fit projection onto $Span(\,[X\ \mathbf{x}^i]\,)$. This results in sub-optimal approximate solution vectors.

This difficulty is not limited to the first column. Throwing away any column besides the last (which only contains components of $\mathbf{x}^i$) has this side effect. It is thus tempting to avoid the problem by throwing away the last column instead. However, this column contains components of the latest solution vector. Throwing it away would cause Q_x to maintain a column space of stale solution vectors (i. e. solution vectors older than $\mathbf{x}^{i-k}$) which may not generate approximate solutions with small residuals. One could remove all or most columns of Q_x and rebuild the column space when the approximation $\widehat{\mathbf{x}}^i$ becomes poor or the matrix becomes full. However, this would still result in suboptimal projections at many time steps (as Q_x would either be stale or in the rebuilding phase much of the time).

With a small alteration to the above formulation and an additional constant factor of work, it is possible to avoid these adverse effects. Rather than append the latest solution vector, we instead prepend it. This gives us

$$\begin{bmatrix}\mathbf{x}^i & X\end{bmatrix} = \begin{bmatrix}(Q_x + \mathbf{q}\rho) & Q_x R\end{bmatrix} = \begin{bmatrix}Q_x & \mathbf{q}\end{bmatrix} \begin{bmatrix}\mathbf{r} & R \\ \rho & \mathbf{0}^T\end{bmatrix}. \tag{2.17}$$

Note that

$$\begin{bmatrix}Q_x & \mathbf{q}\end{bmatrix} \begin{bmatrix}\mathbf{r} & R \\ \rho & \mathbf{0}^T\end{bmatrix}$$

is an incomplete oblique QR factorization of the matrix $[\mathbf{x}^i\ X]$. We can complete the QR factorization by applying an orthogonal transformation via a matrix H to zero all but the first entry of the leading vector $\begin{bmatrix}\mathbf{r} \\ \rho\end{bmatrix}$ so $H \begin{bmatrix}\mathbf{r} \\ \rho\end{bmatrix} = \begin{bmatrix}\alpha \\ \mathbf{0}\end{bmatrix} = \alpha\mathbf{e}_1$. The product $H \begin{bmatrix}R \\ \mathbf{0}^T\end{bmatrix}$ is upper Hessenberg, from which it follows $H \begin{bmatrix}\mathbf{r} & R \\ \rho & \mathbf{0}^T\end{bmatrix} = R'$ is upper triangular.

If we let $Q'_x = [Q_x \ \mathbf{q}]\, H^T$, then because

$$\begin{aligned}(Q'_x)^T A Q'_x &= \left(\begin{bmatrix} Q_x & \mathbf{q} \end{bmatrix} H^T\right)^T A \begin{bmatrix} Q_x & \mathbf{q} \end{bmatrix} H^T \\ &= H \begin{bmatrix} Q_x & \mathbf{q} \end{bmatrix}^T A \begin{bmatrix} Q_x & \mathbf{q} \end{bmatrix} H^T \\ &= H^T H = I,\end{aligned} \tag{2.18}$$

we see Q'_x is A-orthogonal. Further noting

$$\begin{aligned}\begin{bmatrix} \mathbf{x}^i & X \end{bmatrix} &= \begin{bmatrix} Q_x & \mathbf{q} \end{bmatrix} \begin{bmatrix} \mathbf{r} & R \\ \rho & \mathbf{0}^T \end{bmatrix} \\ &= \begin{bmatrix} Q_x & \mathbf{q} \end{bmatrix} H^T H \begin{bmatrix} \mathbf{r} & R \\ \rho & \mathbf{0}^T \end{bmatrix} \\ &= Q'_x R',\end{aligned} \tag{2.19}$$

we see $Q'_x R'$ constitutes a complete QR factorization (in the A-norm) of $[\mathbf{x}^i \ X]$.

A subtle side effect of our decision to prepend $\mathbf{x}^i$ is that the structure of X changes. When appending, we recursively define $X^j := [X^{j-1} \ \mathbf{x}^j]$ with base case $X^{j-k} := [\mathbf{x}^{j-k}]$. By this definition, $[X \ \mathbf{x}^i] = [\mathbf{x}^{i-k} \ \mathbf{x}^{i-(k-1)} \ \ldots \ \mathbf{x}^{i-1} \ \mathbf{x}^i]$. Prepending, by contrast, recursively defines $X^j := [\mathbf{x}^j \ X^{j-1}]$ with base case $X^{j-k} := [\mathbf{x}^{j-k}]$, so the ordering of columns in the resulting matrix, $[\mathbf{x}^i \ X] = [\mathbf{x}^i \ \mathbf{x}^{i-1} \ \ldots \ \mathbf{x}^{i-(k-1)} \ \mathbf{x}^{i-k}]$, is reversed.

A result of this ordering is that $\mathbf{x}^i = \mathbf{q} R'_{11} = \mathbf{q}\alpha$. More generally, the jth column of $[\mathbf{x}^i \ X]$ is a linear combination of the 1st through the jth column of Q'_x. Crucially, only $\mathbf{x}^{i-k}$ depends on the last column of Q'_x. This means components of $\mathbf{x}^{i-k}$, the oldest solution vector in the column space of Q'_x, can be removed while keeping the remaining $k-1$ solution vectors in the column space. Consequently, if $k = k_{max}$, we can update the column space at every time step by simply overwriting the final vector with $\delta\mathbf{x}^i / \|\delta\mathbf{x}^i\|_A$ and performing the action of right multiplying the resulting matrix by H^T.

Because

$$A \begin{bmatrix} Q_x & \mathbf{q} \end{bmatrix} = A \begin{bmatrix} Q_x & \mathbf{q} \end{bmatrix} H^T H = \begin{bmatrix} Q_b & \frac{\delta \mathbf{b}^i}{\rho} \end{bmatrix}, \tag{2.20}$$

it follows that

$$Q'_b = AQ'_x = A \begin{bmatrix} Q_x & \mathbf{q} \end{bmatrix} H^T = \begin{bmatrix} Q_b & \frac{\delta \mathbf{b}^i}{\rho} \end{bmatrix} H^T \tag{2.21}$$

so we can use the same procedure to obtain Q'_b.

Beyond the derivation here, R and R' are never used and we can avoid forming them explicitly. The product $Q_x R$ is useful for error analysis, however, and may be recovered as the product $Q_x Q_x^T B$ by extension of (2.4) and (2.5).[12]

The matrix H can be found using Householder transformations or Givens rotations. The multiplication $[Q_x\ \mathbf{q}]\, H^T$ apparently has cost $O(nk^2)$. However, we shall see in the following sections it is possible to perform the action of this multiplication with only $O(nk)$ work. To conserve memory we update $[Q_x\ \mathbf{q}]$ in place rather than allocate a separate matrix to store Q'_x.

2.3.2 Givens rotations

A Givens rotation, also known as a *plane rotation*, rotates a vector in a plane such that a component of the vector in the plane is annihilated, but the Euclidean norm of the resulting vector is equal to that of original vector.[26] In two dimensions, the system appears in one form as

$$G \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}, \tag{2.22}$$

where s and c are the sine and cosine of the angle of rotation respectively. More generally, the rotation

$$\begin{bmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \dots & c & \dots & s & \dots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & -s & \dots & c & \dots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_i \\ \vdots \\ v_j \\ \vdots \\ v_n \end{bmatrix} = \begin{bmatrix} v_1 \\ \vdots \\ r \\ \vdots \\ 0 \\ \vdots \\ v_n \end{bmatrix} \tag{2.23}$$

[12] Perhaps more straightforwardly, $Q_x Q_x^T B = Q_x Q_x^T AX = Q_x Q_x^T A Q_x R = Q_x R$.

annihilates the jth component of a vector while modifying the ith component ($i \neq j$) so the norm stays constant. $G^T G = I$ so a Givens rotation is an orthogonal transformation.

Solving the system in (2.22) while imposing the constraints $r^2 = \sqrt{v_1^2 + v_2^2}$ and $c^2 + s^2 = 1$ gives us

$$c = \frac{v_1}{\sqrt{v_1^2 + v_2^2}}, \qquad s = \frac{v_2}{\sqrt{v_1^2 + v_2^2}}$$

as solutions. However, the negatives of c, s, or r are also valid solutions. This flexibility has resulted in a great deal of variance in implementations of Givens rotation. We adopt the sign conventions of Bindel, Demmel, and Kahan for our pseudocode in Algorithm 2.5.[48] Givens rotation breaks down in the case $v_i = v_j = 0$ so this case must be handled separately.

Algorithm 2.5 Givens Rotation

```
1: function GIVENS(v_i, v_j)
2:     if v_j ≠ 0 then
3:         h ← HYPOT(v_i, v_j)
4:         h_inv ← 1/h
5:         c ← |v_i| h_inv
6:         s ← COPYSIGN(h_inv, v_i) v_j
7:         r ← COPYSIGN(1, v_i) h
8:     else
9:         c ← 1
10:        s ← 0
11:        r ← v_i
12:    end if
13:    return c, s, r
14: end function
```

Algorithm 2.5 differs slightly from the Givens rotation algorithm suggested by Bindel, Demmel, and Kahan. The three case Givens algorithm they formulate is faster if an input, v_i or v_j here, is frequently zero. This is a rare occurrence in our context so we opt to eliminate a redundant branch while still ensuring division by zero does not occur.

The COPYSIGN function, available in many programming languages, returns the magnitude of the first argument with the sign of the second argument. In some programming languages such as Fortran this function is simply called SIGN. Other languages such as Matlab/Octave have a single argument SIGN function that is instead a SIGNUM function (i. e. it returns -1, 0, or 1 for negative, zero, and positive inputs respectively). However, without special

handling of 0 case SIGNUM cannot be used in Algorithm 2.5. To prevent confusion we avoid using SIGN here.

Calculating the Euclidean norm using the familiar form $\sqrt{a^2+b^2}$ can lead to loss of accuracy in finite precision arithmetic due to underflow or overflow in the intermediate values a^2 or b^2.[49] The latter is potentially a concern in our context because values in $[\mathbf{r}\ \rho]^T$ may be small. The equivalent expressions, $b\sqrt{1+\left(\frac{a}{b}\right)^2}$ and $a\sqrt{1+\left(\frac{b}{a}\right)^2}$, produce results accurate to numerical precision for the cases $|b| \geq |a|$ and $|a| \geq |b|$ respectively (though the case $a = b = 0$ must be specially handled). In many programming languages, an implementation of accurate hypotenuse calculation is available under the function name HYPOT.

We can annihilate all but the first entry of $[\mathbf{r}\ \rho]^T$ using a series of k Givens matrices so

$$H \begin{bmatrix} \mathbf{r} \\ \rho \end{bmatrix} = G_k G_{k-1} \dots G_2 G_1 \begin{bmatrix} \mathbf{r} \\ \rho \end{bmatrix} = \begin{bmatrix} \alpha \\ \mathbf{0} \end{bmatrix}. \tag{2.24}$$

The matrix H, formed as a product of Givens matrices, is Hessenberg at best[18] so the direct matrix-matrix product $[Q_x\ \mathbf{q}]H^T$ is a costly $O(nk^2)$ operation. To avoid this, a tempting alternative is to perform the action of multiplying by H^T by forming each G_i^T as a sparse matrix and employing a series of sparse matrix-vector products to obtain Q'_x.

$$Q'_x = \left(\dots \left(\left(\begin{bmatrix} Q_x & \mathbf{q} \end{bmatrix} G_1^T \right) G_2^T \right) \dots G_k^T \right). \tag{2.25}$$

However, this still results in many unnecessary operations because - except for four entries - each G_i is nearly the identity matrix. Instead, we take advantage of the structure of each Givens matrix and directly modify only the columns each G_i^T affects.[26] Algorithm 2.6 shows the procedure. Initially $Q = [Q_x\ \mathbf{q}]$. We calculate each c_j and s_j as if annihilating $\mathbf{z} := [\mathbf{r}\ \rho]^T$ from bottom to top. Subsequently, we modify Q in-place, performing the action of QG_j^T using only c_j and s_j and two columns of Q. At the end of the procedure, Q is equivalent to Q'_x. The c and s values could instead be calculated inside the second for-loop. We move the calculation to a separate loop to make the procedure more readily parallelizable.

