Building rank-revealing factorizations with randomization

by

Nathan D. Heavner

B.S., Clemson University, 2014

M.S., University of Colorado, 2017

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Applied Mathematics

2019

This thesis entitled:
Building rank-revealing factorizations with randomization
written by Nathan D. Heavner
has been approved for the Department of Applied Mathematics

_____

Prof. Per-Gunnar Martinsson

_____

Prof. Stephen Becker

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Heavner, Nathan D. (Ph.D., Applied Mathematics)

Building rank-revealing factorizations with randomization

Thesis directed by Prof. Per-Gunnar Martinsson

This thesis describes a set of randomized algorithms for computing rank revealing factorizations of matrices. These algorithms are designed specifically to minimize the amount of data movement required, which is essential to high practical performance on modern computing hardware. The work presented builds on existing randomized algorithms for computing low-rank approximations to matrices, but essentially extends the range of applicability of these methods by allowing for the efficient decomposition of matrices of any numerical rank, including full rank matrices. In contrast, existing methods worked well only when the numerical rank was substantially smaller than the dimensions of the matrix.

The thesis describes algorithms for computing two of the most popular rank-revealing matrix decompositions: the column pivoted QR (CPQR) decomposition, and the so called UTV decomposition that factors a given matrix $\mathbf{A}$ as $\mathbf{A} = \mathbf{UTV}^*$, where $\mathbf{U}$ and $\mathbf{V}$ have orthonormal columns and $\mathbf{T}$ is triangular. For each algorithm, the thesis presents algorithms that are tailored for different computing environments, including multicore shared memory processors, GPUs, distributed memory machines, and matrices that are stored on hard drives ("out of core").

The first chapter of the thesis consists of an introduction that provides context, reviews previous work in the field, and summarizes the key contributions. Beside the introduction, the thesis contains six additional chapters:

Chapter 2 introduces a fully blocked algorithm HQRRP for computing a QR factorization with column pivoting. The key to the full blocking of the algorithm lies in using randomized projections to create a low dimensional sketch of the data, where multiple good pivot columns may be cheaply computed. Numerical experiments show that HQRRP is several times faster than the classical algorithm for computing a column pivoted QR on a multicore machine, and the acceleration factor increases with the number of cores.

Chapter 3 introduces randUTV, a randomized algorithm for computing a rank-revealing factorization

of the form $\mathbf{A} = \mathbf{UTV}^*$, where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal and $\mathbf{T}$ is upper triangular. RandUTV uses randomized methods to efficiently build $\mathbf{U}$ and $\mathbf{V}$ as approximations of the column and row spaces of $\mathbf{A}$. The result is an algorithm that reveals rank nearly as well as the SVD and costs at most as much as a column pivoted QR.

Chapter 4 provides optimized implementations for shared and distributed memory architectures. For shared memory, we show that formulating randUTV as an algorithm-by-blocks increases its efficiency in parallel. The fifth chapter implements randUTV on the GPU and augments the algorithm with an oversampling technique to further increase the low rank approximation properties of the resulting factorization. Chapter 6 implements both randUTV and HQRRP for use with matrices stored out of core. It is shown that reorganizing HQRRP as a left-looking algorithm to reduce the number of writes to the drive is in the tested cases necessary for the scalability of the algorithm when using spinning disk storage. Finally, chapter 7 discusses an alternative use for randUTV as a nuclear norm estimator and measures the acceleration gained from trimming down the algorithm when only singular value estimates are required.

Acknowledgements

First, many thanks are due to Professor Per-Gunnar Martinsson. His advisement is responsible for many of the opportunities I have been given to engage with fascinating problems and people in the last few years. Perhaps more importantly, his insight and advice have spurred vast growth in myself, intellectually and professionally. I am grateful to have had an advisor whom I view with great respect both on a personal level and as a mathematician.

Thanks also to Professor Stephen Becker for providing a solid understanding of functional analysis with which to explore the world of undiscovered mathematics; to Professor Gregory Beylkin for teaching me the foundations of numerical analysis for linear algebra; and to Doctor Christian Ketelsen for stoking my interest in numerical linear algebra and data science through various courses and conversations. A special thanks is due to Professor Gregorio Quintanta-Ortí for his invaluable intellectual and technical contributions to this thesis, and for his considerable patience in communicating technical ideas through long email chains.

I am grateful for my fellow graduate students in the APPM department. Sharing the experience of pursuing a doctorate with such inspiring people has made the joys sweeter and the trials easier to endure.

I must also thank my parents, whose unwavering support for and interest in my academic endeavors has propelled me further than I likely would have gone otherwise. As long as you keep asking, I will keep trying to explain what exactly I do.

Contents

Chapter

Chapter 1

Introduction

Most traditional rank-revealing factorization algorithms were designed when communication costs for scientific computing were negligible compared to the cost of floating point operations. As such, they are not easily optimized for many modern scientific computing environments, which often rely heavily on parallel computations. This thesis presents algorithms for computing rank-revealing factorizations which are designed to operate efficiently in parallel environments. In this chapter, we present an overview of the contributions of the thesis. We begin with context, discussing the problem we intend to solve and some distinct challenges we work to address. A summary is then provided of previous work in the field as well as recent innovations that have inspired our contributions. We finally provide brief descriptions of the methods developed in the thesis, leaving thorough discussion and analysis for the following chapters.

1.1 Context

1.1.1 Rank-revealing factorizations

Given a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, it is often desirable to compute a matrix factorization

$$\underset{m \times n}{\mathbf{A}} = \underset{m \times m}{\mathbf{U}} \quad \underset{m \times n}{\mathbf{T}} \quad \underset{n \times n}{\mathbf{V}^*}, \tag{1.1}$$

where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal and $\mathbf{T}$ is triangular, such that the factorization is *rank-revealing*. We will call a matrix rank-revealing if, considering the partitioning of $\mathbf{T}$

$$\mathbf{T} \rightarrow \begin{bmatrix} \mathbf{T}_{11} & \mathbf{T}_{12} \\ \mathbf{T}_{21} & \mathbf{T}_{22} \end{bmatrix},$$

where $\mathbf{T}_{11}$ is $k \times k$ for any $1 \leq k \leq \min(m, n)$, we have that

- $\sigma_k(\mathbf{A}) \approx \sigma_{\min}(\mathbf{T}_{11})$ and
- $\sigma_{k+1}(\mathbf{A}) \approx \sigma_{\max}(\mathbf{T}_{22})$.

This informal definition is a slight generalization of the usual definitions of rank revealing decompositions that appear in the literature, minor variations of which appear in, *e.g.* [27, 114, 24, 26]. The definition given above is useful because it allows us to compare the quality of different rank-revealing factorizations. Given two decompositions $\mathbf{A} = \mathbf{UTV}^*$ and $\mathbf{A} = \mathbf{WSX}^*$, we will say that the first factorization *reveals rank better* than the second if

- $\|\sigma_k(\mathbf{A}) - \sigma_{\min}(\mathbf{T}_{11})\| < \|\sigma_k(\mathbf{A}) - \sigma_{\min}(\mathbf{S}_{11})\|$ and
- $\|\sigma_{k+1}(\mathbf{A}) - \sigma_{\max}(\mathbf{T}_{22})\| < \|\sigma_{k+1}(\mathbf{A}) - \sigma_{\max}(\mathbf{S}_{22})\|$.

Rank revealing factorizations are useful in solving problems such as least squares approximation [26, 25, 54, 80, 17, 56], rank estimation [24, 118, 116], low rank approximation [83, 42, 77, 31, 9], and subspace tracking [16, 114], among others.

It is well known that the singular value decomposition is the optimal rank-revealing factorization in the sense that for an SVD factorization $\mathbf{A} = \mathbf{UDV}^*$, by definition $\sigma_k(\mathbf{A}) = \sigma_{\min}(\mathbf{D}_{11})$ and $\sigma_{k+1}(\mathbf{A}) = \sigma_{\max}(\mathbf{D}_{22})$. This decomposition is expensive to compute, though, and the traditional algorithms do not easily parallelize well. Other options have been developed with varying strengths and weaknesses; this research is summarized in Section 1.2.

Another well known rank-revealing factorization is the column pivoted QR factorization. This decomposition does not reveal rank as well as the SVD, but unlike the SVD it can easily be used to compute

partial factorizations. In traditional computing environments, it was also substantially faster than the SVD due to requiring many fewer floating point operations.

1.1.2 Communication vs flops

Recent trends in technology development have altered goals in the design of algorithms for high-performance computing. Classical algorithms focused on decreasing the floating point operation (flop) count — and for good reason. In 1988, one flop took six times as long as the memory latency [32, Ch. 5], so performing floating point operations was the bottleneck in numerical computations. In recent years, however, matters have changed dramatically.

For example, from 1988 to 2004, the floating-point performance of microprocessors improved by 59% per year, whereas the bandwidth of DRAM chips increased 25% per year and DRAM latency increased by only 5.5% per year. The disparity in growth between floating-point performance and bandwidth has led to the development of processors with multilevel cache systems, which require locality and temporality of data to achieve high efficiency. Furthermore, the relatively slow improvement of DRAM latency means that as of 2004, one hundred flops could be performed in the time it took to access memory once [32, Ch. 5].

This reversal in efficiency for floating point operations vs latency and bandwidth means that even in the case of a single processor, latency or bandwidth restrictions can often be the cause of bottlenecks in numerical computations. The increasing demand for parallel computing architectures (caused by the slowing in growth for even flop performance in recent years [53]) further heightens the need for good data locality and temporality in applications. Thus, any reference to "communication cost" in this manuscript refers to *any cost incurred by the transfer of data*, whether between external storage devices and RAM, two processors working in parallel, or different levels of cache.

Many traditional linear algebra algorithms, with their primary focus of minimizing flops, do not take full advantage of the available processing power of most modern computers. Significant acceleration of traditional algorithms can be achieved by implementing revisions which reduce communication. This fact is demonstrated in Figure 1.1. Observe in particular that this figure includes one algorithm that is significantly faster than another in spite of incurring a higher asymptotic flop count than the competition.

1.2 Previous work

In this section, we briefly review the previous work in rank-revealing matrix factorizations. Section 1.2.1 reviews two of the most commonly used decompositions which reveals rank, and Section 1.2.2 discusses a lesser-known but highly relevant factorization. Finally, Section 1.2.3 briefly summarizes the development of randomization as a key tool in linear algebra problem-solving.

1.2.1 Rank-revealing factorizations.

The singular value decomposition (SVD) is one of the two most widely used rank-revealing factorizations. It provides a wealth of information about the input matrix, including orthonormal bases for its fundamental subspaces, singular values and vectors, and an easily invertible form. Most notably, by the celebrated Eckart-Young-Mirsky Theorem [45, 98], it yields theoretically optimal low rank approximations. Specifically, for an SVD $\mathbf{UDV}^*$ of $\mathbf{A}$, we have that

$$\|\mathbf{A} - \mathbf{U}(:, 1:k)\mathbf{D}(1:k, 1:k)(\mathbf{V}(1:k, :))^*\| = \min\{\|\mathbf{A} - \mathbf{B}\| : \mathbf{B} \text{ has rank } k\}.$$

This high quality information comes at a price, however. The SVD is quite expensive to compute, often prohibitively so.

Furthermore, the traditional algorithms for computing an SVD are challenging to implement well in parallel computing environments. The computation is usually done in two stages. In the first stage, the input matrix is reduced to bidiagonal form with a sequence of orthogonal transformations which must be applied to the matrix serially. In the second stage, the matrix is fully reduced to diagonality with either a variant of the iterative QR algorithm or the divide-and-conquer method, a recursive technique that uses the QR algorithm at the base level [57, 122]. Available options for either stage are not amenable to parallelization.

The other widely used rank-revealing decomposition, the rank-revealing QR factorization (RRQR), was developed as a response to the expense of computing an SVD. Appropriately, then, these are much faster to compute than an SVD, but they do not reveal rank as well. Many algorithms for calculating an RRQR have been developed [21, 55, 61, 51, 24], but the column pivoted QR (CPQR) of [21] has remained the most popular standard for use in software. CPQR usually produces a factorization that reveals rank reasonably

well, but it can in certain known cases give wildly suboptimal results [76]. It is also inherently limited in its parallel performance in that the selection of pivot columns must be done one at a time, preventing the algorithm from being fully blocked (see Section 1.3.1 for discussion on the value of blocked algorithms). Quintana-Ortí *et al.* developed a partial work-around in [107], but the remaining inefficiency can be observed in Figure 1.1. Note that the algorithms used for `dgeqrf` (unpivoted QR) and `dgeqp3` (CPQR) have the same asymptotic flop count (but `dgeqp3` requires more flops overall). The main difference between the two causing the difference in execution time, at least in the limit as n grows large, is that the first is fully blocked while the second is not. Thus, `dgeqp3` spends much more time in communication-related tasks.

1.2.2 Rank-revealing UTV factorizations.

Given an $m \times n$ matrix $\mathbf{A}$, the so called "UTV decomposition" [116, p. 400] takes the form

$$\underset{m \times n}{\mathbf{A}} = \underset{m \times m}{\mathbf{U}} \quad \underset{m \times n}{\mathbf{T}} \quad \underset{n \times n}{\mathbf{V}^*}, \tag{1.2}$$

where $\mathbf{U}$ and $\mathbf{V}$ are unitary matrices and $\mathbf{T}$ is a trapezoidal matrix. (In this work, we will typically assume that $m \geq n$, and seek a factorization with an upper triangular matrix $\mathbf{T}$, but the techniques can with minor modifications be applied to other cases as well.)

Because of the relative flexibility of the matrices $\mathbf{U}, \mathbf{T}, \mathbf{V}$ compared to QR and SVD, the UTV decomposition is presented in the literature as a "compromise" between the SVD and RRQR, featuring desirable properties of both. Standard UTV computation algorithms are faster than SVD and slower than RRQR, while less accurate than SVD and more accurate than RRQR in terms of rank-revealing properties. Importantly, there are efficient methods for updating a UTV factorization [114, 115, 6], which is a significant advantage over an SVD. UTV also gives immediate access to estimations of orthonormal bases for all fundamental subspaces [50], unlike the RRQR.

Most algorithms for computing rank revealing UTVs begin with a triangular factorization and then use a post-processing step that improves the rank-revealing properties of the middle matrix [70]. See [115] and [49] for representatives of this idea for high and low rank matrices, respectively. A survey of these

algorithms and their variations, along with discussions of efficient updating and downdating, can be found in [50].

A randomized algorithm for computing a UTV was introduced by Demmel, Dumitriu, and Holtz in [35]; we refer to this technique as `DDHUTV` in reference to the authors of [35]. The algorithm is simple and fast. Since it does not incorporate information from the row space of $\mathbf{A}$, though, `DDHUTV` leaves room for improvement in the quality of its rank revelation. This is a design choice rather than a defect, since the algorithm was designed to be used iteratively as part of a fast eigensolver. The `DDHUTV` algorithm represents, an early use of the kind of randomized sampling techniques which are further explored in this thesis.

1.2.3 Randomization in matrix factorizations.

Interest in randomized matrix factorization algorithms has grown rapidly in recent years. While minor uses of randomization, such as choosing a random starting vector for iterative solvers, have been common practice for some time, investigation of randomization for key roles in matrix algorithms began only recently. See [130, 85, 69] for a survey of various applications of randomization in matrix approximation.