In terms of floating point operations, the cost of Algorithm 2.6 is dominated by the $O(6nk)$ operations in the nested for-loops. With fused multiply-add (FMA) operations, the cost may

Algorithm 2.6 Reorthogonalization by Givens Rotations

```
for j = k+1, k, ... 2 do
    c_j, s_j, z_{j-1} ← GIVENS(z_{j-1}, z_j)
end for
for j = k+1, k, ... 2 do
    l ← j − 1
    for i = 1 to n do
        t ← c_j Q_il + s_j Q_ij
        Q_ij ← −s_j Q_il + c_j Q_ij
        Q_il ← t
    end for
end for
```

be closer to $O(4nk)$. Other potentially costly operations are the $2k$ divisions and k square root operations in the first loop. However, if $n >> k$ then the cost of these is relatively small. Almost no additional storage is required for this algorithm except for two small vectors to store each c_j and s_j (and even these can be reduced to two floating point variables by moving the call to GIVENS inside the second loop).

2.3.3 Householder reflections

Householder reflections, also known as *Householder transformations*, reflect a vector across a hyperplane such that the resulting vector has the same norm as the original vector but with all components except one annihilated.[26] The Householder matrix has the form

$$H = I - 2\frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}}. \tag{2.26}$$

By inspection, we see H is orthogonal and symmetric. In our case, the *Householder vector* $\mathbf{v}$ is chosen to to annihilate all but the first entry of $\mathbf{z} := [\mathbf{r}\ \rho]^T$. It consequently takes the form

$$\mathbf{v} = \mathbf{z} + \alpha\mathbf{e}_1 \tag{2.27}$$

where α is the Euclidean norm of $\mathbf{z}$ with the sign of z_1 (i. e. $\alpha = \text{COPYSIGN}(\|\mathbf{z}\|_2, z_1)$). The oppositely signed α is also a valid choice, but we make α have the same sign as z_1 to avoid cancellation if z_1 and α are close in magnitude.

A consequence of the form of the Householder matrix is that the action of the matrix-matrix product QH^T can be computed with $O(nk)$ work.[26] Observe

$$QH^T = QH = Q\left(I - 2\frac{\mathbf{v}\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}}\right) = Q - 2\frac{(Q\mathbf{v})\mathbf{v}^T}{\mathbf{v}^T\mathbf{v}}. \tag{2.28}$$

The last expression in (2.28) requires only a matrix-vector product, inner-product, outer-product, and matrix-matrix subtraction. Algorithm 2.7 shows the procedure for calculating $\mathbf{v}$ and performing the action of QH^T. Note that this algorithm does not correctly update $\mathbf{z}$ as we avoid forming R' anyway.

Algorithm 2.7 Reorthogonalization by Householder (version 1)

$\omega \leftarrow z_1$
$z_1 \leftarrow 0$
$\gamma \leftarrow \mathbf{z}^T\mathbf{z}$
$h \leftarrow \sqrt{\omega^2 + \gamma}$
$z_1 \leftarrow \omega + \text{COPYSIGN}(h, \omega)$
$s \leftarrow 2/(z_1^2 + \gamma)$
$\mathbf{v} \leftarrow Q\mathbf{z}$
$\mathbf{z} \leftarrow s\mathbf{z}$
$M \leftarrow \mathbf{v}\mathbf{z}^T$
$Q \leftarrow Q - M$ ▷ Equivalent to $Q \leftarrow Q - (Q\mathbf{z})(s\mathbf{z})^T$

In terms of floating point operations, the main costs of Algorithm 2.7 are the matrix-vector product $Q\mathbf{z}$, the vector-vector outer product $\mathbf{v}\mathbf{z}^T$, and the matrix subtraction $Q - M$ for a total cost of $O(4nk)$. This is a theoretical improvement over the Givens rotation algorithm. The algorithm also requires only one square root and one division, which compares favorably with the Givens rotation algorithm.

Householder transformations are normally applied to an entire vector at once. However, if one entry in the vector is much larger than the others, then the sum of squares of the components may lose digits of precision. This can result a less accurate projection. One can ameliorate this loss of precision by applying a series of small Householder transformations to annihilate components of $[\mathbf{r}\ \rho]^T$ one at a time rather than a single large transformation. In this form, the series of Householder transformation behaves similarly to the series of Given's transformations, annihilating entries from the bottom up. Algorithm 2.8 shows the

procedure.

Algorithm 2.8 Reorthogonalization by Householder (version 2)

1: **for** $j = k+1, k, \ldots 2$ **do**
2: $l = j - 1$
3: $h \leftarrow \text{HYPOT}(z_l, z_j)$
4: $\alpha \leftarrow \text{COPYSIGN}(h, z_l)$
5: $\omega \leftarrow z_l + \alpha$
6: $s \leftarrow 2/(\omega^2 + z_j^2)$
7: $\nu_j \leftarrow \omega s$
8: $u_j \leftarrow z_j s$
9: $z_l \leftarrow -\alpha$
10: **end for**
11: **for** $j = k+1, k, \ldots 2$ **do**
12: $l = j - 1$
13: **for** $i = 1$ **to** n **do**
14: $t \leftarrow \nu_j Q_{il} + u_j Q_{ij}$
15: $Q_{ij} \leftarrow Q_{ij} - t u_j$
16: $Q_{il} \leftarrow Q_{il} - t \nu_j$
17: **end for**
18: **end for**

The theoretical cost of Algorithm 2.8 is nearly identical to that of Algorithm 2.6. The nested for-loop, responsible for most of the work of the algorithm, has $O(7nk)$ floating point operations. As with the Givens algorithm, FMA operations may effectively reduce this to $O(4nk)$ floating point operations. The additional flop required for 2×2 Householder reflections over 2×2 Givens rotations perhaps factored into the wider use of the latter. However, adoption of FMA may have diminished this difference. A potentially favorable feature of this Householder scheme over Givens rotation is that the update of Q_{il} is no longer dependent on the update of Q_{il}. This may provide the compiler more opportunities for pipelining.

Like the Givens rotation algorithm, The first for-loop requires $2k$ divisions and k square root operations. Instead of explicit branching, the Householder algorithm uses a COPYSIGN, though this is unlikely to significantly affect overall reorthogonalization performance.

2.3.4 Efficient re-orthogonalization algorithm

Having developed ways to perform the operation of $[Q\ \mathbf{q}]\,H^T$ with $O(nk)$ cost, we now outline the overall procedure for updating the projection space. Algorithm 2.9 shows the pseudo-code. The algorithm begins by checking if $k = k_{max}$ and discarding the last column of Q_x and Q_b if this is the case. The last column will soon be overwritten so it generally suffices to soft delete the columns by decrementing k. In lines 7, 8, and 9 we calculate $\delta\mathbf{x}$ and $\delta\mathbf{b}$ by subtracting from $\mathbf{x}$ and $\mathbf{b}$ their components already in Q_x and Q_b. In practice this must be done using A-orthogonal MGS or A-orthogonal CGS2 to prevent loss of orthogonality.

Once we calculate $\delta\mathbf{x}$ and $\delta\mathbf{b}$, we then calculate ρ as the square root of their inner product. It is useful at this point to use ρ to ensure $\delta\mathbf{x}$ and $\delta\mathbf{b}$ still have substantial components. This guards against performing useless work re-orthogonalizing Q_x and Q_b against vectors that are nearly $\mathbf{0}$. If ρ exceeds some set tolerance, then $\delta\mathbf{x}$ and $\delta\mathbf{b}$ are linearly independent of Q_x and Q_b. We proceed to form $[\mathbf{r}\ \rho]$, $\left[Q_x\ \frac{\delta\mathbf{x}}{\rho}\right]$, and $\left[Q_b\ \frac{\delta\mathbf{b}}{\rho}\right]$ by appending to $\mathbf{r}$, Q_x and Q_b respectively. Finally, using one of Algorithms 2.6, 2.7, or 2.8 we re-orthogonalize Q_x and Q_b.

We form $\delta\mathbf{x}$ and $\delta\mathbf{b}$ as separate vectors to align the algorithm more closely to the preceding derivation. The algorithm may be simplified somewhat and the storage cost reduced by storing $\mathbf{x} - Q_x\mathbf{r}$ and $\mathbf{b} - Q_b\mathbf{r}$ back into $\mathbf{x}$ and $\mathbf{b}$. In particular, this eliminates the need for the *else* block in the second if-statement.

From a communication standpoint the algorithm may be attractive as it requires only two or three instances of global communication if lines 7, 8, and 9 are implemented with CGS2. If implemented with MGS, the algorithm requires only $k + 1$ instances of global communication. In both implementations, this is a substantial communication reduction compared with straightforward CGS2 or MGS. As an added benefit over Gram-Schmidt, this procedure requires no explicit shifting of vectors. The propagation of information instead is incorporated into the action of the matrix-matrix product. A potential drawback of not performing a complete orthogonalization procedure on Q_x and Q_b is eventual loss of orthogonality or skewing of the projection space due to propagated numerical errors. If detected, a complete Gram-Schmidt procedure applied to Q_x and Q_b could correct the former problem at a some-

Algorithm 2.9 Efficient A-orthogonal projection space update

if $k = k_{max}$ **then**
    Discard kth column of Q_x
    Discard kth column of Q_b
    $k \leftarrow k - 1$
end if
if $k > 0$ **then**
    $\mathbf{r} \leftarrow (Q_x^T \mathbf{b} + Q_b^T \mathbf{x})/2$
    $\delta\mathbf{x} \leftarrow \mathbf{x} - Q_x \mathbf{r}$
    $\delta\mathbf{b} \leftarrow \mathbf{b} - Q_b \mathbf{r}$
else
    $\delta\mathbf{x} \leftarrow \mathbf{x}$
    $\delta\mathbf{b} \leftarrow \mathbf{b}$
end if
$\rho \leftarrow \sqrt{\delta\mathbf{x}^T \delta\mathbf{b}}$
if $\rho >$ tolerance **then**
    Append ρ to $\mathbf{r}$
    Append $\delta\mathbf{x}/\rho$ to Q_x
    Append $\delta\mathbf{b}/\rho$ to Q_b
    Reorthogonalize Q_x using $\mathbf{r}$
    Reorthogonalize Q_b using $\mathbf{r}$
    $k \leftarrow k + 1$
end if

what increased overall cost. In our numerical experiments this has not been a problem and in fact some of the efficient algorithms appear to better maintain orthogonality.

CHAPTER 3

NUMERICAL EXPERIMENTS

The efficient projection space updating algorithms (i. e. Algorithm 2.9 combined with one of Algorithms 2.6, 2.7, or 2.8) were implemented in Nek5000 and the performance compared with reorthogonalization using reduced memory Gram-Schmidt procedures (Algorithms 2.3 and 2.4). Comparisons were made using several Nek5000 test cases.

Nek5000 is a high-order solver for computational fluid dynamics. Written in Fortran 77 and C and using MPI for parallel computation, it has been successfully scaled to over one million MPI processes. Nek5000 uses preconditioned conjugate gradients or GMRES as its linear solvers.

Performances tests were run on the campus cluster of the Urbana campus of the University of Illinois. The cluster consists of 312 nodes with two ten-core 2.5 GHz Intel E5-2670V2 (Ivy Bridge) cores per node. Each node has at least 64 GB of RAM. Internode communication is via an Intel i350 ethernet controller. The 2017-11-15 version of Nek5000 was compiled using version 18.0 of Intel's Fortran and C compilers and run using Intel's MPI implementation. Nek5000's default optimization flags were used during compilation. Error and accuracy tests were conducted on a 1.3 GHz Intel Xeon Phi 7210 with 64 cores, 16GB of MCDRAM, and 96 GB of DDR3 memory. In this environment, Nek5000 was compiled using version 4.8.5-11 of the GNU Compiler Collection with Nek5000's default optimization flags and run using the 2018 version of Intel's MPI implementation.

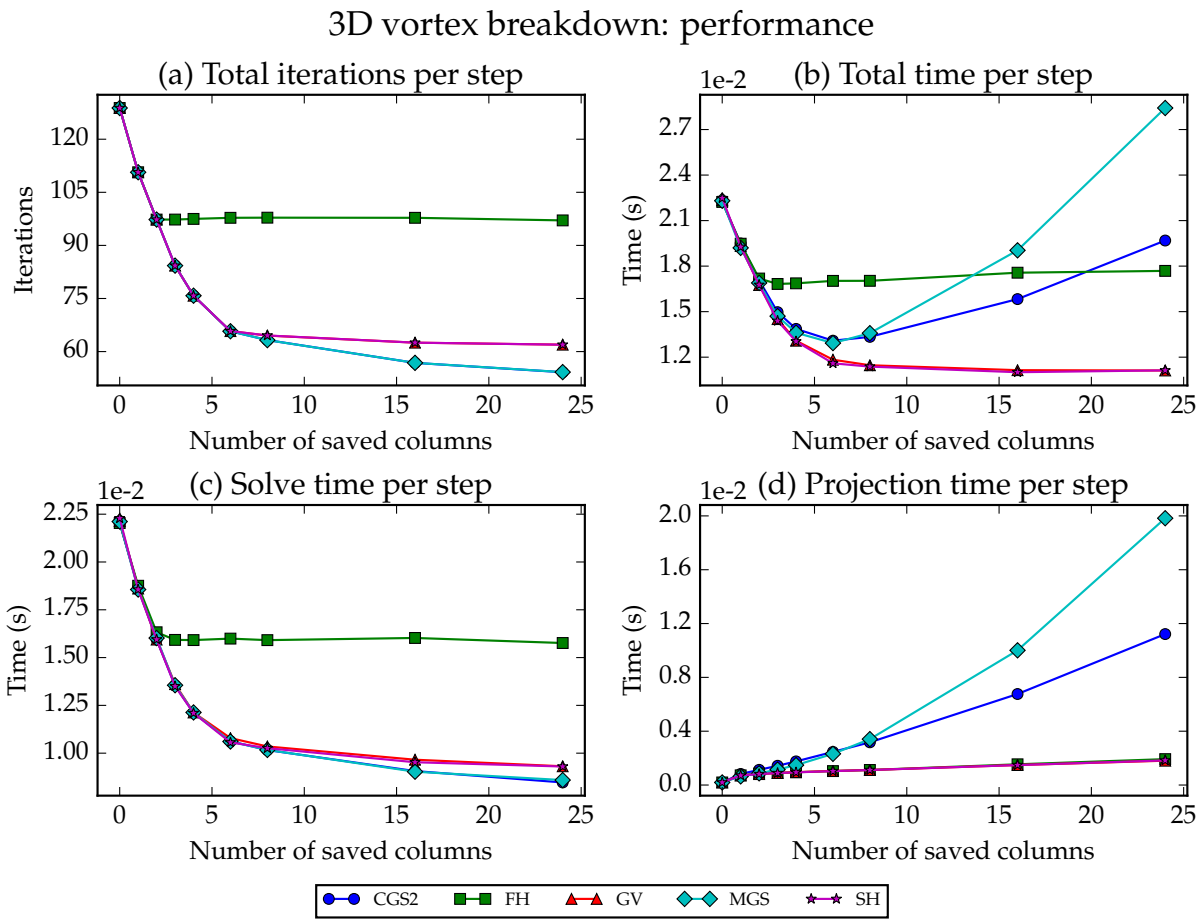

Figure 3.1: Performance of the reorthogonalization algorithms, as measured by average time per step and average iterations per step, on a test case simulating a 3D vortex breakdown. The projection time includes time required to calculate the appriximate solution and update the projection space. The total time is the sum of the per step solve time and projection time.

3.1 Case 1: 3D vortex breakdown

3.1.1 Performance

Figure 3.1 compares the performance of the reorthogonalization algorithms on the 140 element 3D vortex breakdown problem for an increasing maximum number of saved columns in Q_x and Q_b. The plots shown average over 200 time steps. The test case was run with a solver tolerance of 10^{-13}, a step size of 0.05 convective time units, and polynomial order nine. As this is a smaller case, only two nodes and forty processors were used. In these plots, and all plots that follow, MGS and CGS2 are the modified Gram-Schmidt and the

twice repeated classical Gram-Schmidt full reorthogonalization algorithms. GV, SH, and FH are the efficient reorthogonalization algorithm with Givens rotations, small 2×2 Householder transformations, and the full Householder matrix respectively.

Subfigure (a) shows the average number of solver iterations per step as the number of columns increases. In our test cases, Nek5000 uses projection for solving both pressure and velocity; the total number of solver iterations is the sum from these solves. We see here that MGS, CGS2, SH, and GV have a similar number of per step solver iterations up to about eight saved columns. After this point, SH and GV no longer significantly reduce their iteration counts while the Gram-Schmidt algorithms reduce their iteration counts further by a small amount. In contrast to these evidently effective algorithms, FH performs very poorly, showing no improvement beyond in iteration counts beyond two saved columns. Subfigure (c), showing the average time spend in the iterative solver, largely reflects the number of solver iterations seen in (a).

Subfigure (d) shows the average time per time step required to project the approximate solution and update the projection space. The Gram-Schmidt algorithms display a sharp increase in the projection time beyond eight saved columns. Additionally, at this point the reduced communication cost of CGS2 over MGS becomes apparent. In agreement with our complexity analysis, FH, GV, and SH appear to exhibit only linear growth in time cost as k increases.

The overall time required per step (the sum of the time spent in the iterative solver and the time spend updating the projection space) is shown in subfigure (b). As with the solver iteration counts, the projection time here is the sum of the projection times for pressure and velocity. The GS algorithms in this case reduce the total solve time until about $k = 6$. After this point, the cost appears to increase quadratically. As the solve time for these k values is small, it is clear that the projection time comes to dominate the total cost in this region. GV and SH improve in performance until about around eight columns are saved. After this point, there is little reduction in the overall run time, which again aligns with our observations of the per step solve time. Comparing their best cases, the relative improvement of GV and SH over CGS2 and MGS is a 15% reduction in total time. FH, unable to reduce the number of iterations per step, performs poorly in the total run time. Only at very high k when the

quadratic complexity of CGS2 and MGS is in full force does FH manage to perform within the range of MGS.

3.1.2 Stability and accuracy

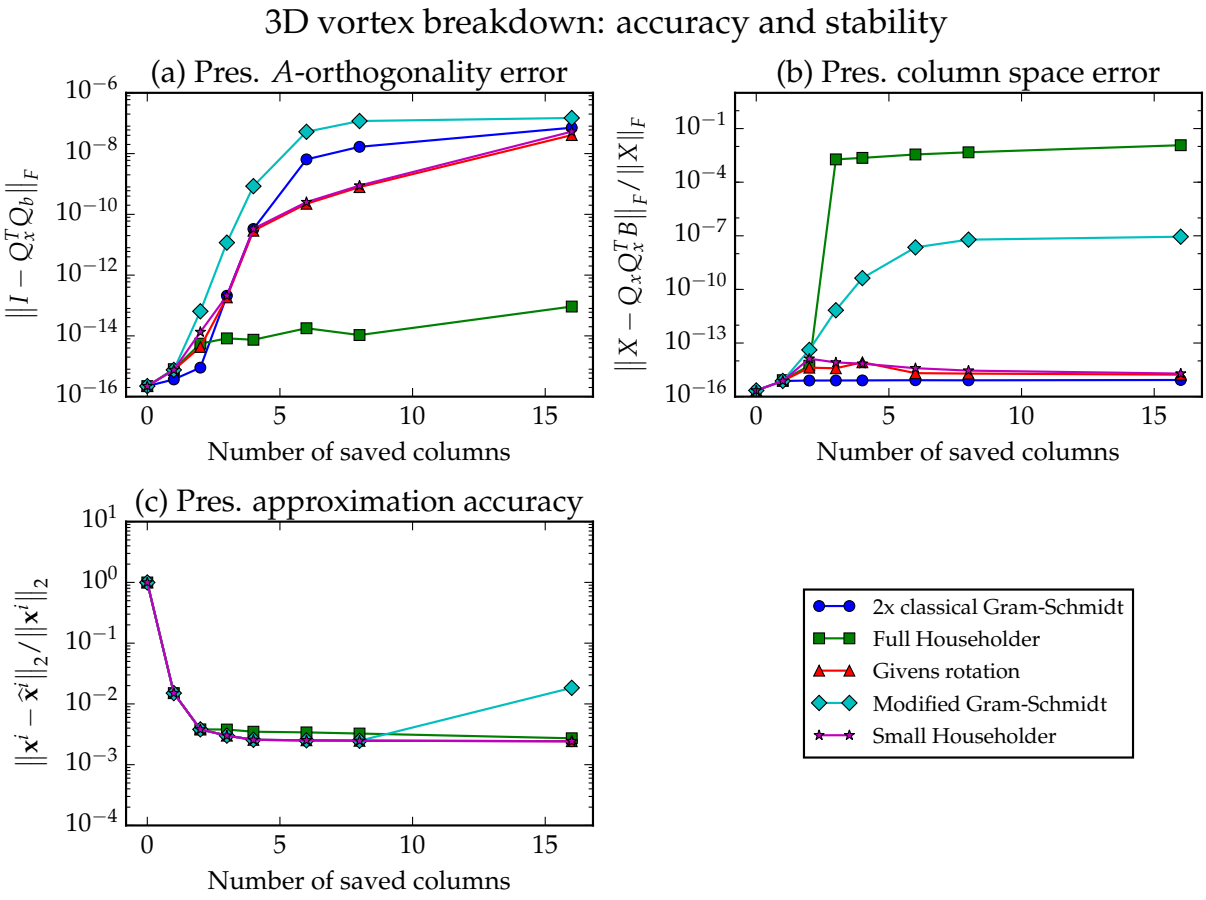

Figure 3.2: Stability and accuracy results for pressure projection from the 3D vortex breakdown case as the number of saved columns increases. Subfigure (a) shows the average Frobenius norm of the deviation from A-orthogonality of the pressure projection space. Subfigure (b) shows the average Frobenius norm of the deviation of the orthogonalized column space from the original column space. Subfigure (c) shows the average error of the pressure projection. Large errors at the early time steps mask more impressive results here.