Utilization of randomization in matrix approximation has been enabled by some key developments in subfields of random matrix theory. A particularly relevant development began in the study of randomized embeddings when Johnson and Lindenstrauss published their celebrated paper [75], in which they showed that random embeddings approximately preserve Euclidean geometry. Their result furthermore says that the dimension of the embedding space required for a good approximation is dependent on the number of points in the dataset rather than the original dimension of those points. Even better, this dependence itself is quite weak – on the order of the *logarithm* of the number of points in the dataset. This result sparked the idea that some previously intractable problems may become computationally feasible by first projecting the data into a space of lower dimension and then solving the problem with the low-dimensional sketch of the data. The algorithms discussed herein draw from this approach, using low-dimensional sketches to efficiently determine row space information before returning to the original (high-dimensional) matrix to compute a full factorization.

Our work here uses randomized sampling to create a matrix approximation with reduced dimension. To our knowledge, this idea first appeared in [101], where Papadimitriou et al. suggested projecting the input matrix onto a random subspace, then compressing the original matrix onto the projection. Though this method provided better accuracy than a competing randomized column selection method with similar error bounds [52], it was less computationally efficient.

Sarlós [112] then argued that the cost of dimension reduction could be reduced by using structured random maps, such as those introduced by Ailon and Chazelle in [1]. Though the initial work by Sarlós did not produce algorithms which improved on classical methods, Woolfe et al. improved on the asymptotic computational cost and applied the resulting refined techniques to scientific computation problems in [131]. See [82, 83] for related work.

The algorithms presented here rely heavily on the work by Martinsson-Rokhlin-Tygert in [90]. Their paper shows that using a Gaussian transform matrix for dimension reduction works better than had been previously realized. In particular, they show that oversampling in the range sampling step has a dramatic effect on the quality of the resulting low-dimensional approximation, in both theory and practice. They also demonstrate how dimension reduction through sampling can be used to compute an interpolative decomposition of an input matrix, a problem closely associated with choosing a good pivoting matrix in a pivoted QR decomposition.

A final relevant development is published in [111], where Rokhlin-Szlam-Tygert make use of a power iteration in combination with dimension reduction to improve the quality of the low-dimensional approximation with not too much increase in computational cost. This idea led to an efficient method for large-scale PCA, detailed in [68]. A power iteration modified to prevent loss of accuracy due accumulation in roundoff error is given in [92, 69].

The recent interest in randomized techniques has led to the appearance of a number of survey papers, including [69, 85, 130, 41]. A particularly accessible introduction can be found in [87] (with an arxiv version at [87]).

1.3 Reducing communication in matrix computations

In this section, we discuss two techniques of algorithm design that reduce communication costs in linear algebra algorithms. Section 1.3.1 reviews blocked algorithms, while Section 1.3.2 discusses a more recently developed method called algorithms-by-blocks.

1.3.1 Blocked algorithms

One way to decrease communication costs in linear algebra algorithms is to make use of *blocking*. Many algorithms in this field, especially ones relating to computing matrix factorizations like QR, process single columns of the input matrix at a time. With this idea, most of the operations are cast in terms of matrix-vector operations and are therefore carried out in implementation by Level 2 Basic Linear Algebra Subprogram (BLAS) [81, 19] routines. However, certain algorithms are amenable to the idea that several columns of the input matrix, or one "block," may be processed at once. In this case, the operations that were matrix-vector in the single column case become matrix-matrix operations, implemented by Level 3 BLAS routines. It is well known that Level 3 routines enable even single core processors to achieve far more flops per second than Level 2 or Level 1 due largely to their much higher ratio of flops to memory accesses [38, Section 1]. On multicore processors and GPUs, the acceleration achieved by Level 3 routines, in particular matrix-matrix multiplication, is even more pronounced. Thus, blocked algorithms are generally faster than their unblocked counterparts in practice, even though the asymptotic flop counts of the two are the same, and despite the fact that sometimes the actual flop count of the blocked algorithm is actually higher.

1.3.2 Algorithms-by-blocks

Even blocked algorithms have limitations in communication efficiency, particularly in severely communication restricted environments. The issue of concern is bottlenecks in data accessibility. For example, even the fully blocked unpivoted QR factorization has a step that can only be efficiently executed by a single processor (the factorization of the block of b columns). Whenever this step is executed, it leaves the other cores idle, wasting processor power. When there are only a few cores, there is no great loss, but the time

lost increases with the number of processors.

A solution, called algorithms-by-blocks, is presented in [106, 23]. Rather than aggregating operations into Level-3 BLAS routines by processing multiple columns at once, an algorithms-by-blocks raises the granularity of the data itself, then builds the desired algorithm as usual. Anything treated as an "element" of the matrix is actually a submatrix, so each operation acts on two matrices and will thus naturally be expressed as Level-3 BLAS when possible. The key benefit, though, lies in the elimination of synchronization points enabled by this philosophy. When the inputs to operations consist only of small submatrices, there is greater freedom in scheduling the operations. Thus, the load on multicore systems is far more balanced. In practice, algorithms-by-blocks is implemented with a runtime system which keeps track of data dependencies and schedules each operation to make efficient use of system resources.

1.4 Randomized SVD

One key algorithm developed in this thesis is an extension of the randomized SVD developed in [68]. In Section 1.4.1, we give an overview of the algorithm, and in Section 1.4.2 we consider how the techniques of the randomized SVD may be used to build a rank-revealing full factorization.

1.4.1 Overview of the algorithm

Given an $m \times n$ input matrix $\mathbf{A}$ and a target rank $k \ll \min(m, n)$, the randomized SVD (RSVD) of [68] efficiently computes an approximate factorization

$$\begin{array}{ccccccc} \mathbf{A} & \approx & \mathbf{U} & & \mathbf{D} & & \mathbf{V}^*, \\ m \times n & & m \times k & & k \times k & & k \times n \end{array} \tag{1.3}$$

where $\mathbf{U}$ and $\mathbf{V}$ are orthonormal and $\mathbf{D}$ is diagonal. The algorithm may be summarized with just a few steps:

1. **Compute an $m \times k$ orthonormal matrix $\mathbf{Q}$ such that $\mathbf{Q}\mathbf{Q}^*\mathbf{A} \approx \mathbf{A}$.** Since $\mathbf{Q}\mathbf{Q}^*$ is an orthogonal projection, this item may also be phrased as, "compute an approximate orthonormal basis for the column space of $\mathbf{A}$." This is the step requiring randomized projections, and the precise algorithm is given below.

2. **Compute $\mathbf{B} = \mathbf{Q}^*\mathbf{A}$.** This multiplication is relatively inexpensive at $\mathcal{O}(mnk)$.

3. **Compute the SVD of $\mathbf{B}$ to obtain $\mathbf{B} = \hat{\mathbf{U}}\mathbf{D}\mathbf{V}^*$.** Since $\mathbf{B}$ is short and wide, this step is relatively cheap as well. We may accomplish it by performing an unpivoted QR factorization ($\mathcal{O}(nk^2)$) followed by an SVD for a $k \times k$ matrix ($\mathcal{O}(k^3)$) and a matrix multiplication to update the right singular vectors ($\mathcal{O}(nk^2)$).

4. **Compute $\mathbf{U} = \mathbf{Q}\hat{\mathbf{U}}$.** This step completes the estimate of the left singular values at a cost of $\mathcal{O}(mk^2)$.

Observe that once these steps are completed, we have

$$\mathbf{A} \approx \mathbf{Q}\mathbf{Q}^*\mathbf{A} = \mathbf{Q}\mathbf{B} = (\mathbf{Q}\hat{\mathbf{U}})\mathbf{D}\mathbf{V}^* = \mathbf{U}\mathbf{D}\mathbf{V}^*.$$

Step 1 contains the key to the novelty and efficiency of the algorithm. It was noted in [90] that if k random samples of $\mathbf{A}$ are collected, the span of these samples is a good approximation for the column space of $\mathbf{A}$. Then $\mathbf{Q}$ may be obtained by orthonormalizing these random samples. This observation led to the `range_finder_basic` described in Algorithm 1.

Algorithm 1 $\mathbf{Q} =$ `range_finder_basic`$(\mathbf{A}, k)$

% `Generate_norm_rand_iid`(m, n) returns a matrix of dimensions $m \times n$ whose entries are independently
% drawn from the standard normal distribution.
% $n(\mathbf{A})$ returns the number of columns of matrix $\mathbf{A}$.

1: $\mathbf{G} =$ `Generate_norm_rand_iid`$(n(\mathbf{A}), k)$
2: $\mathbf{Y} = \mathbf{A}\mathbf{G}$
3: $[\mathbf{Q}, \sim] =$ `qr`$(\mathbf{Y})$

Several improvements to `range_finder_basic` drastically improve the reliability and accuracy of the factorization produced by RSVD. These improvements are discussed below and incorporated into Algorithm 2.

a. **Oversampling.** Based on properties of the random matrix $\mathbf{G}$ in `range_finder_basic`, it can be shown that the approximation provided by $\mathbf{Q}$ can be improved by adding a few, say p, extra samples of the column space. Doing so also dramatically decreases the probability of deviating

substantially from the known error bound, making the use of randomization in the algorithm quite safe. For details on the error bounds, see [68]. In practice, choosing p to be 5 or 10 is sufficient, adding very little to the computational cost.

b. **Power iteration.** The approximate basis provided by $\mathbf{Q}$ from can be quite good but suffers in the case of slow decay in the singular values of $\mathbf{A}$. To remedy this, the sampling matrix can be altered from $\mathbf{A}$ to $(\mathbf{A}\mathbf{A}^*)^q\mathbf{A}$ for some small integer q. This has a similar effect to a power iteration scheme for computing eigenvalues, aligning the samples more closely with singular vectors corresponding to larger singular values of $\mathbf{A}$.

c. **Orthonormalization.** In the case that item b above is used, loss of accuracy due to accumulation in roundoff error becomes a concern. Specifically, the information in the samples from any singular value of $\mathbf{A}$ less than $\epsilon_{\text{machine}}^{2q+1}$ will be lost in floating-point arithmetic. To avoid this loss, we perform the computation $\mathbf{Y} = (\mathbf{A}\mathbf{A}^*)^q\mathbf{A}\mathbf{G}$ incrementally, orthonormalizing the columns of the intermediate matrix in between each application of $\mathbf{A}$ or $\mathbf{A}^*$.

Algorithm 2 $\mathbf{Q} = \texttt{range\_finder}(\mathbf{A}, k)$

% `Generate_norm_rand_iid`(m, n) returns a matrix of dimensions $m \times n$ whose entries are independently
% drawn from the standard normal distribution.
% $n(\mathbf{A})$ returns the number of columns of matrix $\mathbf{A}$.
$\mathbf{G}$ =`Generate_norm_rand_iid`$(n(\mathbf{A}), k + p)$
$\mathbf{Y} = \mathbf{A}\mathbf{G}$
$[\mathbf{Q}, \sim] = \texttt{qr}(\mathbf{Y})$
for $i = 1$ to q **do**
  $\mathbf{Y} = \mathbf{A}^*\mathbf{Q}$
  $[\mathbf{Q}, \sim] = \texttt{qr}(\mathbf{Y})$
  $\mathbf{Y} = \mathbf{A}\mathbf{Q}$
  $[\mathbf{Q}, \sim] = \texttt{qr}(\mathbf{Y})$
end for

1.4.2 Extending the RSVD to compute full factorizations

The RSVD has found popularity in ~~very~~ recent years, probably due largely to its efficiency in parallel environments. Nevertheless, the primary value of the method is seen in the case when the target rank

$k \ll \min(m,n)$. If this is not the case, then the SVD step requires a computation on a matrix roughly the same size as the original, so the multiplication of $\mathbf{A}$ by its approximate column space is just extra work. A primary contribution of this thesis is therefore to present a full matrix factorization which uses the concepts of randomized range finding from RSVD but which retains efficiency for *any* choice of k (*e.g.* $k = \min(m,n)$).

To build a full factorization, we use the `range_finder_basic` of Algorithm 2 to efficiently compute subspace approximations in small blocks at a time. Once we have approximations of the column and row space of the "unprocessed" block of $\mathbf{A}$, we use Householder reflectors to rotate the "mass" of the block to the upper left and satisfy some typical sparsity requirements. The use of Householder reflectors is an important modification that ensures the aggregate cost of applying these rotations is $\mathcal{O}(n^3)$. The result is a full matrix rank revealing factorization $\mathbf{A} = \mathbf{UTV}^*$ that is fully blocked and casts most of its flops in terms of matrix-matrix multiplies, making the algorithm efficient in parallel environments.

1.5 Contributions of this thesis

The scientific contributions of this thesis center on two new communication-efficient algorithms: `HQRRP` and `randUTV`. The specific strengths of each are discussed in the following sections and are conveniently summarized by Figure 1.2. Particularly, `HQRRP` provides significant acceleration over the traditional CPQR while resulting in similar approximation accuracies. `randUTV` yields factorizations that reveal rank nearly as well as the SVD at speeds faster than CPQR.[1] The work on these two algorithms is organized into 6 chapters, as follows:

Ch. 2: *Householder QR Factorization With Randomization for Column Pivoting (HQRRP)*

Ch. 3: *randUTV: A blocked randomized algorithm for computing a rank-revealing UTV factorization*

Ch. 4: *Efficient algorithms for computing a rank-revealing UTV factorization on parallel computing architectures*

Ch. 5: *Efficient algorithms for computing rank-revealing factorizations on a GPU*

[1] In almost all tested computing environments, for $n \gtrsim 5000$.

Ch. 6: *Computing rank-revealing factorizations of matrices stored out-of-core*

Ch. 7: *Efficient nuclear norm approximation via the randomized UTV algorithm*

In the remainder of this introduction, we briefly describe the main ideas behind each of the six manuscripts.

1.5.1 "Householder QR Factorization With Randomization for Column Pivoting (HQRRP)"

The first algorithmic contribution, `HQRRP` (Householder QR factorization with Randomized Pivoting), computes a rank-revealing QR factorization of an input matrix $\mathbf{A}$. The algorithm is a solution to the communication limitation of traditional CPQR discussed in Section 1.5.1. `HQRRP` finds multiple, say b, pivot columns at a time by projecting columns of the input matrix into a low dimensional space and (cheaply) finding b pivot columns there. In practice, this is accomplished by drawing a random Gaussian matrix $\mathbf{G} \in \mathbb{R}^{b \times m}$ composed of i.i.d. entries that follow the standard normal distribution and computing the projection $\mathbf{Y} = \mathbf{GA}$. Since random linear projections approximately preserve distances between vectors by the Johnson-Lindenstrauss theorem, the permutation that brings b good pivot columns to the front of $\mathbf{Y}$ is likely to bring b good pivot columns to the front of $\mathbf{A}$. Thus, we may form `HQRRP` as a blocked algorithm, finding b good pivot columns during each step and introducing zeros to these columns as necessary with Householder vectors.

`HQRRP` importantly employs a downdating procedure when forming the sampling matrix $\mathbf{Y}$ during each step. By choosing the random matrix $\mathbf{G}$ to be a rotation of the random matrix in the previous step, we can obtain the new sampling matrix $\mathbf{Y}$ inexpensively. Thus, while `HQRRP` requires more flops than unpivoted QR, factorization with Householder reflectors, the *asymptotic* flop count for the two are the same.

With its fully blocked approach, `HQRRP` provides an improvement on traditional CPQR. Extensive numerical experiments show that the pivot columns selected by `HQRRP` are as good as those of CPQR, so the factorizations from each algorithm provide low rank approximations of approximately the same quality. In shared memory computing environments, `HQRRP` outperforms CPQR in speed, and the gap between the two increases as the number of processor cores increases. This relationship is depicted in Figure 1.3.