The quality of an A-orthogonal projection is assessed in several ways. Most significantly, we assess the accuracy of the projection by calculating a norm of the relative difference between it and the numerical solution. Using the two-norm, this is $\|\mathbf{x}^i - \widehat{\mathbf{x}}^i\|_2 / \|\mathbf{x}^i\|_2$. We may also assess the quality of the projection space.[43] One way of doing this is to determine

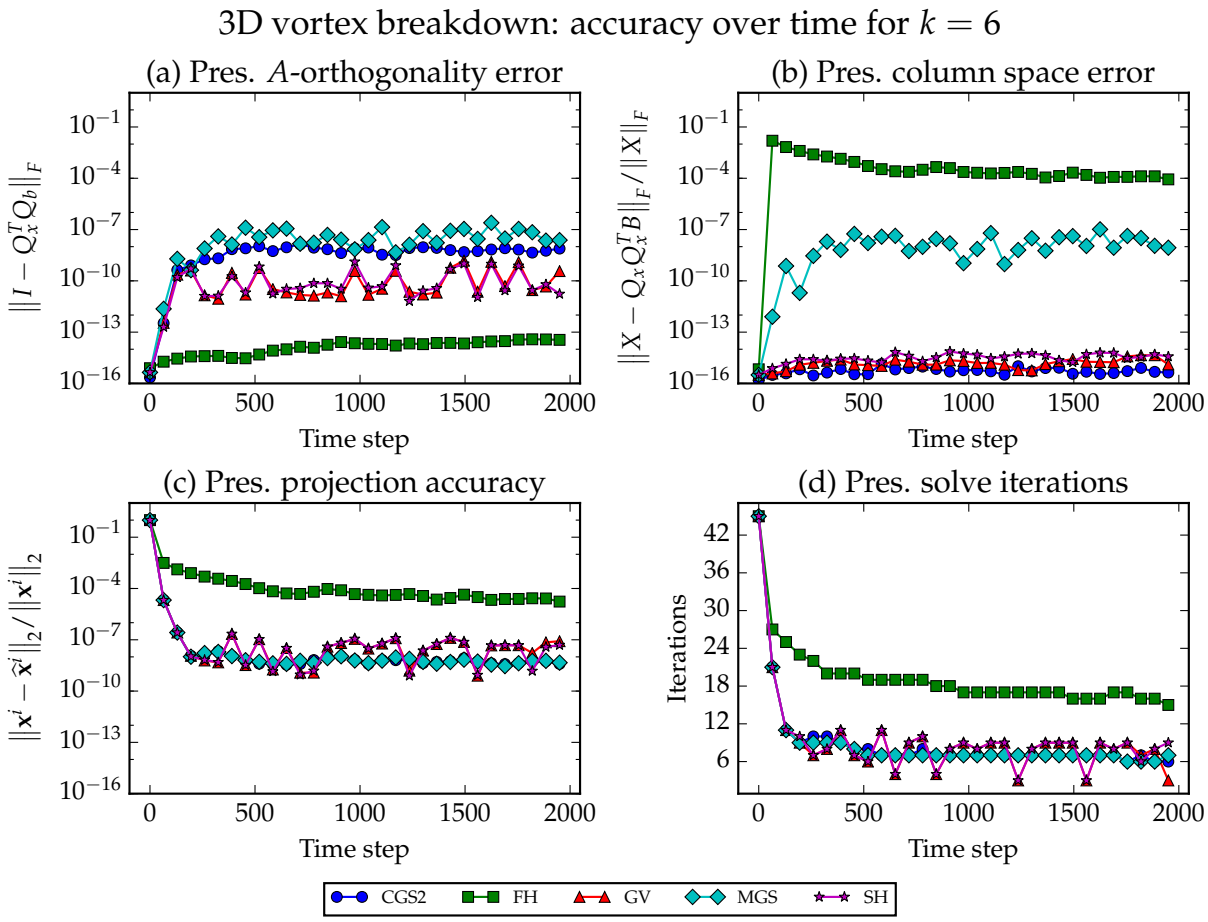

Figure 3.3: Stability and accuracy of pressure projection over the course of 2000 time steps for the 3D vortex breakdown case with $k = 6$. Data points are shown for every 64th time step. Subfigure (a) shows the average Frobenius norm of the deviation from A-orthogonality of the pressure projection space. Subfigure (b) shows the average Frobenius norm of the deviation of the orthogonalized column space from the original column space. Subfigure (c) shows the average error of the pressure projection. Subfigure (d) shows the number of iterations of the pressure solver.

the extent to which A-orthogonality is maintained. As A-orthogonality requires $Q_x^T A Q_x = I$, we can conveniently calculate this as $\left\|I - Q_x^T A Q_x\right\|_F = \left\|I - Q_x^T Q_b\right\|_F$. Another way of assessing the projection space is to determine the extent to which column space of the original matrix (X in this case) is preserved. Ordinarily this error in projection is calculated as a norm of $X - Q_x R$. We do not maintain R so we instead use the equivalent expression $\left\|X - Q_x Q_x^T B\right\|_F$ as mentioned in 2.3.1. This is made a relative error by dividing by the norm of X: $\left\|X - Q_x Q_x^T B\right\|_F / \|X\|_F$.

Figure 3.2 shows the average of these deviations over 2000 timesteps for the pressure

projection of the 3D vortex breakdown case. Most notably, the subfigure (a) shows FH preverves A-orthogonality orders of magnitude better than the other algorithms. The trade-off for this is revealed in subfigure (b), which shows FH maintains high a high measure of A-orthogonality by altering the column space of Q_x so it significantly diverges from X.

This is likely the case because the first entry of vector $\mathbf{z}$ in Algorithm 2.7 is the coefficient corresponding to the contribution of the solution at the immediate previous time step. When the solution at the current time step is similar to the solution at the previous time step, this coefficient may be many orders of magnitude larger than the smallest component of $\mathbf{z}$. In forming $\mathbf{v}$, this difference in magnitude has the effect of chopping off least significant digits from columns with smaller coefficients. The resulting Q'_x is A-orthogonal, but its column space is altered (particularly with respect to the oldest solutions) so it contains X' to fewer accurate digits. The end result is a poorer projection and more iterations in the solver. GV and SM avoid this problem by operating on only two adjacent columns at a time. Adjacent columns typically have coefficients that are closer together in magnitude which results in fewer lost digits of precision.

Examining the remaining algorithms, we see MGS performs worse than CGS2, GV, and SH in maintaining both the column space and A-orthogonality. Two iterations of CGS2 maintains the column space to near numerical precision, but trails GV and SH in maintaining orthogonality. GV and SH perform nearly identically. The two algorithms maintain A-orthogonality very well compared with the GS algorithms but trail CGS2 by a very small amount in preserving the column space. All algorithms besides FH appear to converge in the A-orthonality measure when k becomes large.

The plot of average approximation accuracy in subfigure (c) appears to show CGS2, MGS, GV, and SH have nearly identical performance. However, the average masks performance differences because the error at early timesteps is orders of magnitude larger than the error at later time steps. The plot does confirm FH produces less accurate approximations. The spike in the error of MGS at $k = 16$ is likely due to instability of the algorithm at high k.

Figure 3.3 shows measures of the stability and accuracy for the reorthogonalization algorithms over 2000 time steps of the 3D vortex breakdown case with $k = 6$. Values at every 64th time step are shown. Results here largely reflect those in Figure 3.2. In subfigure (c),

we can now see the effectiveness of projection that was previously obscured by averaging. CGS2 and MGS reduce the error of the initial approximation to the magnitude of 10^{-8} or 10^{-9}. GV and SH fluctuate somewhat more widely, but have perhaps a factor of 10 larger error. FH, as we previously observed, has error orders of magnitude larger still.

Overall, all algorithms display fairly stable behavior after around timestep 250. Presumably this is when the algorithms have finished building out their projection spaces and the problem has evolved past any initial transient. GV and SH display very similar behavior at all time, though GS appears marginally better at maintaining the column space. FH increases in projection accuracy over time, though this is likely due to the problem approaching steady state.

3.2 Case 2: 3D flow past a hemisphere

3.2.1 Performance

Performance results of the reorthogonalization algorithms a 2,042 element test case simulating three-dimensional fluid flow past a hemisphere are shown in Figure 3.4. The test case was run for 200 time steps with a step size of 0.005 convective time units, polynomial order 5, and a solver tolerance of 10^{-14}. Eight nodes and 160 total processors on the cluster were used.

Results here are similar to the 3D vortex breakdown case. The performance improvement provided by efficient reorthogonalization is here reduced to about 7%. As in the previous test, FH is unable to decrease the number of iterations once more than three columns are saved. The Gram-Schmidt algorithms once again appear slightly more effective at decreasing the number of iterations at higher k. As the optimal k value for GS is once again six, this does not appear helpful minimizing the total time per step.

In subfigure (d), we can now see the FH updating algorithm scales worse than GV and SH. This is somewhat surprising as it theoretically has a lower flop count. However, as previously noted, Algorithm 2.7 requires twice loading the entirety of Q from memory. If Q cannot be stored in the cache, then these loads may be slow. While SH and GV access most

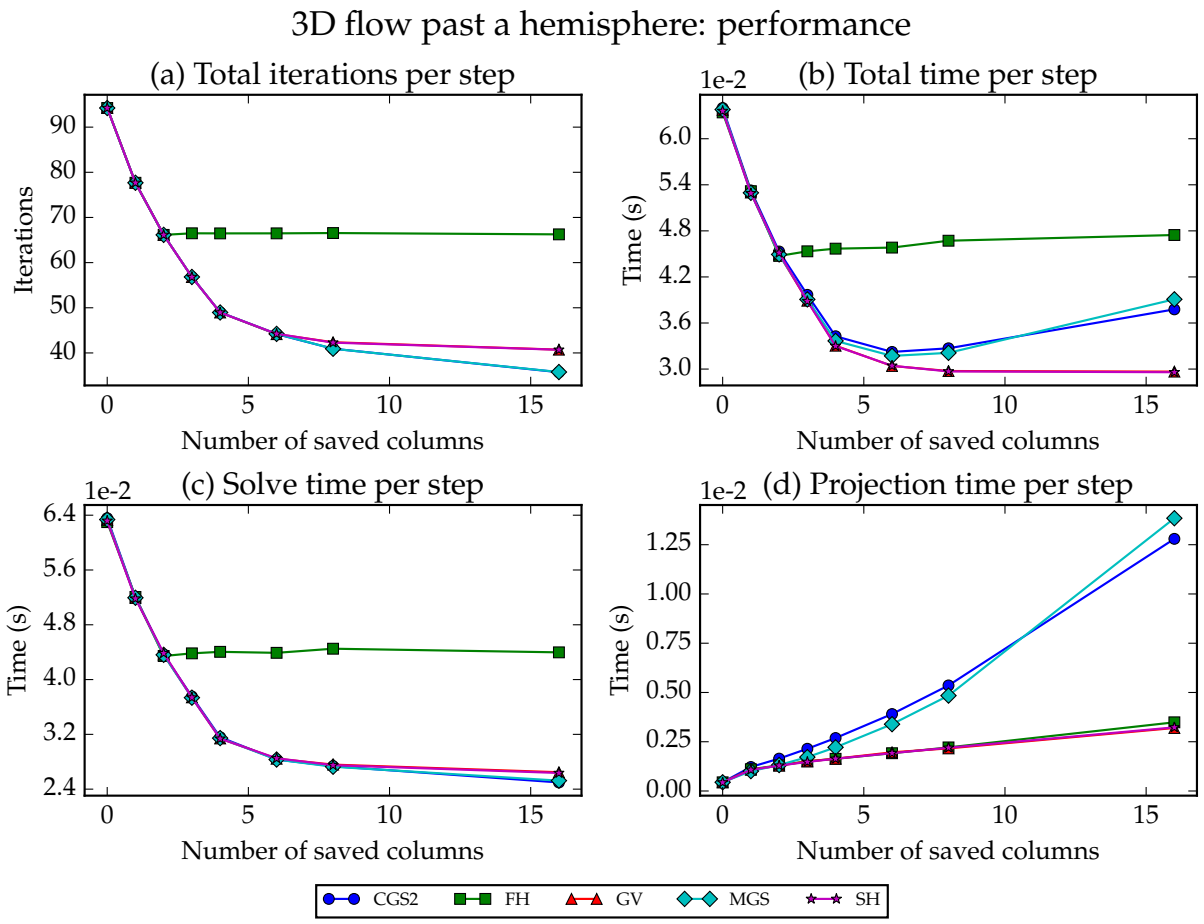

Figure 3.4: Performance results of the reorthogonalization algorithms on a test case simulating three-dimensional flow past a hemisphere.

columns of Q twice (the first and last columns are accessed only once), they work with only two columns at a time, one of which was used at the previous iteration of the inner loop. Because less data is used, it is more likely this column is retained in cache.

3.2.2 Stability and accuracy

In contrast to the 3D vortex case, FH in this case is is unable to maintain A-orthogonality of the pressure projection space. Figure 3.5 shows as the maximum number of saved columns increases, the deviation grows to approach that of the other algorithms. The abrupt change in column space error and the average approximation accuracy at $k = 24$ is due to FH becoming unstable.

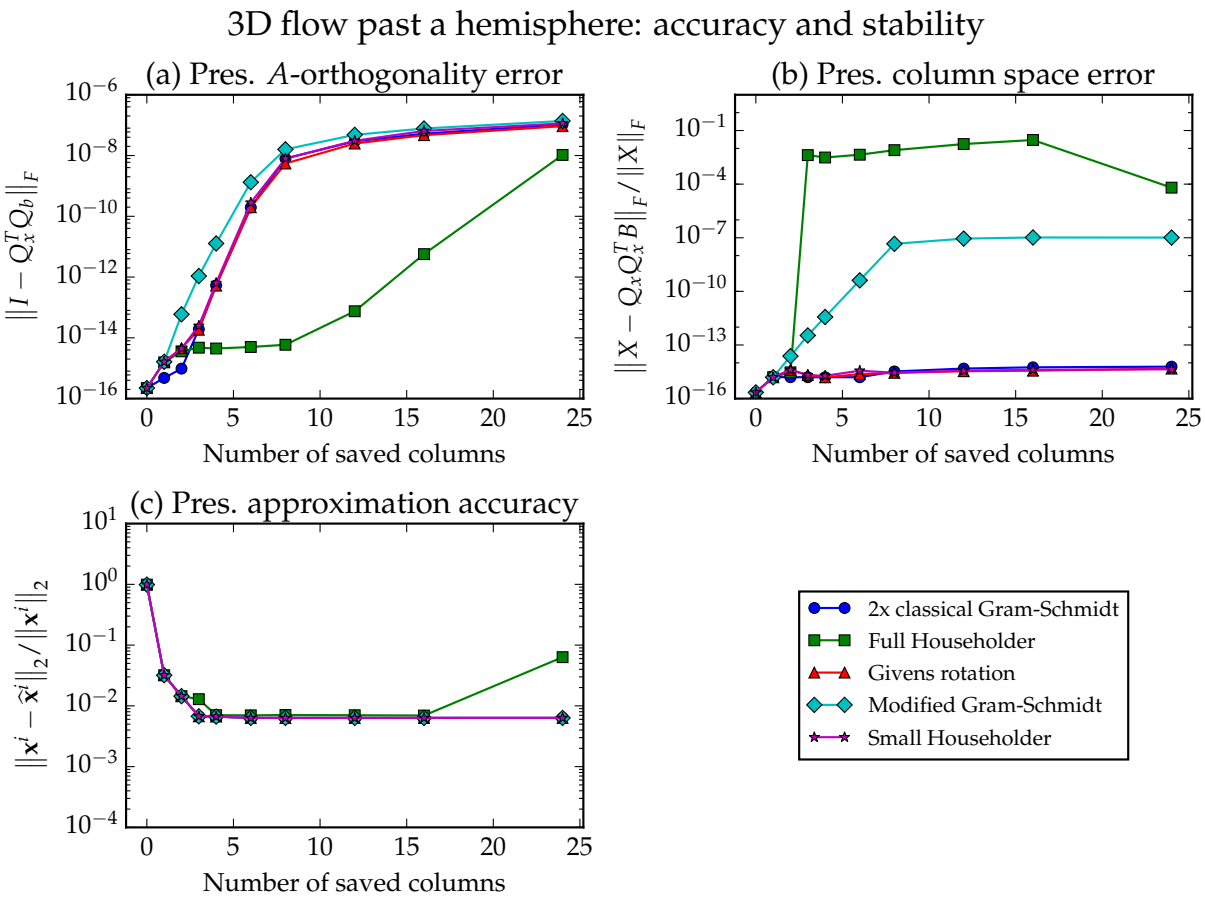

Figure 3.5: Measurements of the average accuracy of the reorthogonalization algorithms over 200 time steps of the flow past a hemisphere test case.

CGS2, GV, and SH appear to similarly maintain both the A-orthogonality of the column space and its integrity as compared with the column space of X. In Figures 3.5 and 3.6 (the latter plotted at every 13th time step), the three algorithms again perform marginally better than MGS in orthogonality and orders of magnitude better in preserving the original column space. Here, this error not appear to affect the accuracy of projections made from the MGS reorthogonalized basis. Once again, the averaging of the approximation error masks important detail better shown in 3.6 subfigure (a), though it does inform us of the instability in FH.

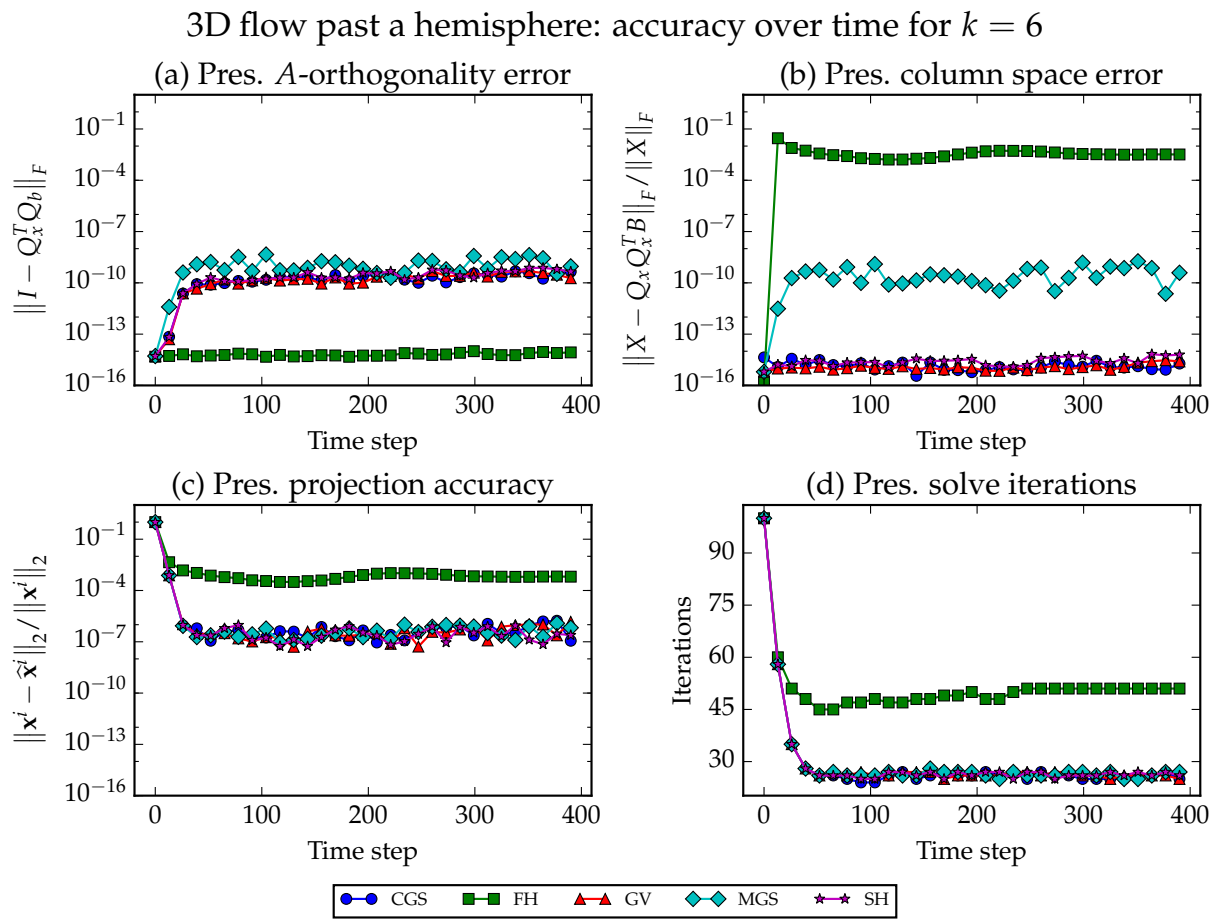

Figure 3.6: Accuracy measurements at every six time steps within a run of the flow past a hemisphere case for $k = 6$.

3.3 Case 3: 3D flow in a carotid artery

3.3.1 Performance

Our tests on a 3D carotid artery fluid simulation are perhaps more indicative of the capabilities of the efficient updating algorithms on real world problems. Containing 2,544 elements, we ran the fluid simulation at polynomial order 4 with a step size of 5×10^{-5} convective time units and a solver tolerance of 10^{-8}. Tests were run on 8 nodes and 160 total processors.