1.5.2 "randUTV: A blocked randomized algorithm for computing a rank-revealing UTV factorization"

This chapter presents the second fundamental algorithm presented in this thesis, `randUTV`. This algorithm provides orthogonal matrix factorizations that reveal rank nearly as well as the SVD at speeds that rival, and often surpass, CPQR. It achieves this largely because it casts the bulk of its operations in terms of matrix-matrix multiplication, which is exceptionally efficient in parallel computing environments.

Given an $m \times n$ matrix $\mathbf{A}$, `randUTV` computes orthogonal matrices $\mathbf{U}$ and $\mathbf{V}$ upper triangular matrix $\mathbf{T}$ such that $\mathbf{A} = \mathbf{U}\mathbf{T}\mathbf{V}^*$. $\mathbf{T}$ has the additional sparsity property that the $b \times b$ blocks on its main diagonal are themselves diagonal.

`randUTV` builds the matrix $\mathbf{T}$ in blocks of b columns at a time. In each step, a local rotation matrix $\mathbf{V}^{(i)}$ is chosen to approximate the subspace spanned by the leading right singular vectors of an appropriate submatrix of $\mathbf{T}$. Such a matrix $\mathbf{V}^{(i)}$ can be built efficiently using the `range_finder_basic` algorithm of Section 1.4.1. The second rotation matrix $\mathbf{U}^{(i)}$ can then be built in a manner mirroring the building of the $\mathbf{Q}$ matrix in a traditional QR factorization, where $\mathbf{V}^{(i)}$ takes the place of the pivoting matrix P. A final SVD operation on a $b \times b$ submatrix of $\mathbf{T}$ updates $\mathbf{U}^{(i)}$ and $\mathbf{V}^{(i)}$, increasing the accuracy of the approximation of the diagonal values of $\mathbf{T}$ to the singular values of $\mathbf{A}$.

The value of `randUTV` is best seen in the experiments depicted in Figures 1.4, 1.5, and 1.6. In Figures 1.4 and 1.5, observe that `randUTV` easily outperforms CPQR, even in the case of slow singular value decay in $\mathbf{A}$ (a known difficulty for the `range_finder` used by `randUTV`). Increasing the parameter q (see Algorithm 2) improves the low rank approximations provided by the factorization even further, and Figure 1.6 shows that the increase does not cost too much in extra computation time. Furthermore, Figure 1.6 shows that `randUTV` is comparable to CPQR even for single core. When the number of cores is increased, `randUTV` is *faster* than CPQR starting around $n = 5000$, even for higher values of q.

Remark 1. *The QLP algorithm in Figures 1.4 and 1.5 was introduced in [120, ?] and is included here as a potential competitor to `randUTV`. While it may give factorizations that reveal rank slightly better than `randUTV` with $q = 0$, QLP requires two column pivoted QR factorizations. It is therefore much slower than*

`randUTV`, and for $q > 0$ `randUTV` gives better low rank approximations as well.

1.5.3 "Efficient algorithms for computing a rank-revealing UTV factorization on parallel computing architectures"

In this chapter, efficient implementations of the `randUTV` algorithm are developed for both shared and distributed memory architectures.

The optimized shared memory implementation re-formulates `randUTV` as an algorithm-by-blocks (see Section 1.3.2). This change largely does not affect the actual operations that are performed, with the exception that the unpivoted QR factorizations must be reorganized based on updating a QR factorization that has already been computed. However, an algorithm-by-blocks allows greater flexibility in the order of completion of tasks throughout `randUTV`. The acceleration achieved by reducing the number of synchronization points is significant and increases with the number of cores used.

The distributed memory implementation explores the details of optimizing `randUTV` for this architecture. While the algorithm itself is not modified, the implementation requires careful consideration of system communication and storage. We optimize for the architecture and perform scalability analysis for `randUTV` and competing factorizations.

1.5.4 "Efficient algorithms for computing rank-revealing factorizations on a GPU"

The GPU implementation of `randUTV` in this chapter takes advantage of the extremely efficient performance of matrix-matrix multiplication in that setting. It introduces modifications to `randUTV` which incorporate the oversampling parameter p of `range_finder` to further increase the rank revealing properties of the resulting factorization. The modified algorithm also makes use of a bootstrapping technique wherein the extra samples of one step may be used in the following step, reducing the cost of using $p > 0$. The resulting algorithm, which we call `randUTV_boosted`, provides factorizations that reveal rank even better than `randUTV` at a reasonable extra computation cost.

1.5.5 "Adapting efficient rank-revealing factorization algorithms for matrices stored out of core"

This chapter implements both `randUTV` and `HQRRP` for matrices large enough that they must be stored out of core. For `HQRRP`, we experiment with several options for managing communication and I/O, including a "left-looking" algorithm variant which decreases the amount of time spent writing to the hard drive. For `randUTV`, we re-write the method as an algorithm-by-blocks, to the effect that the I/O operations required by the disk with the flops executed by the processor are largely overlapped. The `HQRRP` implementation is useful for partial factorizations when subset selection is in view. The `randUTV` implementation performs quite well for full factorizations, costing just slightly more than when the matrix can be stored in RAM.

1.5.6 "Efficient nuclear norm approximation via the randomized UTV algorithm"

`randUTV` has a secondary potential use as a nuclear norm estimator. The nuclear norm $\|\cdot\|_\infty$ of a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ is the sum of its singular values, *i.e.*

$$\|\mathbf{A}\|_\infty = \sum_{i=1}^{\min(m,n)} \sigma_i(\mathbf{A}).$$

The diagonal entries of the matrix $\mathbf{T}$ resulting from `randUTV` are good estimates of the singular values of $\mathbf{A}$. If just the singular values of $\mathbf{A}$ are desired, rather than the full factorization, then some of the work related to building the $\mathbf{V}$ and $\mathbf{U}$ matrices may be cut out of the algorithm. One chapter of the thesis examines this idea in more detail and reports on the accuracy of the singular value estimates as well as the acceleration gained by trimming down `randUTV` for use as a nuclear norm estimator.

Acknowledgments: The research reported was supported by the National Science Foundation, under the awards DMS-1407340 and DMS-1620472, by the Office of Naval Research, under the grant N00014-18-1-2354, and by DARPA under the grant N00014-18-1-2354. The author also thanks Francisco D. Igual-Peña from the Universidad Complutense de Madrid (Spain) for granting access to the computing platform where some of the experiments were conducted. Additionally, the author expresses his appreciation to Nvidia for the donation of the GPUs used for the development work in Chapter 5.

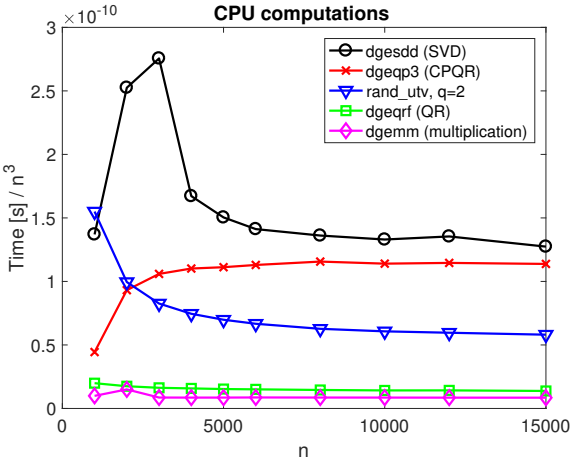


Figure 1.1: A demonstration of the effect of communication cost on execution times of algorithms executed on a shared memory multicore architecture. `rand_utv` is a recently developed factorization algorithm introduced later in this thesis. Note in particular that `dgeqrf` and `dgeqp3` have the same asymptotic flop count (though `dgeqp3` has a higher overall flop count), and `rand_utv` has a *higher* asymptotic flop count that `dgeqp3`.

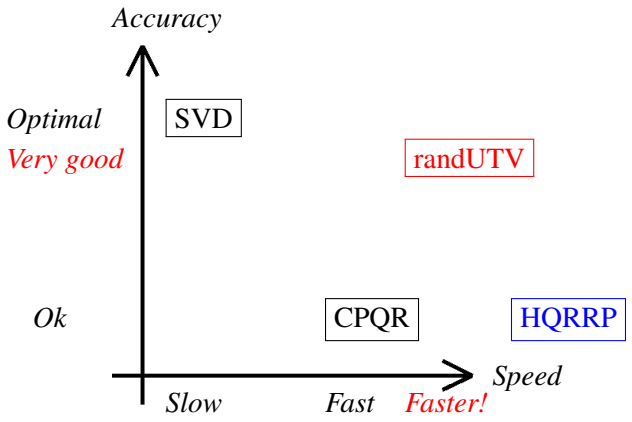


Figure 1.2: A graphic comparing the relative strengths and weaknesses of the most widely used rank-revealing decompositions to the algorithms `HQRRP` and `randUTV` presented in this thesis.

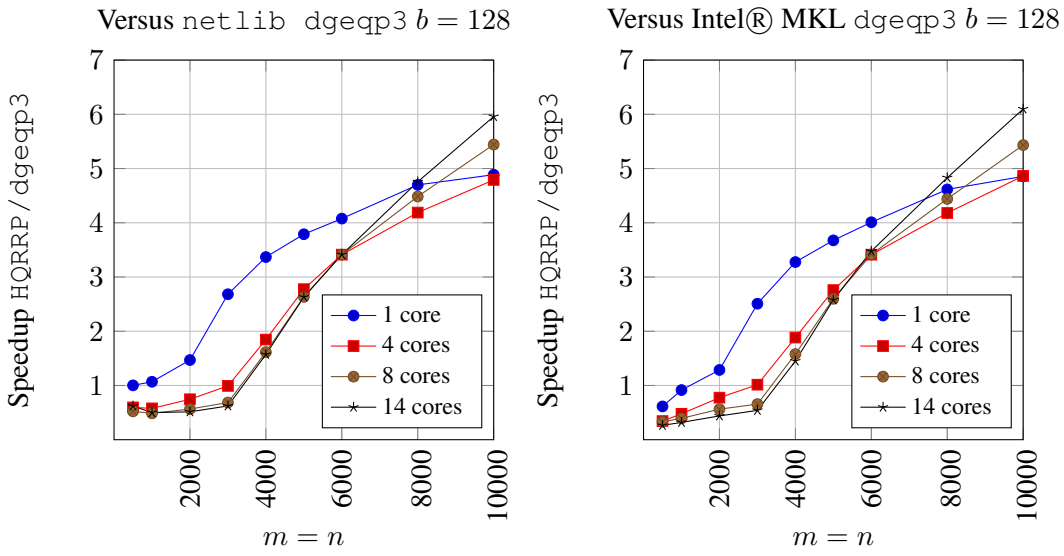


Figure 1.3: Speedup of new blocked Householder QR factorization with randomized column pivoting (HQRRP) relative to LAPACK's faster routine (dgeqp3) on a 14-core Intel Xeon E5-2695 v3.

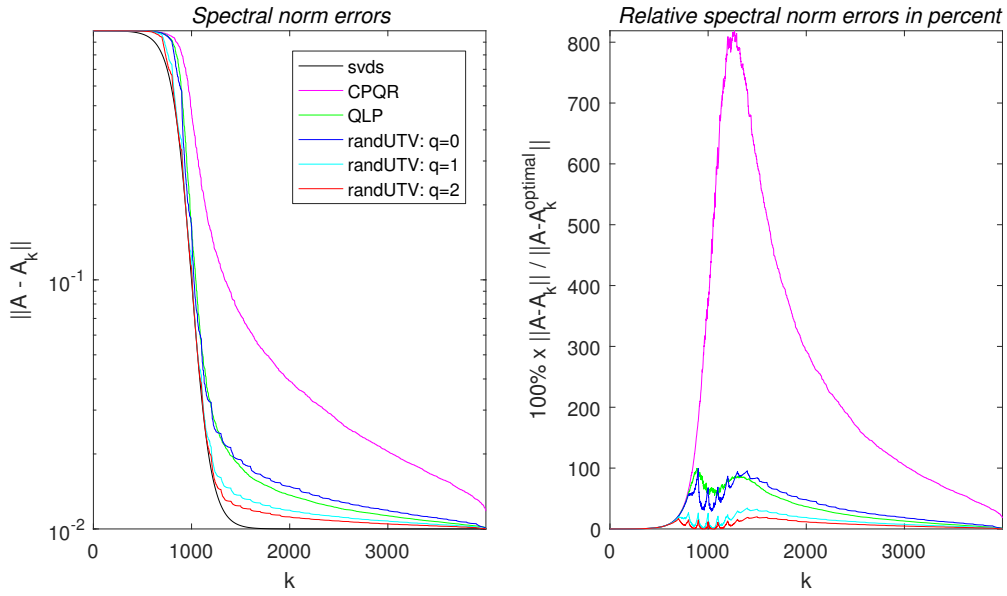


Figure 1.4: Rank-k approximation errors for a matrix of size 4000×4000 with "S-shaped" decay in its singular values. The block size was $b = 100$. Left: Absolute errors in spectral norm, where $\mathbf{A}_k = \mathbf{U}(:, 1:k)\mathbf{T}(:, 1:k)\mathbf{V}^*$. The black line marks the theoretically minimal errors. Right: Relative error.

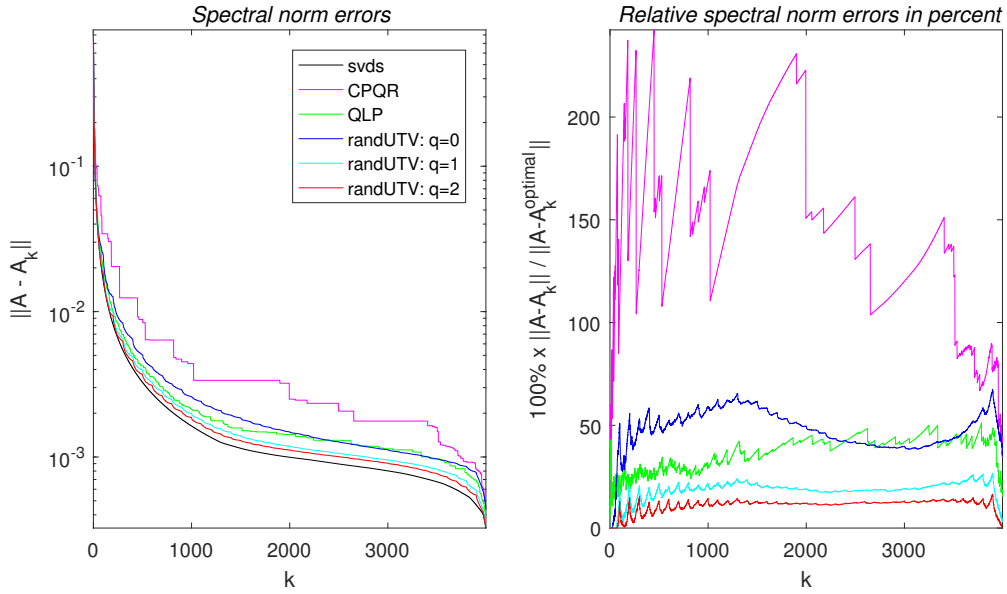


Figure 1.5: Rank-k approximation errors for a matrix of size 4000×4000 determined by a discretized boundary integral equation. The block size was $b = 100$. Left: Absolute errors in spectral norm, where $\mathbf{A}_k = \mathbf{U}(:, 1 : k)\mathbf{T}(:, 1 : k)\mathbf{V}^*$. The black line marks the theoretically minimal errors. Right: Relative errors.

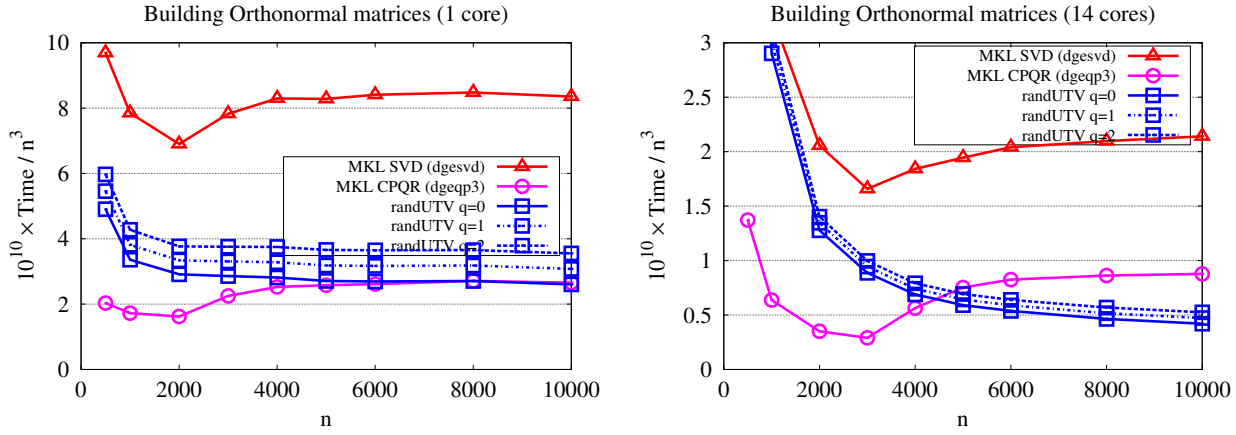


Figure 1.6: Computational cost (the lower, the better performances) of `randUTV` compared to the cost of the LAPACK routines `dgesvd` (SVD) and `dgeqp3` (CPQR). The algorithms were applied to double-precision real matrices of size $n \times n$. The measured computational time divided by n^3 is plotted against n (e.g. the red line is T_{svd}/n^3).

Chapter 2

Householder QR Factorization With Randomization for Column Pivoting (HQRRP)

A fundamental problem when adding column pivoting to the Householder QR factorization is that only about half of the computation can be cast in terms of high performing matrix-matrix multiplications, which greatly limits the benefits that can be derived from so-called blocking of algorithms. This paper describes a technique for selecting groups of pivot vectors by means of randomized projections. It is demonstrated that the asymptotic flop count for the proposed method is $2mn^2 - (2/3)n^3$ for an $m \times n$ matrix, identical to that of the best classical unblocked Householder QR factorization algorithm (with or without pivoting). Experiments demonstrate acceleration in speed of close to an order of magnitude relative to the GEQP3 function in LAPACK, when executed on a modern CPU with multiple cores. Further, experiments demonstrate that the quality of the randomized pivot selection strategy is roughly the same as that of classical column pivoting. The described algorithm is made available under Open Source license and can be used with LAPACK or libflame.

2.1 Introduction

The QR factorization is a staple of linear algebra, with applications ranging from Linear Least-Squares solution of overdetermined systems to the identification of low rank approximation via the determination of an approximate orthonormal basis for the column space. Standard algorithms for computing the QR factorization include Gram-Schmidt orthogonalization and those based on Householder transformations (reflectors). When it is desirable for the QR factorization to also reveal the approximate rank of the original matrix, it is important that the elements of the diagonal of R be ordered with larger elements in magnitude

appearing earlier. In this case, column pivoting (swapping) is employed during the QR factorization, yielding QR factorization with column pivoting (QRP). It is well-known that the Householder QR factorization (HQR) yields columns of Q that are orthogonal to a high degree of precision, making these algorithms the weapon of choice in many situations. Pivoting can be added to HQR to yield HQR with column pivoting (HQRP). This topic is covered by standard texts on numerical linear algebra [57].

To achieve high performance for dense linear algebra algorithms, so-called blocked algorithms are employed that cast most computation in terms of matrix-matrix operations supported by the widely used level-3 Basic Linear Algebra Subprograms (BLAS) [38, 40] because such operations can be implemented to achieve very high performance on modern processors via a combination of careful reuse of data in the caches and low level implementation in terms of assembly code or intrinsic vector operations. Widely used current implementations of the level-3 BLAS are based on techniques exposed by Goto [60, 59] and available in open source libraries including the OpenBLAS [132] (a fork of the GotoBLAS) and BLIS [126], as well as vendor implementions including AMD's ACML [3], Intel's MKL [73], and IBM's ESSL [71] libraries.

The fundamental problem with the classical approach to HQRP is that only half of the computation can be cast in terms of GEMM, as described in the paper [107] that underlies LAPACK's `geqp3` routine [4]. This means that blocking can only improve performance by, at best, a factor two, which is inherent from the fact that it must be known how remaining columns will be updated in order to compute the 2-norms of remaining columns. Bischof and Quintana-Ortí describe in a pair of papers [14, 13] an attempt to overcome this problem by using so called "window pivoting" in combination with HQR. While much faster than `geqp3`, this approach is more complicated than the method proposed in this paper and never made it into LAPACK.

The present paper proposes to solve the problem by means of randomized projections. To describe the idea, suppose that we seek to determine a set of b good pivot columns in an $m \times n$ matrix A. We then draw a Gaussian random matrix G of size $b \times m$ and form a *sampling matrix* $Y = GA$. Then determine the b pivot columns by executing QRP of the sampling matrix Y. Since Y is small compared to A, this task can be executed very efficiently. (To be precise, one needs to perform a very slight amount of over-sampling, see Section 2.3.1.) With this observation, it becomes easy to block the Householder QR factorization with

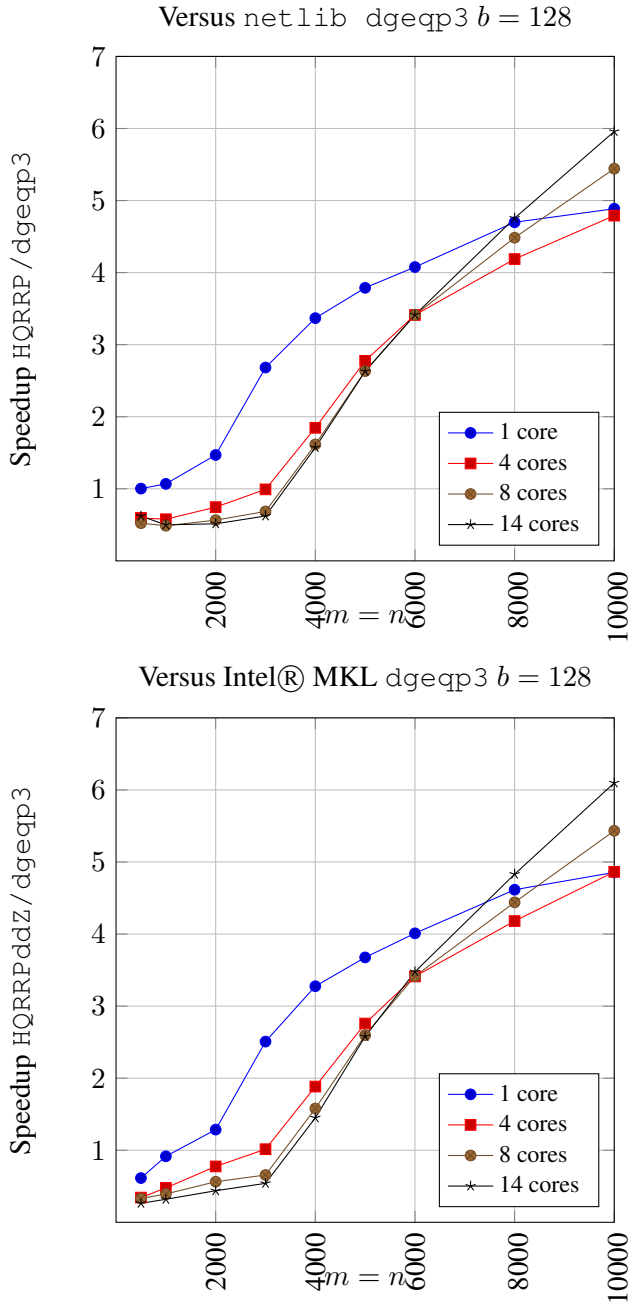


Figure 2.1: Speedup of new blocked Householder QR factorization with randomized column pivoting (HQRRP) relative to LAPACK's faster routine (dgeqp3) on a 14-core Intel Xeon E5-2695 v3, see Section 2.4.1 for details.

column pivoting. At each iteration of the blocked algorithm, we use the randomized sampling approach to identify a set of b columns are that are then moved to the front of the actual matrix, at which point a regular

step of HQR can be used to move the computation forward, optionally with additional column pivoting only within a narrow panel of the matrix. Importantly, the sampling matrix can be cheaply downdated rather than recomputed at each step, allowing the performance of the proposed algorithm to asymptotically approach that of a standard blocked HQR implementation that does not pivot columns. Fig. 2.1 illustrates the dramatic performance improvements that are realized.

The idea to use randomized sampling to pick blocks of pivot vectors was first published by Martinsson on ArXiv in May 2015 [93]. A very similar technique was published by Duersch and Gu in September 2015 [43], also on ArXiv. The observation that downdating of the sampling matrix enables the randomized scheme to attain the same asymptotic flop count as classical HQRP was discovered independently by the two groups and was first published in [43]. More broadly, the idea that one can select a subset of columns of a matrix that forms a good approximate basis for the column space of the matrix by performing QRP on a small matrix whose rows are random linear combinations of the rows of the original matrix was first described in [90, Sec. 4.1] and later elaborated in [83, 69, 91]. This problem is closely related to the problem of finding a set of columns of maximal spanning volume [64], and to the problem of finding so called *CUR* and *interpolative* decompositions [128]. These ideas tie in to a larger literature on randomized techniques for computing low-rank approximations of matrices that includes [52, 42, 85, 86].

This paper describes a practical implementation of the method proposed that can be incorporated in libraries like LAPACK and `libflame` [124, 125]. Implementation details that are important for attaining high practical performance are described to enable readers to reproduce and extend the ideas. The paper provides a cost analysis that shows that asymptotically the number of floating point operations approaches that of HQR without pivoting while most computation is cast in terms of matrix-matrix multiplication like the corresponding blocked HQR without pivoting. It reports unprecedented performance for pivoted QR factorization on current architectures and provides empirical quality results. Importantly, the implementation is made available for use by the computational science community under an Open Source license. The conclusion discusses how these results pave the way for future opportunities.

The paper is organized as follows: Section 2 lists some standard facts about Householder reflectors and pivoted QR factorizations that we need in the presentation. Section 3 describes how the classical House-

holder QR factorization algorithm can be blocked by using randomization in the pivot selection step. Section 4 reports the results from numerical experiments investigating the speed of the algorithm and the quality of the pivoting selection strategy. Sections 5 summarizes the key results and discusses future work. Section 6 describes publicly available software that implements the techniques presented.

2.2 Householder QR Factorization

In this section, we briefly review the state-of-the art regarding Householder factorization based on Householder transformations (HQR). Throughout, we use the FLAME notation for representing dense linear algebra algorithms [104, 65].

2.2.1 Householder transformations (reflectors)

A standard topic in numerical linear algebra is the concept of a reflector, also known as a Householder transformation [57]. The review in this subsection follows [74] in which a similar notation is also employed.

Given a nonzero vector $u \in \mathbb{C}^n$, the matrix $H(u) = I - \frac{1}{\tau} u u^H$ with $\tau = \frac{u^H u}{2}$ has the property that it reflects a vector to which it is applied with respect to the subspace orthogonal to u. Given a vector x, the vector u and scalar τ can be chosen so that $H(u)x$ equals a multiple of e_0, the first column of the identiy matrix. Furthermore, u can be normalized so that its first element equals one.

In our discussions, given a vector $x = \left(\begin{array}{c} \chi_1 \\ \hline x_2 \end{array}\right)$, the function $\left[\left(\begin{array}{c} \rho \\ \hline u_2 \end{array}\right), \tau\right] := \mathrm{Housev}\left(\left(\begin{array}{c} \chi_1 \\ \hline x_2 \end{array}\right)\right)$ computes the vector $u = \left(\begin{array}{c} 1 \\ \hline u_2 \end{array}\right)$ and $\tau = \frac{u^H u}{2}$ so that $H(u)x = \rho e_0$,

2.2.2 Unblocked Householder QR factorization

A standard unblocked algorithm for HQR of a given matrix $A \in \mathbb{C}^{m \times n}$, typeset using the FLAME notation, is given in Fig. 2.2 (left). The body of the loop computes

$$\left[\left(\begin{array}{c} \rho_{11} \\ \hline u_{21} \end{array}\right), \tau_{11}\right] := \mathrm{Housev}\left(\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)\right),$$

which overwrites a_{11} with ρ_{11} and a_{21} with u_{21}, after which the remainder of A is updated by

$$\left(\begin{array}{c} a_{12}^T \\ \hline A_{22} \end{array}\right) := \left(I - \frac{1}{\tau_{11}} \left(\begin{array}{c} 1 \\ \hline u_{21} \end{array}\right) \left(\begin{array}{c} 1 \\ \hline u_{21} \end{array}\right)^H \right) \left(\begin{array}{c} a_{12}^T \\ \hline A_{22} \end{array}\right).$$

Upon completion, the (Householder) vectors that define the Householder transformations have overwritten the elements in which they introduced zeroes, and the upper triangular part of A contains R. How the matrix T fits into the picture will become clear next.

2.2.3 The UT transform: Accumulating Householder transformations

Given $A \in \mathbb{C}^{n \times b}$, let U contain the Householder vectors computed during the HQR of A. Let us assume that $H(u_{b-1}) \cdots H(u_1)H(u_0)A = R$. Then there exists an upper triangular matrix so that $I - UT^{-H}U^H = H(u_{b-1}) \cdots H(u_1)H(u_0)$. The desired matrix T equals the strictly upper triangular part of U^HU with the diagonal elements equal to $\tau_0, \ldots, \tau_{b-1}$. The matrix T can be computed during the unblocked HQR, as indicated in Fig. 2.2 (left). In [74], the transformation $I - UT^{-1}U^H$ that equals the accumulated Householder transformations is called the **UT transform**. The UT transform is conceptually related to the more familiar WY transform [15] and compact WY transform [113], see [74] for details on how the different representations relate to one another.

2.2.4 A blocked QR Householder factorization algorithm

A blocked algorithm for HQR that exploits the insights that resulted in the UT transform can now be described as follows. Partition

$$A \rightarrow \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right)$$

where A_{11} is $b \times b$. We can use the unblocked algorithm in Fig. 2.2 (left) to factor the panel $\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$, creating matrix T_{11} as a side effect.