Performance results from the carotid artery simulation shown in 3.7 and average over 200 time steps. The results appear generally consistent with the prior two test cases, with

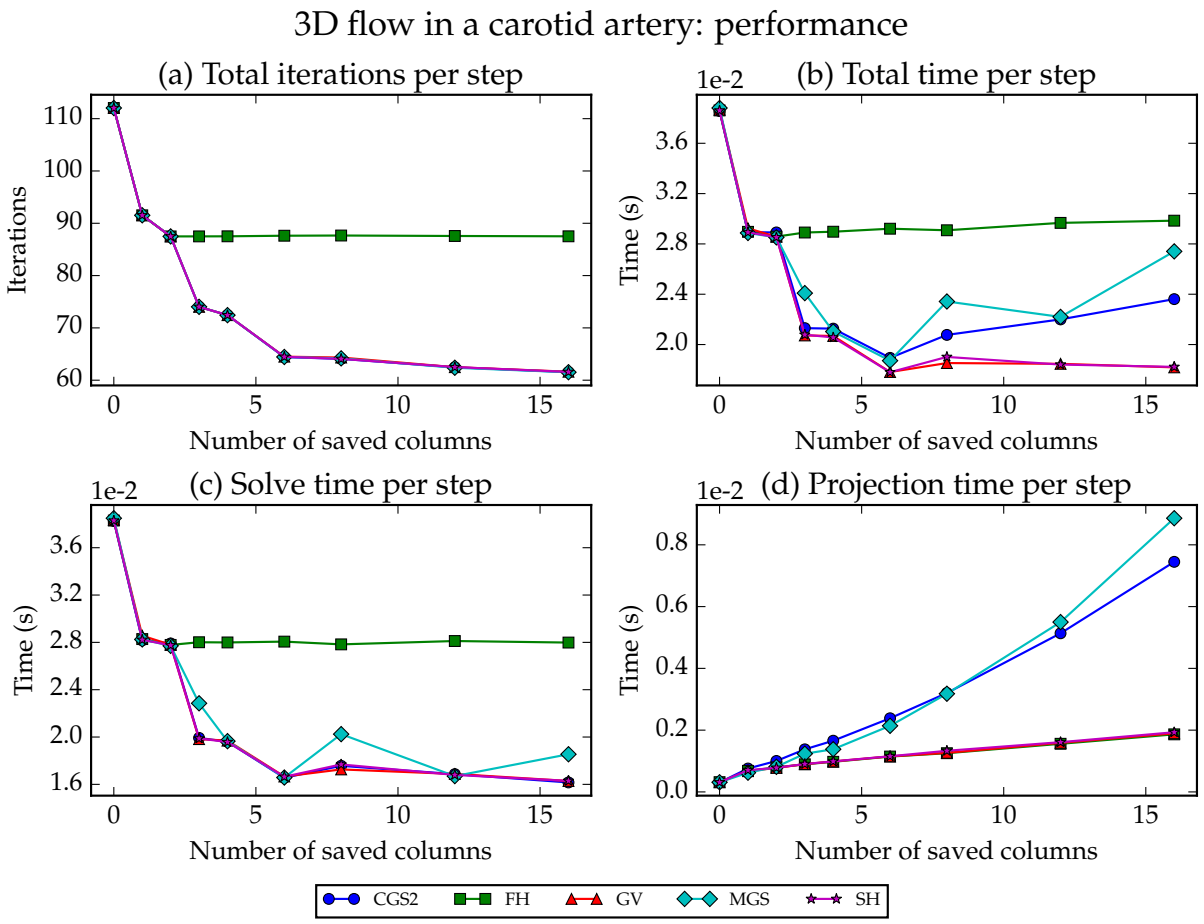

Figure 3.7: Performance of the reorthogonalization algorithms on a 3D simulation of blood flow through a carotid artery with stenosis.

around six being the optimal number of saved columns for all algorithms except FH. One notable difference is an elevation in the MGS solve time (though not the MGS iteration count) for the three, eight and sixteen column cases. This was duplicated in several test runs, though its cause remained elusive. Another difference we note is the stairstep pattern of decreasing iteration counts. This seems to indicate that in this problem the information of every other time step is more useful for projection. The overall improvement of efficient reorthogonalization over Gram-Schmidt methods is about a 5% reduction in the total time per step.

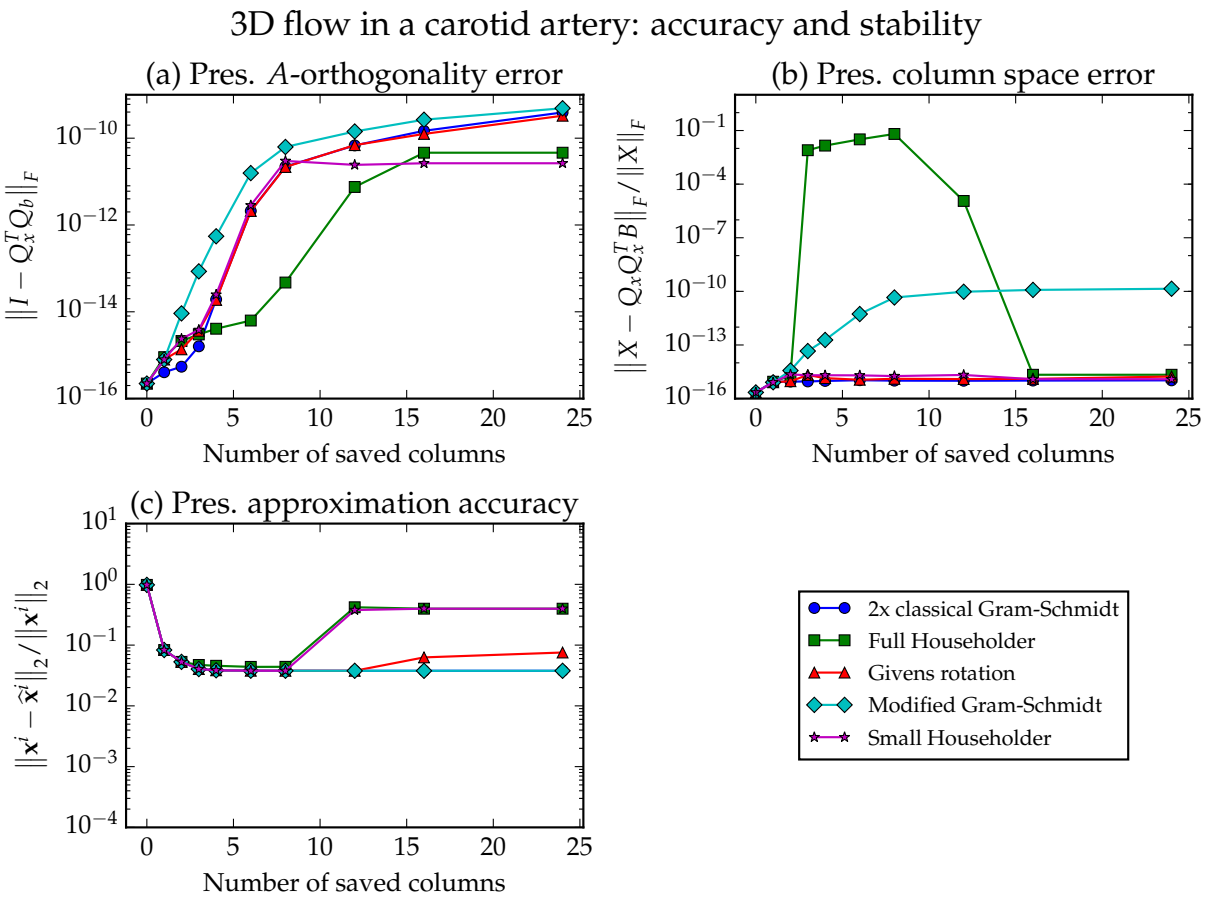

Figure 3.8: Accuracy measures of the reorthogonalization algorithms averaged over 200 time steps in a simulation of blood flow through a carotid artery.

3.3.2 Stability and accuracy

Plots of the accuracy metrics over time for $k = 6$ in Figure 3.3 reveal no surprising differences from the previous two cases. Data points here are shown for every sixth time timestep. The behavior of the average errors over 200 time steps in 3.2 is also consistent at low k. At higher k we see this case strenuously tests the stability of the reorthogonalization algorithms: the Householder algorithms become unstable after more than eight columns are saved, Givens loses some amount of accuracy with sixteen or higher saved columns, and the Gram-Schmidt algorithms retain stability over all values of k tested here.

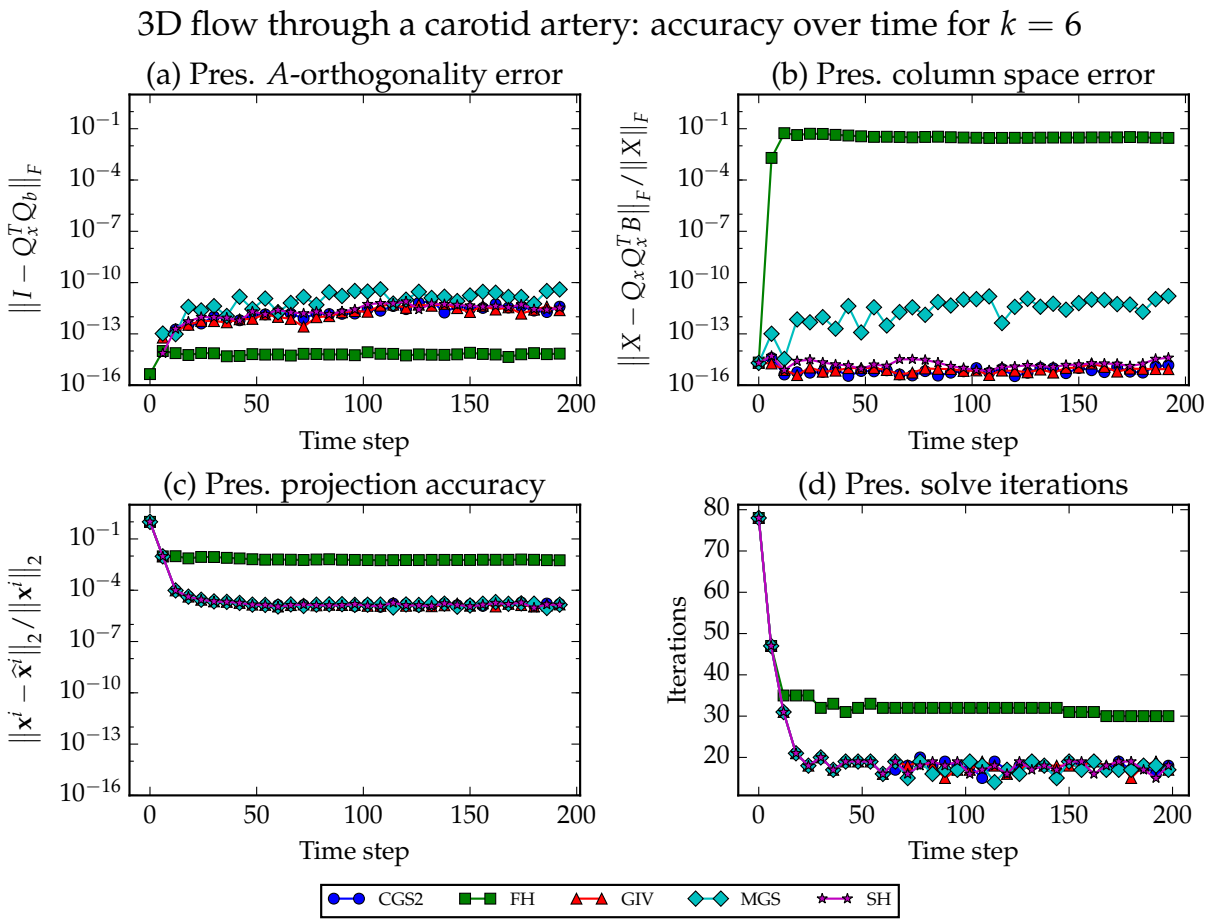

Figure 3.9: Accuracy measures of reorthogonalization algorithms over 200 time steps in a simulation of blood flow through a carotid artery.