Algorithm: $[A, T] := \text{HQR\_UNB}(A, T)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$,
$T \rightarrow \left(\begin{array}{c|c} T_{TL} & T_{TR} \\ \hline 0 & T_{BR} \end{array}\right)$
where A_{TL} is 0×0, T_{TL} is 0×0

while $m(A_{TL}) < m(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right),$$
$$\left(\begin{array}{c|c} T_{TL} & T_{TR} \\ \hline 0 & T_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} T_{00} & t_{01} & T_{02} \\ \hline 0 & \tau_{11} & t_{12}^T \\ \hline 0 & 0 & T_{22} \end{array}\right)$$

where α_{11} is 1×1, τ_{11} is 1×1

$$\left[\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right), \tau_{11}\right] := \text{Housev}\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$$
$$w_{12}^T := (a_{12}^T + a_{21}^H A_{22})/\tau_{11}$$
$$\left(\begin{array}{c} a_{12}^T \\ \hline A_{22} \end{array}\right) := \left(\begin{array}{c} a_{12}^T - w_{12}^T \\ \hline A_{22} - a_{21} w_{12}^T \end{array}\right)$$
$$t_{01} := (a_{10}^T)^H + A_{20}^H a_{21}$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right),$$
$$\left(\begin{array}{c|c} T_{TL} & T_{TR} \\ \hline 0 & T_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} T_{00} & t_{01} & T_{02} \\ \hline 0 & \tau_{11} & t_{12}^T \\ \hline 0 & 0 & T_{22} \end{array}\right)$$

endwhile

Algorithm: $[A, T] := \text{HQR\_BLK}(A, T)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$,
$T \rightarrow \left(\begin{array}{c} T_T \\ \hline T_B \end{array}\right)$
where A_{TL} is 0×0, T_T has 0 rows

while $m(A_{TL}) < m(A)$ **do**
Determine block size b
Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right),$$
$$\left(\begin{array}{c} T_T \\ \hline T_B \end{array}\right) \rightarrow \left(\begin{array}{c} T_0 \\ \hline T_1 \\ \hline T_2 \end{array}\right)$$

where A_{11} is $b \times b$, T_1 has b rows

$$\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), T_1\right] := \text{HQR\_UNB}\left(\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), T_1\right)$$
$$W_{12} := T_1^{-H}(U_{11}^H A_{12} + U_{21}^H A_{22})$$
$$\left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right) := \left(\begin{array}{c} A_{12} - U_{11} W_{12} \\ \hline A_{22} - U_{21} W_{12} \end{array}\right)$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right),$$
$$\left(\begin{array}{c} T_T \\ \hline T_B \end{array}\right) \leftarrow \left(\begin{array}{c} T_0 \\ \hline T_1 \\ \hline T_2 \end{array}\right)$$

endwhile

Figure 2.2: Left: Unblocked Householder transformation based QR factorization merged with the computation of T for the UT transform. Right: Blocked Householder transformation based QR factorization. In this algorithm, U_{11} is the unit lower triangular matrix stored below the diagonal of A_{11} and U_{21} is stored in A_{21}.

Now we need to also apply the UT transform to the rest of the columns:

$$\left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right) := \left(I - \left(\begin{array}{c} U_{11} \\ \hline U_{21} \end{array}\right) T_{11}^{-1} \left(\begin{array}{c} U_{11} \\ \hline U_{21} \end{array}\right)^H\right)^H \left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right) = \left(\begin{array}{c} A_{12} - U_{11} W_{12} \\ \hline A_{22} - U_{21} W_{12} \end{array}\right),$$

where $W_{12} = T_{11}^{-H}(U_{11}^H A_{12} + U_{21}^H A_{22})$. This motivates the blocked HQR algorithm in Fig. 2.2 (right) which we will refer to as HQR_BLK.

The benefit of the blocked algorithm is that it casts most computation in terms of the computations $U_{21}^H A_{22}$ (row panel times matrix multiply) and $A_{22} - U_{21}W_{12}$ (rank-b update). Such matrix-matrix multiplications can attain high performance by amortizing data movement between memory layers.

These insights form the basis for the LAPACK routine GEQRF (except that it uses a compact WY transform instead of the UT transform).

2.2.5 Householder QR factorization with column pivoting

An unblocked (rank-revealing) Householder QR factorization with column pivoting (HQRP) swaps the column of A_{BR} with largest 2-norm with the first column of that matrix at the top of the loop body. As a result, the diagonal elements of matrix R are ordered from largest to smallest in magnitude, which, for example, allows the resulting QR factorization to be used to identify a high quality approximate low-rank orthonormal basis for the column space of A. (To be precise, column pivoted QR returns a high quality basis in most cases but may produce strongly sub-optimal results in rare situations. For details, see [76, 64], and the description of "Matrix 4" in Section 2.4.2.)

The fundamental problem with the best known algorithm for HQRP, which underlies LAPACK's routine `geqp3`, is that it only casts half of the computation in terms of matrix-matrix multiplication [107]. The unblocked algorithm called from the blocked algorithm operates on the entire "remaining matrix" (A_{BR} in the blocked algorithm), computes b more Householder transforms and b more rows of R, computes the matrix W_2, and returns the information about how columns were swapped. In the blocked algorithm itself, only the update $A_{22} - A_{21}W_2$ remains to be performed. When only half the computation can be cast in terms of matrix-matrix multiplication, the resulting blocked algorithm is only about twice as fast as the unblocked algorithm.

Algorithm: $[A, T, s] := \text{HQRP\_RANDOMIZED\_BLK}(A, T, s, n_b, p)$

$G := \text{RAND\_IID}(n_b + p, n(A))$
$Y := GA$
Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), T \rightarrow \left(\begin{array}{c} T_T \\ \hline T_B \end{array}\right), s \rightarrow \left(\begin{array}{c} s_T \\ \hline s_B \end{array}\right), Y \rightarrow \left(\begin{array}{c|c} Y_L & Y_R \end{array}\right)$
where A_{TL} is 0×0, T_T has 0 rows, s_T has 0 rows, Y_L has 0 columns

while $m(A_{TL}) < m(A)$ **do**

Determine block size $b = \min(n_b, n(A_{BR}))$
Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} T_T \\ \hline T_B \end{array}\right) \rightarrow \left(\begin{array}{c} T_0 \\ \hline T_1 \\ \hline T_2 \end{array}\right),$$

$$\left(\begin{array}{c} s_T \\ \hline s_B \end{array}\right) \rightarrow \left(\begin{array}{c} s_0 \\ \hline s_1 \\ \hline s_2 \end{array}\right), \left(\begin{array}{c|c} Y_L & Y_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} Y_0 & Y_1 & Y_2 \end{array}\right)$$

where A_{11} is $b \times b$, T_1 has b rows, s_1 has b rows, Y_1 has b columns

---

$s_1 := \text{DETERMINEPIVOTS}(\left(\begin{array}{c|c} Y_1 & Y_2 \end{array}\right), b)$
$\left(\begin{array}{c|c} A_{01} & A_{02} \\ \hline A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) := \text{SWAPCOLS}(s_1, \left(\begin{array}{c|c} A_{01} & A_{02} \\ \hline A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right))$
$\left[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), T_1, s_1'\right] := \text{HQR P\_UNB}(\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), T_1)$
$A_{01} := \text{SWAPCOLS}(s_1', A_{01})$
$s_1 := \text{UPDATEPIVOTINFO}(s_1', s_1)$
$W_{12} := T_1^{-H}(U_{11}^H A_{12} + U_{21}^H A_{22})$
$\left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right) := \left(\begin{array}{c} A_{12} - U_{11} W_{12} \\ \hline A_{22} - U_{21} W_{12} \end{array}\right)$
$\left(\begin{array}{c|c} Y_1 & Y_2 \end{array}\right) := \text{SWAPCOLS}(s_1, \left(\begin{array}{c|c} Y_1 & Y_2 \end{array}\right))$
$Y_2 := Y_2 - \left(G_1 - (G_1 U_{11} + G_2 U_{21}) T_{11}^{-1} {U_{11}}^H\right) R_{12}.$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} T_T \\ \hline T_B \end{array}\right) \leftarrow \left(\begin{array}{c} T_0 \\ \hline T_1 \\ \hline T_2 \end{array}\right),$$

$$\left(\begin{array}{c} s_T \\ \hline s_B \end{array}\right) \leftarrow \left(\begin{array}{c} s_0 \\ \hline s_1 \\ \hline s_2 \end{array}\right), \left(\begin{array}{c|c} Y_L & Y_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} Y_0 & Y_1 & Y_2 \end{array}\right)$$

endwhile

Figure 2.3: Blocked Householder transformation based QR factorization with column pivoting based on randomization. In this algorithm, U_{11} equals the unit lower triangiular matrix stored below the diagonal of A_{11}, $U_{21} = A_{21}$, and $R_{12} = A_{12}$. The steps highlighted in gray constitute the blocked QR factorization without column pivoting from Fig. 2.2.

2.3 Randomization to the Rescue

This section describes a computationally efficient technique for picking a selection of b columns from a given $n \times n$ matrix A that form good choices for the first b pivots in a blocked HQRP algorithm. Observe that this task is closely related to the problem of finding an index set s of length b such that the columns in $A(:, s)$ form a good approximate basis for the column space of A. Another way of expressing this problem is that we are looking for a collection of b columns whose spanning volume in $\mathbb{C}^n$ is close to maximal. To find the absolutely optimal choice here is a hard problem [64], but luckily, for pivoting purposes it is sufficient to find a choice that is "good enough."

2.3.1 Randomized pivot selection

The strategy that we propose is simple. The idea is to perform classical QR factorization with column pivoting (QRP) on a matrix Y that is much smaller than A, so that performing QRP with that matrix constitutes a lower order cost. As a bonus, it may fit in fast cache memory. This smaller matrix can be constructed by forming random linear combinations of the rows of A as follows:

1. Fix an over-sampling parameter p. Setting $p = 5$ or $p = 10$ are good choices.

2. Form a random matrix G of size $(b + p) \times n$ whose entries are drawn independently from a normalized Gaussian distribution.

3. Form the $(b + p) \times n$ sampling matrix $Y = GA$.

The sampling matrix Y has as many columns as A, but much fewer rows. Now execute b steps of a column pivoted QR factorization to determine an integer vector with b elements that capture how columns need to be pivoted:

$$s = \text{DETERMINEPIVOTS}(Y, b).$$

In other words, the columns $Y(:, s)$ are good pivot columns for Y. Our claim is that due to the way Y is constructed, the columns $A(:, s)$ are then also good choices for pivot columns of A. This claim is supported by extensive numerical experiments, some of which are given in Section 2.4.2. There is theory supporting

the claim that these b columns form a good approximate basis for the column space of A, see, e.g. [69, Sec. 5.2] and [83, 128], but it has not been directly proven that they form good choices as pivots in a QR factorization. This should not be surprising given that it is *known* that even classical column pivoting can result in poor choices [76]. Known algorithms that are provably good are all far more complex to implement [64].

Notice that there are many choices of algorithms that can be employed to determine the pivots. For example, since high numerical accuracy is not necessary, the classical Modified Gram-Schmidt (MGS) algorithm with column pivoting is a simple yet effective choice.

The randomized strategy described here for determining a block of pivot vectors is inspired by a technique published in [90, Sec. 4.1] for computing a low-rank approximation to a matrix, and later elaborated in [83, 69, 91].

Remark 2 (Choice of over-sampling parameter p)**.** *The reliability of the procedures described in this section depends on the choice of the over-sampling parameter p. It is well understood how large p needs to be in order to determine a high-quality approximate basis for the column space of A with extremely high reliability: the choice $p = 5$ is very good, $p = 10$ is excellent, and $p = b$ is almost always over-kill [69]. The pivot selection problem is less well studied, but is more forgiving. (The choice of pivots does not necessarily have to be particularly optimal.) Numerical experiments indicate that even setting $p = 0$ typically results in good choices. However, the choices $p = 5$ or $p = 10$ appear to be good generic values that have resulted in excellent choices in every experiment we have run.*

Remark 3 (Intuition of random projections)**.** *To understand why the pivot columns selected by processing the small matrix Y also form good choices for the original matrix A, it might be helpful to observe that for a Gaussian random matrix G of size $\ell \times n$, it is the case that for any $x \in \mathbb{R}^n$, we have $\mathbb{E}\big[\|Gx\|\big] = \|x\|$, where $\mathbb{E}$ denotes expectation. Moreover, as the number of rows ℓ grows, the probability distribution of $\|Gx\|$ concentrates tightly around its expected value, see, e.g., [130, Sec. 2.4] and the references therin. This means that for any pair of indices $i, j \in \{1, 2, \ldots, n\}$ we have $\mathbb{E}\big[\|Y(:, i) - Y(:, j)\|\big] = \|A(:, i) - A(:, j)\|$. This simple observation does not in any way provide a proof that the randomized strategy we propose works,*

but might help understand the underlying intuition.

2.3.2 Efficient downdating of the sampling matrix Y

For the QRP factorization algorithm, it is well known that one does not need to recompute the column norms of the remainder matrix after each step. Instead, these can cheaply be downdated, as described, e.g., in [116, Ch.5, Sec. 2.1]. In terms of asymptotic flop counts, this observation makes the cost of pivoting become a lower order term, and consequently both unpivoted and pivoted Householder QR algorithms have the same leading order term $(4/3)n^3$ in their asymptotic flop[1] counts. In this section, we describe an analogous technique for the randomized sampling strategy described in Section 2.3.1. This downdating strategy was discovered by one of the authors; a closely related technique was discovered independently and published to arXiv by Duersch and Gu [43] in September 2015.

First observe that if the randomized sampling technique described in Section 2.3.1 is used in the obvious fashion, then each step of the iteration requires the generation of a Gaussian random matrix G and a matrix-matrix multiply involving the remaining portion of A in the lower right corner to form the sampling matrix Y. The number of flops required by the matrix-matrix multiplications add up to an $O(n^3)$ term. However, it turns out to be possible to avoid computing a sampling matrix Y from scratch at every step. The idea is that if we select the randomizing matrix G in a particular way in every step beyond the first, then the corresponding sampling matrix Y can inexpensively be computed by downdating the sampling matrix from the previous step.

To illustrate, suppose that we start with an $n \times n$ original matrix $A = A^{(0)}$. In the first blocked step, we draw a $(b + p) \times n$ randomizing matrix $G^{(1)}$ and form the $(b + p) \times n$ sampling matrix

$$Y^{(1)} = G^{(1)} A^{(0)}. \tag{2.1}$$

Using the information in $Y^{(1)}$, we identify the b pivot vectors and form the corresponding permutation matrix $P^{(1)}$. Then the matrix $Q^{(1)}$ representing the b Householder reflectors dictated by the b pivot columns

[1] We use the standard convention of counting one multiply and one add as one flop, regardless of whether a complex or real operation is performed.

is formed. Applying these transforms to the right and the left of $A^{(0)}$, we obtain the matrix

$$A^{(1)} = \big(Q^{(1)}\big)^* A^{(0)} P^{(1)}. \tag{2.2}$$

To select the pivots in the next step, we need to form a randomizing matrix $G^{(2)}$ and a sampling matrix $Y^{(2)}$ that are related through

$$Y^{(2)} = G^{(2)}\big(A^{(1)} - R^{(1)}\big), \tag{2.3}$$

where $R^{(1)}$ holds the top b rows of $A^{(1)}$ so that

$$A^{(1)} - R^{(1)} = \begin{pmatrix} A_{11}^{(1)} & A_{12}^{(1)} \\ \text{o} & A_{22}^{(1)} \end{pmatrix} - \begin{pmatrix} A_{11}^{(1)} & A_{12}^{(1)} \\ \text{o} & \text{o} \end{pmatrix} = \begin{pmatrix} \text{o} & \text{o} \\ \text{o} & A_{22}^{(1)} \end{pmatrix}.$$

The key idea is now to *choose* the randomizing matrix $G^{(2)}$ according to the formula

$$G^{(2)} = G^{(1)} Q^{(1)}. \tag{2.4}$$

Inserting (2.4) into (2.3), we now find that the sampling matrix is

$$Y^{(2)} = G^{(1)} Q^{(1)}\big(A^{(1)} - R^{(1)}\big) = \{\text{Use (2.2)}\} =$$
$$G^{(1)} A^{(0)} P^{(1)} - G^{(1)} Q^{(1)} R^{(1)} = \{\text{Use (2.1)}\} = Y^{(1)} P^{(1)} - G^{(1)} Q^{(1)} R^{(1)}. \tag{2.5}$$

Evaluating formula (2.5) is inexpensive since the first term is a permutation of the columns of the sampling matrix $Y^{(1)}$ and the second term is a product of thin matrices (recall that $Q^{(1)}$ is a product of b Householder reflectors).