CHAPTER 4

ANALYSIS AND CONCLUSION

Retrospective analysis provides insight into why the efficient projection algorithms have limited capability to improve performance. We model the total time as the sum of the time spent in the iterative solver and the time spent updating the projection space, $t_t = t_p + t_s$. We know t_p scales linearly with n and so represent it as $t_p = ckn$ where c is the per column update cost ($\sim$6 for the Givens rotation based updating algorithm), k is the number of columns, and n is the number of entries per column.

We model the cost of the iterative solver as the product of the number of desired accurate digits a and the cost per digit d.[1] For example, if we desire sixteen accurate digits then $t_s = 16d$. Projection has the effect of increasing the number of accurate digits from the outset, allowing us to subtract from t_s the cost of solving for those digits. If the error in the initial guess is $\mathbf{e} = \mathbf{x} - \widehat{\mathbf{x}}$, then the number of accurate digits in $\widehat{\mathbf{x}}$ is close to $-\log(\|\mathbf{e}\|_2)$. This appears to break down if the projected solution is identical to the calculated solution, however, the accuracy of the projection is limited by machine precision. Additionally, error from numerical approximation (e. g. error from not solving $A\mathbf{x} = \mathbf{b}$ exactly or from a poorly conditioned A) also induces some amount of error. Accounting for this intrinsic error ε, the total number of accurate digits is $-\log(\|\mathbf{e}\|_2 + \varepsilon)$. Consequently, the total number of digits to solve for after projection is $a - \log(\|\mathbf{e}\|_2 + \varepsilon)$, which reduces the solve cost to $t_s = d(a + \log(\|\mathbf{e}\|_2 + \varepsilon))$.

The per digit cost of the solver is likely related to the size of the problem, so we let $d = c_2 n^p$. The error scaling of a projection approximation is at least good as the error scaling of an approximation from polynomial extrapolation, which - as we saw in subsection 2.1.1 - scales as $c_3 \Delta t^k$. We can therefore reasonably use this as an estimate for the projection

[1] This is somewhat simplistic as the convergence behavior can change dramatically depending on the solver and preconditioner.

error. Altogether, this gives us a total time cost model of

$$t_t = c_1 nk + c_2 n^p (a - \log(c_3 \Delta t^k + \varepsilon)) \tag{4.1}$$

Setting the derivative with respect to k of (4.1) equal to zero and solving for k results in an expression for the optimal number of columns in the projection space:

$$k_{opt} = \frac{\log\left(\frac{-\varepsilon c_1 n}{c_3 \Delta t c_1 n + c_3 \Delta t c_2 n^p \log \Delta t}\right)}{\log(\Delta t)} = \frac{\log\left(\frac{-\varepsilon c_1}{c_3 (c_1 + c_2 n^{p-1} \log \Delta t)}\right)}{\log(\Delta t)} - 1 \tag{4.2}$$

Equation (4.2) provides some interesting insights, the primary ones being that the optimal k value is most sensitive to the step size Δt and the precision limit ε. This makes sense because high-order approximations are not necessary to maintain accuracy when the step size is very small and because the magnitude ε places a very direct limit on the the usable accuracy of the approximation. The model also shows k_{opt} has dependence on the per digit cost of the solver (and hence on the problem size n), though this grows only logarithmically. The most significant unknown quantity is c_3. We model c_3 as a constant factor here for simplicity. However, it is likely a function of k (e. g. $1/k!$ as in the case of Taylor expansion) and may have a significant damping effect on k_{opt}. A 3D surface plot of the function against Δt and n is shown in Figure 4.1. The model assumes the behavior of the underlying problem remains fairly smooth at arbitrarily large step sizes. If the behavior is not smooth then the error estimate is likely no longer valid. Future refinement of the model may account for reduced smoothness at larger step sizes.

Except for problems with very small solution tolerances and problems with long time steps, the optimal number of saved solutions is apparently often small. Data from the numerical experiments seem to agree with this. It so happens for small k the $O(nk)$ efficient updating algorithm and the $O(nk^2)$ Gram-Schmidt algorithms have nearly identical performance. The efficient updating algorithm therefore likely offers little improvement above Gram-Schmidt for many problems. In 3D problems, where relatively short time steps are needed to maintain stability, this effect is likely particularly relevant. In the above problems, the optimal k was always near six. It would be helpful to determine if this holds generally as this would be a

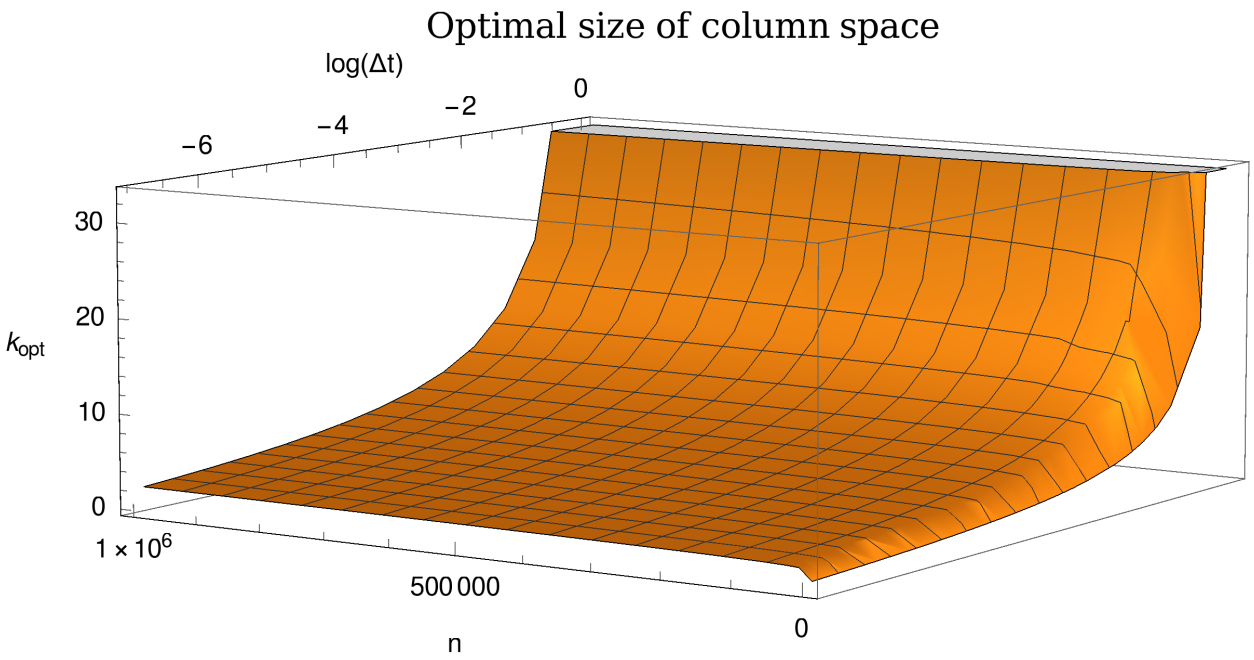

Figure 4.1: 3D plot showing one instance of the surface of optimal k values with varying n and Δt according to (4.2). The non-varying model parameters in this plot are $\varepsilon = 10^{-16}$, $c_1 = 6$, $c_2 = 5$, $c_3 = 5 \times 10^{-7}$, and $p = 3$. Note the strong dependence on Δt.

useful heuristic.

Although the performance benefits are modest, on programs that can run for thousands or millions of core hours even a 5% to 15% improvement may be significant. The projection scheme is applicable to almost any iterative solver for smoothly evolving problems and this efficient updating algorithm is a "low hanging fruit" as coding Algorithm 2.9 is not much more difficult than implementing A-orthogonal Gram-Schmidt (at least now that it has been conveniently formulated). Additionally, the efficient updating algorithm broadens the values of k for which the total time is near the minimum. In applications where the k may be modified by the user, this broader minimum may significantly reduce the time lost due to choosing a larger than optimal number of columns to save. Further, the reduction in required communication may be a desireable quality of the efficient algorithm

In implementation, use of the full Householder matrix as formulated here should be avoided. There does not appear to be a significant performance difference between use of 2×2 Givens matrices and 2×2 Householder matrices, even with the theoretically higher flop count of Householder. The numerical experiments suggest, for certain situations at

least, the Givens based reorthogonalization algorithm may have slightly better accuracy and stability.

4.1 Summary

We have developed efficient algorithms for reorthogonalizing an A-orthogonal projection space used for approximating an accurate initial guess for an iterative solver after a new solution is added. We find use of full Householder matrices results in poor projection and consequently very poor performance. The performance improvement using the Givens and small Householder algorithms is likely highly dependent on the step size and solver tolerance, among other factors. One 3D problem tested displayed run time reductions of $\sim$15% while others showed reductions closer to 5%. When stable, the small Householder and small Givens based updating algorithms did not ever appear to negatively affect performance in the cases tested.

BIBLIOGRAPHY

[1] P. F. Fischer, "Projection techniques for iterative solution of Ax = b with successive right-hand sides," *Computer Methods in Applied Mechanics and Engineering*, vol. 163, no. 1, pp. 193–204, 1998, ISSN: 0045-7825. DOI: 10.1016/S0045-7825(98)00012-7.

[2] L. Grinberg and G. E. Karniadakis, "Extrapolation-Based Acceleration of Iterative Solvers: Application to Simulation of 3D Flows," *Communications in Computational Physics*, vol. 9, no. 3, pp. 607–626, 2011. DOI: 10.4208/cicp.301109.080410s.

[3] H. M. Tufo and P. F. Fischer, "Terascale Spectral Element Algorithms and Implementations," in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, ser. SC '99, Portland, Oregon, USA: ACM, 1999, ISBN: 1-58113-091-0. DOI: 10.1145/331532.331599.

[4] V. Mehrmann and C. Schröder, "Nonlinear eigenvalue and frequency response problems in industrial practice," *Journal of Mathematics in Industry*, vol. 1, no. 1, p. 7, Jul. 2011, ISSN: 2190-5983. DOI: 10.1186/2190-5983-1-7.

[5] V. Peiffer and S. Sherwin, "CFD Challenge: Solutions Using an In-House Spectral Element Solver, NEKTAR," in *ASME 2012 Summer Bioengineering Conference, Parts A and B*, Jun. 2012, pp. 145–146. DOI: 10.1115/SBC2012-80686.

[6] N. Jiang and W. Layton, "Numerical analysis of two ensemble eddy viscosity numerical regularizations of fluid motion," *Numerical Methods for Partial Differential Equations*, vol. 31, no. 3, pp. 630–651, 2015, ISSN: 1098-2426. DOI: 10.1002/num.21908.

[7] J. Szumbarski, "Numerical study of the Yosida method applied to viscous incompressible internal flows with open boundary conditions," *Archives of Mechanics*, vol. 68, pp. 133–160, Jan. 2016. [Online]. Available: http://am.ippt.pan.pl/am/article/view/v68p133.

[8] N. Offermans *et al.*, "On the Strong Scaling of the Spectral Element Solver Nek5000 on Petascale Systems," in *Proceedings of the Exascale Applications and Software Conference 2016*, ser. EASC '16, Stockholm, Sweden: ACM, 2016, 5:1–5:10, ISBN: 978-1-4503-4122-6. DOI: 10.1145/2938615.2938617. arXiv: 1706.02970 [cs.DC].

[9] R. Löhner, "Projective prediction of pressure increments," *Communications in Numerical Methods in Engineering*, vol. 21, no. 4, pp. 201–207, 2005, ISSN: 1099-0887. DOI: 10.1002/cnm.743.

[10] Y. Saad, "Analysis of Augmented Krylov Subspace Methods," *SIAM Journal on Matrix Analysis and Applications*, vol. 18, no. 2, pp. 435–449, 1997. DOI: 10.1137/S0895479895294289.

[11] T. F. Chan and W. L. Wan, "Analysis of Projection Methods for Solving Linear Systems with Multiple Right-Hand Sides," *SIAM Journal on Scientific Computing*, vol. 18, no. 6, pp. 1698–1721, 1997. DOI: 10.1137/S1064827594273067.

[12] S. Sirisup, G. E. Karniadakis, D. Xiu, and I. G. Kevrekidis, "Equation-free/Galerkin-free POD-assisted computation of incompressible flows," *Journal of Computational Physics*, vol. 207, no. 2, pp. 568–587, 2005, ISSN: 0021-9991. DOI: https://doi.org/10.1016/j.jcp.2005.01.024.

[13] R. Markovinović and J. D. Jansen, "Accelerating iterative solution methods using reduced-order models as solution predictors," *International Journal for Numerical Methods in Engineering*, vol. 68, no. 5, pp. 525–541, 2006, ISSN: 1097-0207. DOI: 10.1002/nme.1721.

[14] D. Tromeur-Dervout and Y. Vassilevski, "POD acceleration of fully implicit solver for unsteady nonlinear flows and its application on grid architecture," *Advances in Engineering Software*, vol. 38, no. 5, pp. 301–311, 2007, High Performance Computing in Science and Engineering, ISSN: 0965-9978. DOI: https://doi.org/10.1016/j.advengsoft.2006.08.007.

[15] M. Al Sayed Ali and M. Sadkane, "Improved predictor schemes for large systems of linear ODEs," *Electronic Transactions on Numerical Analysis*, vol. 39, pp. 253–270,

2012, ISSN: 10689613. [Online]. Available: http://www.emis.ams.org/journals/ETNA/vol.39.2012/pp253-270.dir/pp253-270.html.

[16] M. L. Parks, E. de Sturler, G. Mackey, D. D. Johnson, and S. Maiti, "Recycling Krylov Subspaces for Sequences of Linear Systems," *SIAM Journal on Scientific Computing*, vol. 28, no. 5, pp. 1651–1674, 2006. DOI: 10.1137/040607277.

[17] M. E. Kilmer and E. de Sturler, "Recycling subspace information for diffuse optical tomography," *SIAM Journal on Scientific Computing*, vol. 27, no. 6, pp. 2140–2166, 2006. DOI: 10.1137/040610271.

[18] P. E. Gill, G. H. Golub, W. Murray, and M. A. Saunders, "Methods for Modifying Matrix Factorizations," *Mathematics of Computation*, vol. 28, no. 126, pp. 505–505, May 1974, ISSN: 1088-6842. DOI: 10.1090/s0025-5718-1974-0343558-6.

[19] J. W. Daniel, W. B. Gragg, L. Kaufman, and G. W. Stewart, "Reorthogonalization and Stable Algorithms for Updating the Gram-Schmidt QR Factorization," *Mathematics of Computation*, vol. 30, no. 136, p. 772, Oct. 1976. DOI: 10.2307/2005398.

[20] J. Dongarra, J. Bunch, C. Moler, and G. Stewart, "Updating QR & Cholesky Decompositions," in *LINPACK Users' Guide*, Society for Industrial and Applied Mathematics, Jan. 1979, ch. 10, p. 10.23, ISBN: 978-1-61197-181-1. DOI: 10.1137/1.9781611971811.ch10.

[21] A. Buckley, "Algorithm 580: QRUP: A Set of FORTRAN Routines for Updating QR Factorizations [F5]," *ACM Transactions on Mathematical Software*, vol. 7, no. 4, pp. 548–549, Dec. 1981, ISSN: 0098-3500. DOI: 10.1145/355972.355983.

[22] ——, "Remark on 'Algorithm 580: QRUP: A Set of FORTRAN Routines for Updating QR Factorizations'," *ACM Transactions on Mathematical Software*, vol. 8, no. 4, p. 405, Dec. 1982, ISSN: 0098-3500. DOI: 10.1145/356012.356021.

[23] L. Reichel and W. B. Gragg, "Algorithm 686: FORTRAN Subroutines for Updating the QR Decomposition," *ACM Transactions on Mathematical Software*, vol. 16, no. 4, pp. 369–377, Dec. 1990, ISSN: 0098-3500. DOI: 10.1145/98267.98291.

[24] S. Hammarling, N. J. Higham, and C. Lucas, "LAPACK-Style Codes for Pivoted Cholesky and QR Updating," in *Applied Parallel Computing. State of the Art in Scientific Computing: 8th International Workshop, PARA 2006, Umeå, Sweden, June 18-21, 2006, Revised Selected Papers*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 137–146, ISBN: 978-3-540-75755-9. DOI: 10.1007/978-3-540-75755-9_17. MIMS EPrint: 2006.385. [Online]. Available: http://eprints.ma.man.ac.uk/689/1/covered/MIMS_ep2006_385.pdf.

[25] S. Hammarling and C. Lucas, "Updating the QR factorization and the least squares problem," Nov. 14, 2008, ISSN: 1749-9097. MIMS EPrint: 2008.111. [Online]. Available: http://eprints.maths.manchester.ac.uk/1192.

[26] G. Golub and C. Van Loan, *Matrix Computations*, Fourth, ser. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 2013, ISBN: 978-1-42140-794-4.

[27] Å. Björck, "Modified Least Squares Problems," in *Numerical Methods for Least Squares Problems*. 1996, pp. 127–152, ISBN: 978-1-61197-148-4. DOI: 10.1137/1.9781611971484.ch3.

[28] W. Gander, M. J. Gander, and F. Kwok, "Least squares problems," in *Scientific Computing - An Introduction using Maple and MATLAB*. Cham: Springer International Publishing, 2014, pp. 311–320, ISBN: 978-3-319-04325-8. DOI: 10.1007/978-3-319-04325-8_6.

[29] Å. Björck, "Linear least squares problems," in *Numerical Methods in Matrix Computations*. Cham: Springer International Publishing, 2015, pp. 305–311, ISBN: 978-3-319-05089-8. DOI: 10.1007/978-3-319-05089-8_2.

[30] M. Galassi *et al.*, "Linear Algebra," in *GNU Scientific Library Release 2.4 Reference Manual*, Jul. 14, 2017, ch. 14, p. 131. [Online]. Available: https://www.gnu.org/software/gsl/doc/latex/gsl-ref.pdf.

[31] E. Jones, T. Oliphant, P. Peterson, *et al.*, *SciPy v1.0.0 Reference Guide*, Oct. 25, 2017. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.qr_insert.html.

[32] J. Hajek, *qrupdate*, version 1.1.2. [Online]. Available: https://sourceforge.net/projects/qrupdate (visited on 11/19/2017).

[33] J. W. Eaton, D. Bateman, S. Hauberg, and R. Wehbring, "Matrix factorizations," in *GNU Octave version 4.2.0 manual: a high-level interactive language for numerical computations*, 2016. [Online]. Available: https://www.gnu.org/software/octave/doc/interpreter/Matrix-Factorizations.html#XREFqrinsert.

[34] S. N. Afriat, "Orthogonal and oblique projectors and the characteristics of pairs of vector spaces," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 53, no. 4, pp. 800–816, 1957. DOI: 10.1017/S0305004100032916.

[35] L. Rebollo-Neira, "Constructive updating/downdating of oblique projectors: a generalization of the Gram-Schmidt process," *Journal of Physics A: Mathematical and Theoretical*, vol. 40, no. 24, p. 6381, 2007. DOI: 10.1088/1751-8113/40/24/007.

[36] S. Kayalar and H. L. Weinert, "Oblique projections: Formulas, algorithms, and error bounds," *Mathematics of Control, Signals and Systems*, vol. 2, no. 1, pp. 33–45, Mar. 1989, ISSN: 1435-568X. DOI: 10.1007/BF02551360.

[37] S. J. Thomas, "A block algorithm for orthogonalization in elliptic norms," in *Parallel Processing: CONPAR 92—VAPP V: Second Joint International Conference on Vector and Parallel Processing Lyon, France, September 1–4, 1992 Proceedings*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 379–385, ISBN: 978-3-540-47306-0. DOI: 10.1007/3-540-55895-0_434.

[38] G. W. Stewart, "On the numerical analysis of oblique projectors," *SIAM Journal on Matrix Analysis and Applications*, vol. 32, no. 1, pp. 309–348, Mar. 2011, ISSN: 0895-4798. DOI: 10.1137/100792093.

[39] M. Rozložník, M. Tůma, A. Smoktunowicz, and J. Kopal, "Numerical stability of orthogonalization methods with a non-standard inner product," *BIT Numerical Mathematics*, vol. 52, no. 4, pp. 1035–1058, Dec. 2012, ISSN: 1572-9125. DOI: 10.1007/s10543-012-0398-9.

[40] B. R. Lowery and J. Langou, "Stability Analysis of QR factorization in an Oblique Inner Product," Jan. 2014. arXiv: 1401.5171 [math.NA].

[41] A. Imakura and Y. Yamamoto, "Efficient implementations of the modified Gram-Schmidt orthogonalization with a non-standard inner product," Mar. 2017. arXiv: 1703.10440 [math.NA].

[42] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Second.: Society for Industrial and Applied Mathematics, 2003, ISBN: 978-0-89871-800-3. DOI: 10.1137/1.9780898718003. [Online]. Available: http://www-users.cs.umn.edu/~saad/IterMethBook_2ndEd.pdf.

[43] A. D. Okano, "An Oblique QR Factorization of Tall and Skinny Matrices in Julia," Master's thesis, 2015, ISBN: 978-1-339-06575-5. [Online]. Available: https://search.proquest.com/docview/1724033852.

[44] J. Q. Zhao, "S-Orthogonal QR Decomposition Algorithms on Multicore Systems," English, Master's thesis, 2013, ISBN: 978-1-30379-523-7. [Online]. Available: https://search.proquest.com/docview/1526494869.

[45] A. Ruhe, "Numerical Aspects of Gram-Schmidt Orthogonalization of Vectors," *Linear Algebra and its Applications*, vol. 52-53, no. Supplement C, pp. 591–601, 1983, ISSN: 0024-3795. DOI: 10.1016/0024-3795(83)80037-8.

[46] 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. 2005, ISSN: 0945-3245. DOI: 10.1007/s00211-005-0615-4.

[47] G. W. Stewart, "When Is Twice Enough? (The Oblique Case)," in *Householder Symposium XVIII on Numerical Linear Algebra*, Tahoe City, California, Jun. 2011, p. 220. [Online]. Available: http://www.mims.manchester.ac.uk/~higham/conferences/householder/HH11_Abstracts.pdf#page=232.

[48] D. Bindel, J. Demmel, W. Kahan, and O. Marques, "On Computing Givens Rotations Reliably and Efficiently," *ACM Transactions on Mathematical Software*, vol. 28, no. 2, pp. 206–238, Jun. 2002, ISSN: 0098-3500. DOI: 10.1145/567806.567809.

[49] M. Harris. (Apr. 29, 2014). CUDA Pro Tip: Fast and Robust Computation of Givens Rotations, NVIDIA, [Online]. Available: https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-fast-robust-computation-givens-rotations.