Remark 4. *Since the probability distribution for Gaussian random matrices is invariant under unitary maps, the formula (2.4) appears quite safe. After all, $G^{(1)}$ is Gaussian, and $Q^{(1)}$ is just a sequence of reflections, so it might be tempting to conclude that the new randomizing matrix must be Gaussian too. However, the matrix $Q^{(1)}$ unfortunately depends on the draw of $G^{(1)}$, so this argument does not work. Nevertheless, the dependence of $Q^{(1)}$ on $G^{(1)}$ is very subtle since this $Q^{(1)}$ is dictated primarily by the directions of the good pivot columns. Extensive practical experiments (see, e.g., Section 2.4.2) indicate that the pivoting strategy described in this section based on downdating is just as good as the one that uses "pure" Gaussian matrices that was described in Section 2.3.1.*

2.3.3 Detailed description of the downdating procedure

Having described the downdating procedure informally in Section 2.3.2, we in this section provide a detailed description using the notation for HQR that we used in Section 2.2. First, let us assume that one iteration of the blocked algorithm has completed, so that, at the bottom of the loop body, the matrix A contains

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{c|c} U\backslash R_{11} & R_{12} \\ \hline U_{21} & \widehat{A}_{22} - U_{21}W_{12} \end{array}\right).$$

Here $\widehat{A}$ denotes the original contents of AP_1, where P_1 captures how columns have been swapped so far. Hence, cf. (2.2),

$$\underbrace{\left(I - \left(\begin{array}{c} U_{11} \\ \hline U_{21} \end{array}\right) T_{11}^{-H} \left(\begin{array}{c} U_{11} \\ \hline U_{21} \end{array}\right)^H\right)}_{\left(\begin{array}{c|c} Q_1 & Q_2 \end{array}\right)} \left(\begin{array}{c|c} \widehat{A}_{11} & \widehat{A}_{12} \\ \hline \widehat{A}_{21} & \widehat{A}_{22} \end{array}\right) P_1 = \left(\begin{array}{c|c} R_{11} & R_{12} \\ \hline 0 & A_{22} \end{array}\right)$$

Now, let $\widetilde{G}_2$ be the next sampling matrix and $\widetilde{Y}_2 = \widetilde{G}_2 A_{22}$. In order to show how this new sampling matrix can be computed by downdating the last sampling matrix, consider that

$$\left(\begin{array}{c|c} \widetilde{Y}_1 & \widetilde{Y}_2 \end{array}\right) = \left(\begin{array}{c|c} \widetilde{G}_1 & \widetilde{G}_2 \end{array}\right) \left(\begin{array}{c|c} 0 & 0 \\ \hline 0 & A_{22} \end{array}\right)$$

for some matrix $\widetilde{G}_1$ and that

$$\begin{aligned} \left(\begin{array}{c|c} \widetilde{G}_1 & \widetilde{G}_2 \end{array}\right) \left(\begin{array}{c|c} 0 & 0 \\ \hline 0 & A_{22} \end{array}\right) &= \left(\begin{array}{c|c} \widetilde{G}_1 & \widetilde{G}_2 \end{array}\right) \left(\left(\begin{array}{c|c} R_{11} & R_{12} \\ \hline 0 & A_{22} \end{array}\right) - \left(\begin{array}{c|c} R_{11} & R_{12} \\ \hline 0 & 0 \end{array}\right)\right) \\ &= \left(\begin{array}{c|c} \widetilde{G}_1 & \widetilde{G}_2 \end{array}\right) \left(\begin{array}{c|c} R_{11} & R_{12} \\ \hline 0 & A_{22} \end{array}\right) - \left(\begin{array}{c|c} \widetilde{G}_1 & \widetilde{G}_2 \end{array}\right) \left(\begin{array}{c|c} R_{11} & R_{12} \\ \hline 0 & 0 \end{array}\right) \\ &= \left(\begin{array}{c|c} \widetilde{G}_1 & \widetilde{G}_2 \end{array}\right) \left(\begin{array}{c|c} Q_1 & Q_2 \end{array}\right) \widehat{A}P_1 - \left(\begin{array}{c|c} \widetilde{G}_1 & \widetilde{G}_2 \end{array}\right) \left(\begin{array}{c|c} Q_1 & Q_2 \end{array}\right) \left(\begin{array}{c|c} Q_1 & Q_2 \end{array}\right)^H \left(\begin{array}{c|c} R_{11} & R_{12} \\ \hline 0 & 0 \end{array}\right) \end{aligned} \tag{2.6}$$

The choice of randomizing matrix analogous to (2.4) is now

$$\left(\begin{array}{c|c} \widetilde{G}_1 & \widetilde{G}_2 \end{array}\right) = \left(\begin{array}{c|c} G_1 & G_2 \end{array}\right) \left(\begin{array}{c|c} Q_1 & Q_2 \end{array}\right). \tag{2.7}$$

Inserting the choice (2.7) into (2.6), we obtain

$$\left(\widetilde{Y}_1 \middle| \widetilde{Y}_2\right) = \underbrace{\left(G_1 \middle| G_2\right)\widehat{A}}_{\left(Y_1 \middle| Y_2\right)} P_1 - \left(G_1 \middle| G_2\right)\left(I - \begin{pmatrix} U_{11} \\ \hline U_{21} \end{pmatrix} T_{11}^{-H} \begin{pmatrix} U_{11} \\ \hline U_{21} \end{pmatrix}^H\right)^H \left(\begin{array}{c|c} R_{11} & R_{12} \\ \hline 0 & 0 \end{array}\right).$$

Letting $\left(\overline{Y}_1 \middle| \overline{Y}_2\right) = \left(Y_1 \middle| Y_2\right) P_1$ we conclude that

$$\begin{aligned}
\widetilde{Y}_2 &= \overline{Y}_2 - \left(G_1 \middle| G_2\right)\left(I - \begin{pmatrix} U_{11} \\ \hline U_{21} \end{pmatrix} T_{11}^{-1} \begin{pmatrix} U_{11} \\ \hline U_{21} \end{pmatrix}^H\right)\begin{pmatrix} R_{12} \\ \hline 0 \end{pmatrix} \\
&= \overline{Y}_2 - \left(G_1 \middle| G_2\right)\left(\begin{pmatrix} R_{12} \\ \hline 0 \end{pmatrix} - \begin{pmatrix} U_{11} \\ \hline U_{21} \end{pmatrix} T_{11}^{-1} {U_{11}}^H R_{12}\right) \\
&= \overline{Y}_2 - \left(G_1 - (G_1 U_{11} + G_2 U_{21}) T_{11}^{-1} {U_{11}}^H\right) R_{12}.
\end{aligned}$$

which can then be used to downdate the sampling matrix Y.

2.3.4 The blocked algorithm

In Fig. 2.3, we give the blocked algorithm that results when the randomized pivot selection strategy described in Section 2.3.1 is combined with the downdating techniques described in Section 2.3.3. In that figure, we use an unblocked HQR algorithm with column pivoting to factor the current column panel so that the diagonal elements within blocks on the diagonal of R are ordered from largest to smallest in magnitude.

2.3.5 Asymptotic cost analysis

In analyzing the cost of the algorithm, we note that all steps in Fig. 2.3 highlighted in grey are part of the blocked HQR algorithm, which is known to have a cost of approximately $4/3n^3$ flops. This leaves us to discuss the overhead related to the other operations.

- $G := \text{RAND\_IID}(b+p, n(A))$: Cost: ignored.
- $Y := GA$: Cost: $2(b+p)n^2$ flops.

- $s_1 :=$ DETERMINEPIVOTS$(\left(\begin{array}{c|c} Y_1 & Y_2 \end{array} \right), b)$: Cost: $O(b(b+p)(n-kb))$ flops during the kth iteration of the blocked algorithm, for a total of $O((b+p)n^2)$ flops. (Recall that the factorization of this matrix can stop after the first b columns have been identified.)

- $\cdots :=$ SWAPCOLS$(s_1, \cdots)$: Cost: ignored.

- $Y_2 := Y_2 - \left(G_1 - (G_1 U_{11} + G_2 U_{21}) T_{11}^{-1} {U_{11}}^H\right) R_{12}$: Aggregate cost: $O((b+p)n^2)$.

Thus, the overhead is $O((b+p)n^2)$ flops and the total cost is

$$4/3n^3 + O((b+p)n^2) \text{ flops},$$

which, asymptotically, approaches $4/3n^3$ flops. An analogous analysis of the cost for factoring a rectangular matrix shows that the asymptotic flop count is $2mn^2 - \frac{2}{3}n^3$ whenever $m \geq n$.

2.4 Experiments

This section describes the results from two sets of experiments. Section 2.4.1 compares the computational speed of the proposed scheme to existing state-of-the-art methods for computing column pivoted QR factorizations. Section 2.4.2 investigates how well the proposed randomized technique works at selecting pivot columns. Specifically, we investigate how well the rank-k truncated QR factorization approximates the original matrix, and compare the results to those obtained by classical column pivoting.

2.4.1 Performance experiments

We have implemented the proposed HQRRP algorithm using the `libflame` [125, 124] library that allows our implementations to closely resemble the algorithms as presented in the paper.

The implementation described is efficient, but is far from fully optimized. Our objective for now is to demonstrate that HQRRP is competitive and worthy of future study. We return to this point in the conclusion.

Platform details. All experiments reported in this article were performed on an Intel Xeon E5-2695 v3 (Haswell) processor (2.3 GHz), with 14 cores. In order to be able to show scalability results, the clock

speed was throttled at 2.3 GHz, turning off so-called turbo boost. Each core can execute 16 double precision floating point operations per cycle, meaning that the peak performance of one core is 36.8 GFLOPS (billions of floating point operations per second). For comparison, on a single core, `dgemm` achieves around 33.6 GFLOPS. Other details of interest include that the OS used was Linux (Version 2.6.32-504.el6.x86_64), the code was compiled with gcc (Version 4.4.7), `dgeqrf` and `dgeqp3` were taken from LAPACK (Release 3.4.0), and the implementations were linked to BLAS from Intel's MKL library (Version 11.2.3). Our implemetations were coded with `libflame` (Release 11104).

Implementations. We report performance for five implementations:

`dgeqrf`**.** The implementation of blocked HQR that is part of the `netlib` implementation of LAPACK, modified so that the block size can be controlled.

`dgeqp3`**.** The implementation of blocked HQRP that is part of the `netlib` implementation of LAPACK, based on [107], modified so that the block size can be controlled.

`HQRRPbasic`**.** Our implementation of HQRRP that computes new matrices G and Y in every iteration. This implementation deviates from the algorithm in Fig. 2.3 in that it also incorporates additional column pivoting within the call to HQRRP_UNB.

`HQRRP`**.** The implementation of HQRRP that downdates Y. (It also includes pivoting within HQRRP_UNB).

`dgeqpx`**.** An implementation of HQRP with window pivoting [14, 13], briefly mentioned in the introduction. This algorithm consists of two stages: The first stage is a QR with window pivoting. An incremental condition estimator is employed to accept/reject the columns within a window. The size of the window is about about twice the block size used by the algorithm to maintain locality. If all the columns in the window are unacceptable, all of them are rejected and fresh ones are brought into the window. The second stage is a postprocessing stage to validate the rank, that is, to check that all good columns are in the front, and all the bad columns are in the rear. This step is required because the window pivoting could fail to reveal the rank due to its shortsightedness (having only checked the window and having employed a cheap to compute condition estimator.) Sometimes,

some columns must be moved between R_{11} that has been computed and matrices R_{12} and R_{22}, and then retriangularization must be performed with Givens rotations.

In all cases, we used algorithmic block sizes of $b = 64$ and 128. While likely not optimal for all problem sizes, these blocks sizes yield near best performance and, regardless, it allows us to easily compare and contrast the performance of the different implementations.

Results. As is customary in these kinds of performance comparisons, we compute the achieved performance as

$$\frac{4/3n^3}{\text{time (in sec.)}} \times 10^{-9} \text{ GFLOPS}.$$

Thus, even for the implementations that perform more computations, we only count the floating point operations performed by an unblocked HQR without pivoting.

Figure 2.4 reports performance on one and four cores. The single core performance is very encouraging. Not only does `HQRRP` handily outperform `dgeqp3` and to a lesser degree `dgeqpx`, but asymptotically its performance approaches that of `dgeqrf`. For four cores, although the performance of all implementations asymptote much lower, the relative performance of the different implementations is consistent with that observed when utilizing one core. As more cores are added to the computation, the absolute speed of all algorithms tested drop substantially, but relatively speaking, the new algorithm `HQRRP` drops less than its competitors, as shown in Figure 2.1.

The bottom line is that for all these algorithms, much research is needed to improve parallel performance. We discuss this further in the conclusion.

2.4.2 Quality experiments

In this section, we describe the results of numerical experiments that were conducted to compare the quality of the pivot choices made by our randomized algorithm HQRRP to those resulting from classical column pivoting. Specifically, we compared how well partial factorizations reveal the numerical ranks of four different test matrices:

- *Matrix 1 (fast decay):* This is an $n \times n$ matrix of the form $A = UDV^*$ where U and V are randomly

drawn matrices with orthonormal columns (obtained by performing QR on a random Gaussian matrix), and where D is diagonal with entries given by $d_j = \beta^{(j-1)/(n-1)}$ with $\beta = 10^{-5}$.

- *Matrix 2 (S shaped decay):* This matrix is built in the same manner as "Matrix 1", but now the diagonal entries of D are chosen to first hover around 1, then decay rapidly, and then level out at 10^{-6}, as shown in Figure 2.6 (black line).

- *Matrix 3 (Single Layer BIE):* This matrix is the result of discretizing a Boundary Integral Equation (BIE) defined on a smooth closed curve in the plane. To be precise, we discretized the so called "single layer" operator associated with the Laplace equation using a 6^{th} order quadrature rule designed by Alpert [2]. This is a well-known ill-conditioned problem for which column pivoting is essential in order to stably solve the corresponding linear system.

- *Matrix 4 (Kahan):* This is a variation of the "Kahan counter-example" [76] which is designed to trip up classical column pivoting. The matrix is formed as $A = SK$ where:

$$S = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots \\ 0 & \zeta & 0 & 0 & \cdots \\ 0 & 0 & \zeta^2 & 0 & \cdots \\ 0 & 0 & 0 & \zeta^3 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix} \quad \text{and} \quad K = \begin{bmatrix} 1 & -\phi & -\phi & -\phi & \cdots \\ 0 & 1 & -\phi & -\phi & \cdots \\ 0 & 0 & 1 & -\phi & \cdots \\ 0 & 0 & 0 & 1 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

  for some positive parameters ζ and ϕ such that $\zeta^2 + \phi^2 = 1$. In our experiments, we chose $\zeta = 0.99999$.

For each test matrix, we computed QR factorizations

$$AP = QR \tag{2.8}$$

using three different techniques:

- `HQRP`: The standard QR factorization `qr` built in to Matlab R2015a.

- `HQRRPbasic`: The randomized algorithm described in Figure 2.3, but without the updating strategy for computing the sample matrix Y.

- `HQRRP`: The randomized algorithm described in Figure 2.3.

Our implementations of both `HQRRPbasic` and `HQRRP` deviate from what is shown in Figure 2.3 in that they also incorporate column pivoting within the call to HQRRP_UNB. In all experiments, we used test matrices of size $4\,000 \times 4\,000$, a block size of $b = 100$, and an over-sampling parameter of $p = 5$.

As a quality measure of the pivoting strategy, we computed the errors e_k incurred when the factorization is truncated to its first k components. To be precise, these residual errors are defined via

$$e_k = \|AP - Q(:, 1:k)R(1:k, :)\| = \|R((k+1):n, (k+1):n)\|. \tag{2.9}$$

The results are shown in Figures 2.5 – 2.8, for the four different test matrices. The black lines in the graphs show the theoretically minimal errors incurred by a rank-k approximation. These are provided by the Eckart-Young theorem [45] which states that, with $\{\sigma_j\}_{j=1}^{n}$ denoting the singular values of A:

$$\begin{aligned} e_k &\geq \sigma_{k+1} && \text{when errors are measured in the spectral norm, and} \\ e_k &\geq \left(\textstyle\sum_{j=k+1}^{n} \sigma_j\right)^{1/2} && \text{when errors are measured in the Frobenius norm.} \end{aligned}$$

We observe in all cases that the quality of the pivots chosen by the randomized method very closely matches those resulting from classical column pivoting. The one exception is the Kahan counter-example ("Matrix 4"), where the randomized algorithm performs much better. (The importance of the last point should not be over-emphasized since this example is designed specifically to be adversarial for classical column pivoting.)

When classical column pivoting is used, the factorization (2.8) produced has the property that the diagonal entries of R are strictly decaying in magnitude

$$|R(1,1)| \geq |R(2,2)| \geq |R(3,3)| \geq \cdots$$

When the randomized pivoting strategies are used, this property is not enforced. To illustrate this point, we show in Figure 2.9 the values of the diagonal entries obtained by the randomized strategies versus what is obtained with classical column pivoting.

2.5 Conclusions and future work

We have described the algorithm HQRRP which is a blocked version of Householder QR with column pivoting. The main innovation compared to earlier work is that pivots are determined in groups using a technique based on randomized projections. We demonstrated that the quality of the chosen pivots is for practical purposes indistinguishable from traditional column pivoting (cf. Figures 2.5 – 2.7), and that the dominant term in the asymptotic flop count equals that of non-pivoted QR. Importantly, we also demonstrate through numerical experiments that HQRRP is very fast, in fact almost as fast as unpivoted HQR.

The technique described opens up several potential avenues for future research. The speed gains we demonstrate on single core and shared memory multicore machines is due to the reduction in data movement. Equivalently, data moved between memory layers is carefully amortized. We expect the technique described to have an even more pronounced advantage over traditional column pivoted QR when implemented in more severely communication constrained environments such as a matrix processed on a GPU or a distributed memory parallel machine, or stored out-of-core.

The randomized sampling techniques we describe can also be used to construct very close to optimal rank-revealing factorizations. To describe the idea, we note that a column pivoted QR factorization of a given matrix A can be written as

$$A = Q\,R\,P^*, \tag{2.10}$$

where Q is orthonormal, R is upper triangular, and P is a permutation matrix. In this manuscript, we used randomized sampling to determine the permutation matrix P. It turns out that for a modest additional cost, one can build a factorization that takes the form (2.10), but with both Q and P built as products of Householder reflectors. This generalization allows us to bring R not only to upper triangular form, but very close to being diagonal, with accurate approximations to the singular values of A on its diagonal. This discovery was reported in [93], and is a subject of ongoing research.

How to scale the presented algorithm to very large numbers of cores is an open research question. Techniques such as "compute ahead" will have to be employed to ensure that the factorization of the current panel (A_{11} and A_{21}) and downdate of Y do not start dominating the parts of the computation that can be

cast in terms of GEMM.

2.6 Software

A number of implementations of the discussed algorithm are available under 3-clause (modified) BSD license from:

`https://github.com/flame/hqrrp`

Included are implementations that directly link to LAPACK [4] as well as implementations that use the libflame [104, 65] library. For those who use the LAPACK routine dgeqp3 routine, a plug compatible routine dgeqp4 is provided.

A distributed memory implementation of the algorithm has been incorporated into the Elemental software package by Jack Poulson et al [103], available at:

`https://github.com/elemental/Elemental`

Acknowledgments:

The research reported was supported by DARPA, under the contract N66001-13-1-4050, and by the NSF, under the contracts DMS-1407340 and ACI-1148125/1340293. *Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).*

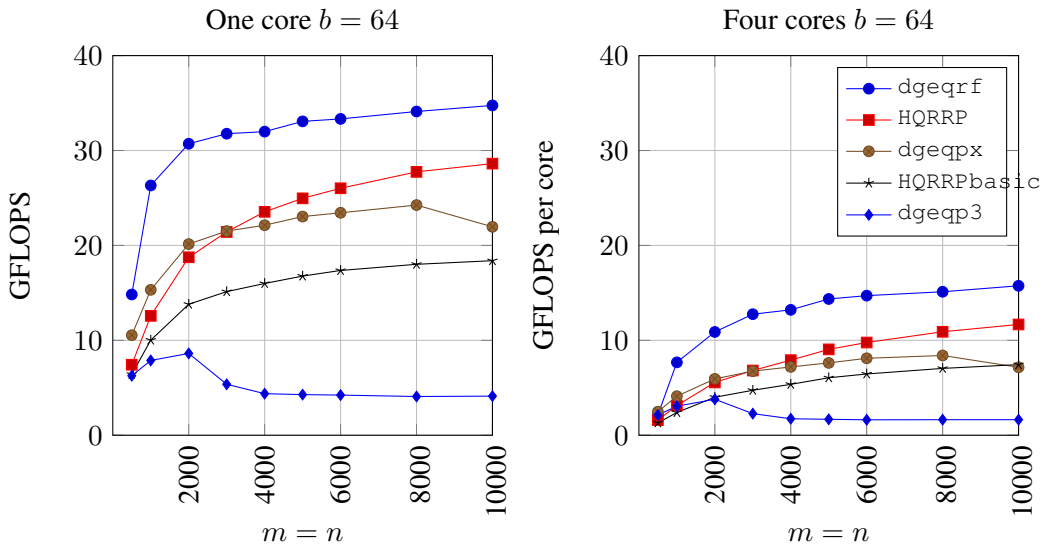


Figure 2.4: Performance of the various implementations. There is a substantial degradation in performance from one core to four. However, what is important is that the relative performance of the different algorithms remains essentially the same. This pattern continues as more cores are added, as shown in Figure 2.1.

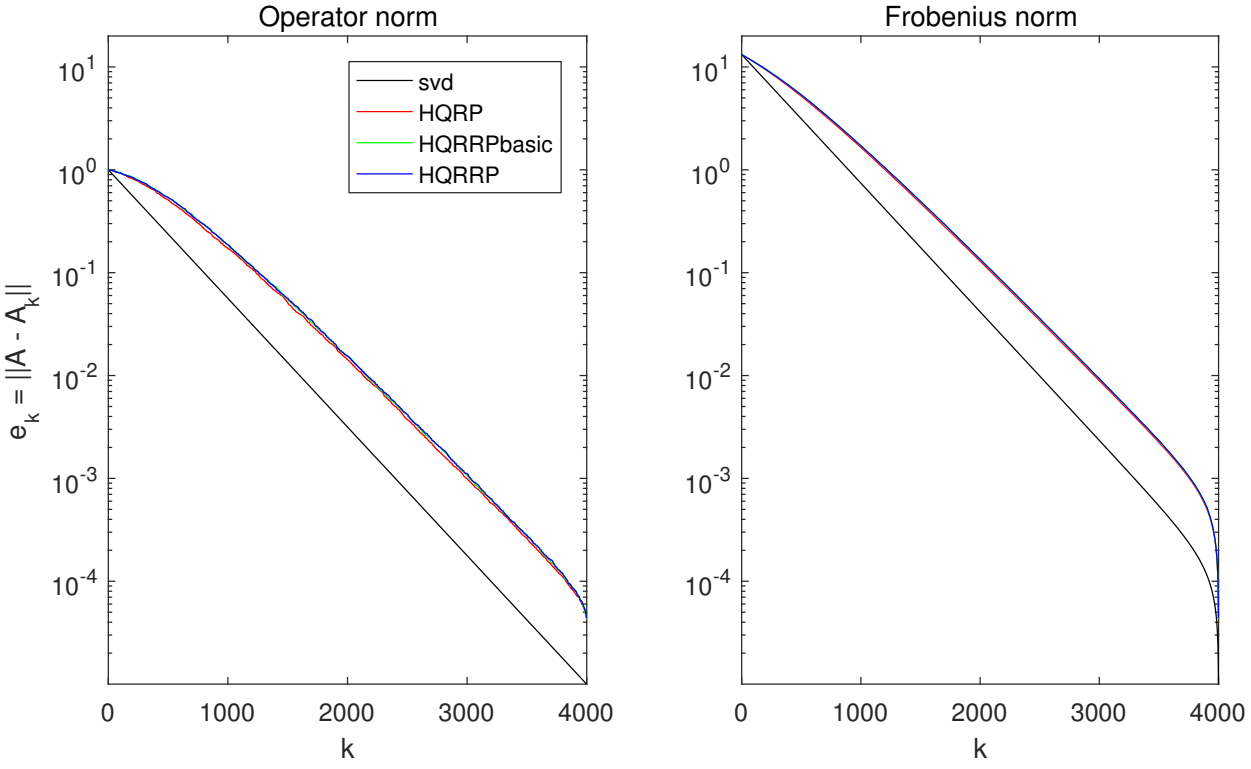


Figure 2.5: Residual errors e_k for "Matrix 1" as a function of the truncation rank k, cf. (2.9). The red line shows the results from traditional column pivoting, while the green and blue lines refer to the randomized methods we propose. The black line indicates the theoretically minimal errors resulting from a rank-k partial singular value decomposition.

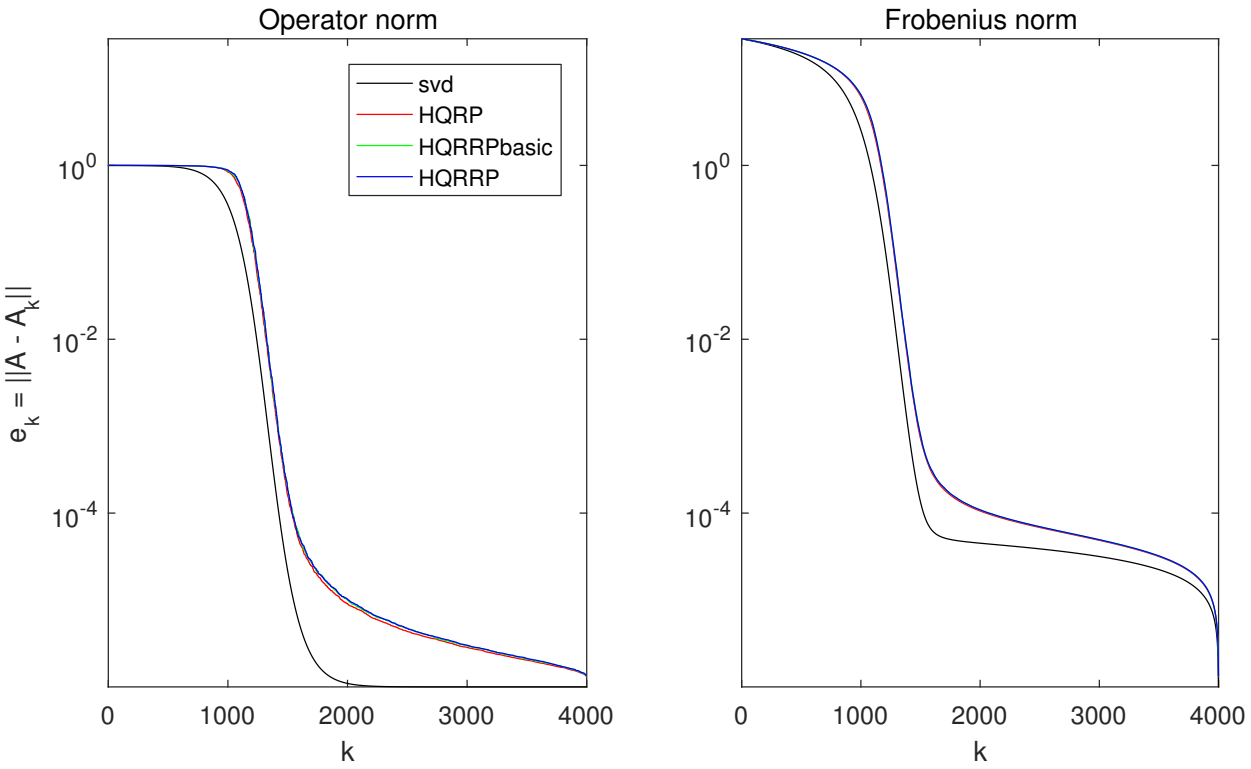


Figure 2.6: Residual errors e_k for "Matrix 2" as a function of the truncation rank k. Notation is the same as in Figure 2.5.

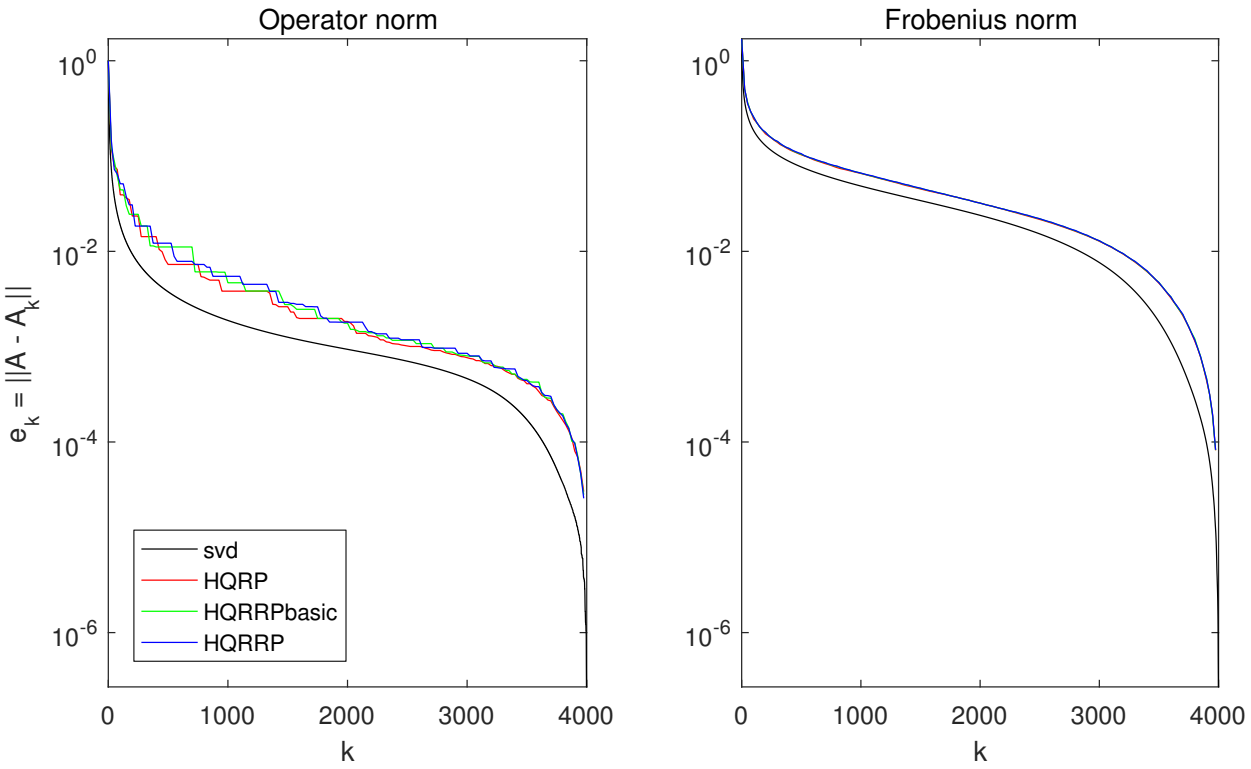


Figure 2.7: Residual errors e_k for "Matrix 3" (discretized boundary integral operator) as a function of the truncation rank k. Notation is the same as in Figure 2.5.

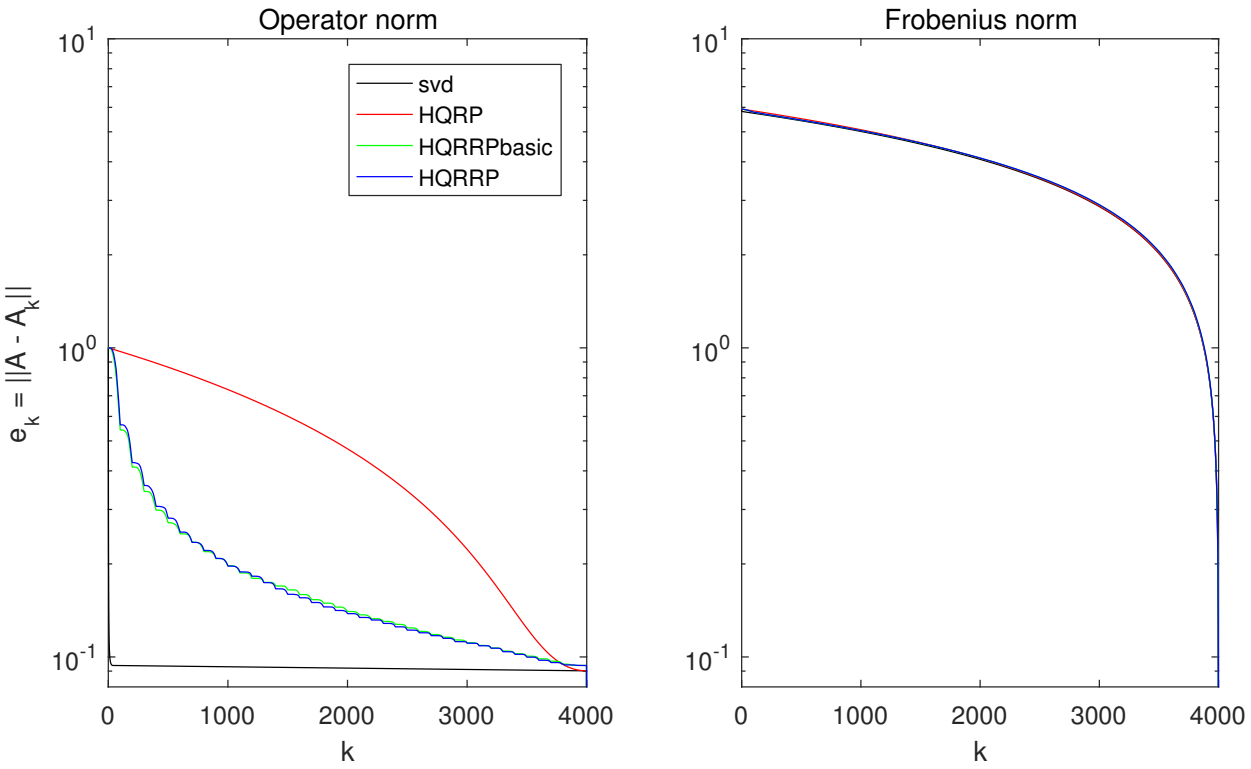


Figure 2.8: Residual errors e_k for "Matrix 4" (Kahan) as a function of the truncation rank k. Notation is the same as in Figure 2.5.

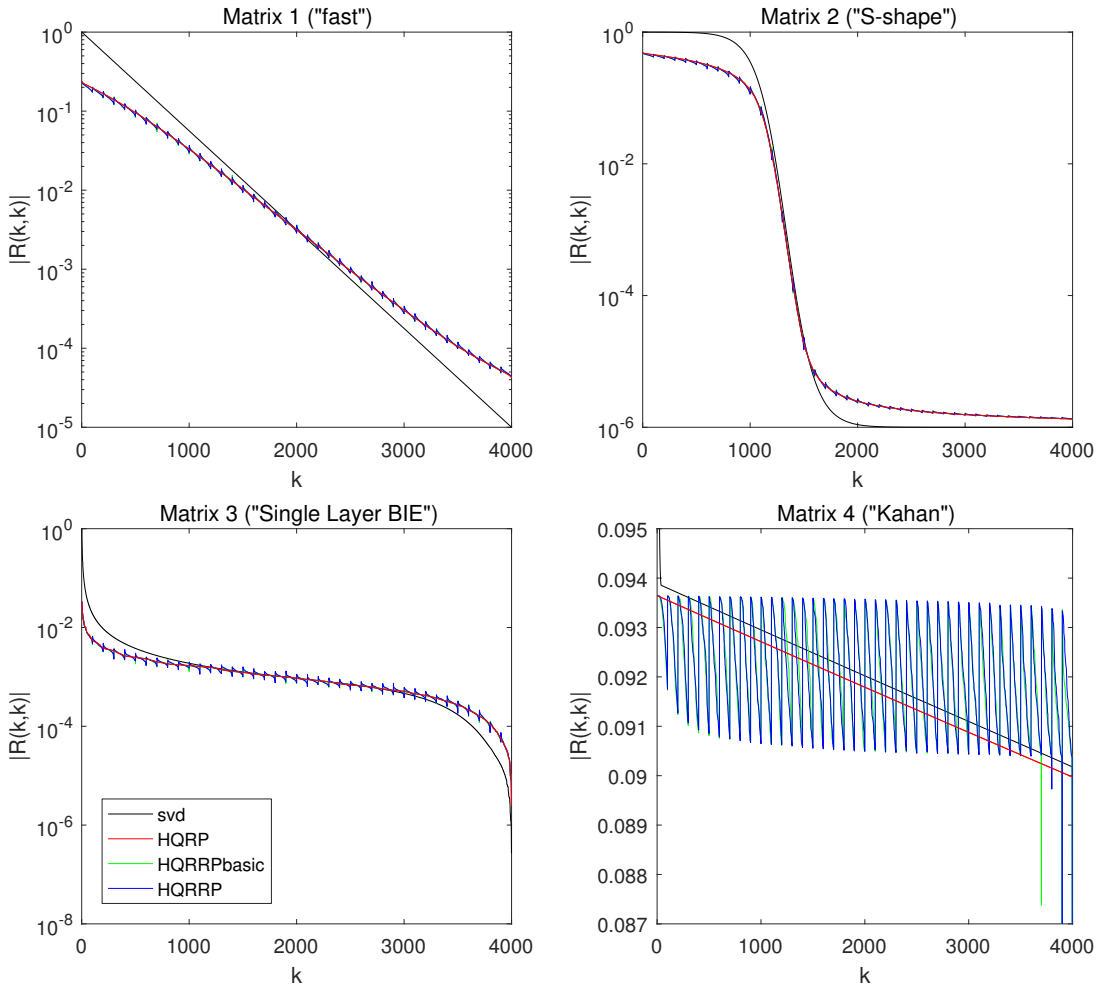


Figure 2.9: For each of the four test matrices described in Section 2.4.2, we show the magnitudes of the diagonal entries in the "R"-factor in a column pivoted QR factorization. We compare classical column pivoting (red) with the two randomized techniques proposed here (blue and green). We also show the singular values of each matrix (black) for reference.

Chapter 3

randUTV: A blocked randomized algorithm for computing a rank-revealing UTV factorization

A randomized algorithm for computing a so called UTV factorization efficiently is presented. Given a matrix $\mathbf{A}$, the algorithm "randUTV" computes a factorization $\mathbf{A} = \mathbf{U}\mathbf{T}\mathbf{V}^*$, where $\mathbf{U}$ and $\mathbf{V}$ have orthonormal columns, and $\mathbf{T}$ is triangular (either upper or lower, whichever is preferred). The algorithm randUTV is developed primarily to be a fast and easily parallelized alternative to algorithms for computing the Singular Value Decomposition (SVD). randUTV provides accuracy very close to that of the SVD for problems such as low-rank approximation, solving ill-conditioned linear systems, determining bases for various subspaces associated with the matrix, etc. Moreover, randUTV produces highly accurate approximations to the singular values of $\mathbf{A}$. Unlike the SVD, the randomized algorithm proposed builds a UTV factorization in an incremental, single-stage, and non-iterative way, making it possible to halt the factorization process once a specified tolerance has been met. Numerical experiments comparing the accuracy and speed of randUTV to the SVD are presented. Other experiments also demonstrate that in comparison to column-pivoted QR, which is another factorization that is often used as a relatively economic alternative to the SVD, randUTV compares favorably in terms of speed while providing far higher accuracy.

3.1 Introduction

3.1.1 Overview

Given an $m \times n$ matrix $\mathbf{A}$, the so called "UTV decomposition" [116, p. 400] takes the form

$$\begin{array}{ccccc} \mathbf{A} & = & \mathbf{U} & \mathbf{T} & \mathbf{V}^*, \\ m \times n & & m \times m & m \times n & n \times n \end{array} \tag{3.1}$$

where $\mathbf{U}$ and $\mathbf{V}$ are unitary matrices, and $\mathbf{T}$ is a triangular matrix (either lower or upper triangular). The UTV decomposition can be viewed as a generalization of other standard factorizations such as, e.g., the *Singular Value Decomposition (SVD)* or the *Column Pivoted QR decomposition (CPQR).* (To be precise, the SVD is the special case where $\mathbf{T}$ is diagonal, and the CPQR is the special case where $\mathbf{V}$ is a permutation matrix.) The additional flexibility inherent in the UTV decomposition enables the design of efficient updating procedures, see [116, Ch. 5, Sec. 4] and [50, 119, 102]. In this manuscript, we describe a randomized algorithm we call randUTV that exploits the additional flexibility provided by the UTV format to build a factorization algorithm that combines some of the most desirable properties of standard algorithms for computing the SVD and CPQR factorizations.

3.1.2 Where randUTV fits in the pantheon of matrix factorization algorithms

The algorithm we describe is designed primarily to serve as a competitive alternative to the column pivoted QR factorization for applications such as detecting the numerical rank of a matrix, solving ill-conditioned linear systems, computing least-squares solutions to linear systems with rectangular or numerically rank-deficient coefficient matrices, etc. The optimal factorization to use for many of these tasks is the SVD, but the CPQR remains popular since it is faster to compute, and has a crucial advantage in that the factorization can be halted once a specified tolerance has been met.

We argue that randUTV outperforms CPQR in almost every regard. It does a much better job than CPQR of revealing the rank-structure of the underlying matrix, which makes it a better tool for revealing the numerical rank, and for solving ill-conditioned linear systems. It is often faster to compute than the CPQR, and parallelizes better as the number of cores is increased. Finally, it shares the key advantage of the CPQR

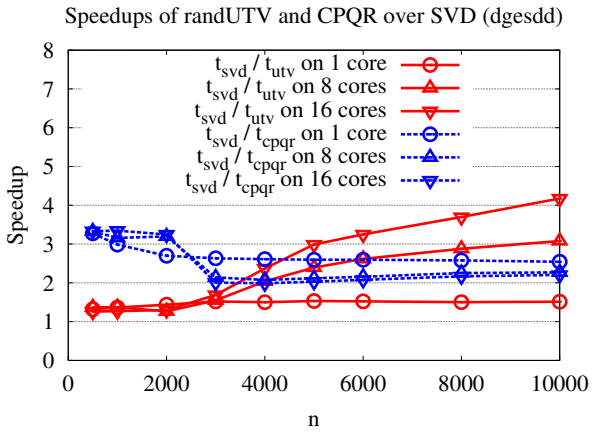


Figure 3.1: Solid lines show the acceleration of randUTV with the parameter choice $q = 1$ (cf. Section 3.4.3) compared to Intel's MKL SVD (`dgesdd`). Dashed lines show the analogous acceleration of CPQR over SVD.

that the factorization can be halted once a specified tolerance has beet met.

The reason for the computational efficiency of randUTV is that it can (unlike the CPQR) be *blocked*, and can be implemented so that the vast majority of flops are spent in dongarra1990set operations, rather than in BLAS2 operations. We will demonstrate that this leads to high performance on a single core, and excellent performance as the number of cores is increased. We provide detailed numerical experiments in Section 3.6.1, but the key findings are summarized in Figure 3.1. We see that randUTV is much faster than state-of-the-art functions for computing the SVD, and that it is often faster than the CPQR, in particular as the number of cores increases. Since randUTV can be blocked, it is also very well suited for implementation on distributed memory machines, for matrices stored out-of-core, etc.

Remark 5 (randUTV as an alternative to the SVD)**.** *The method we propose is sufficiently accurate that it can sometimes be used as an economical method for computing an approximate singular value decomposition. The numerical experiments in Section 3.6.2 indicate that the matrix* $\mathbf{T}$ *often approximates the diagonal matrix* $\mathbf{D}$ *in the SVD to two or three digits of accuracy or more. This level of accuracy is often sufficient for purposes of data analysis, and we believe that randUTV provides an excellent alternative to the SVD for applications such as, e.g., Principal Component Analysis (PCA). However, there of course remain many applications where computing the SVD using existing methodologies remains the superior choice, including*

situations where the actual singular values are required to high accuracy, where the singular values of the matrix decay very slowly, or simply situations where the matrix is small enough that computing the full SVD is very fast.

Remark 6 (Where CPQR is better than randUTV)**.** *Our numerical experiments indicate that for small matrices, our implementation of randUTV is at the current level of optimization slower than the best implementations of LAPACK's* `dgeqp3` *function [107] (see also Figure 3.1). Moreover, the CPQR provides important information about how to pick subsets of the columns of a matrix to use as a basis for its column space. This makes the CPQR an important component in algorithms for computing so called interpolatory and CUR decompositions [94, 128].*

3.1.3 A randomized algorithm for computing the UTV decomposition

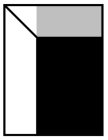
The algorithm we propose is blocked for computational efficiency. For concreteness, let us assume that $m \geq n$, and that an *upper* triangular factor $\mathbf{T}$ is sought. With b a block size, randUTV proceeds through $\lceil n/b \rceil$ steps, where at the i'th step the i'th block of b columns of $\mathbf{A}$ is driven to upper triangular form, as illustrated in Figure 3.2.

In the first iteration, randUTV uses a randomized subspace iteration inspired by [111, 69] to build two sets of b orthonormal vectors that approximately span the spaces spanned by the b dominant left and right singular vectors of $\mathbf{A}$, respectively. These basis vectors form the first b columns of two unitary "transformation matrices" $\mathbf{U}^{(1)}$ and $\mathbf{V}^{(1)}$. We use these to build a new matrix

$$\mathbf{A}^{(1)} = \left(\mathbf{U}^{(1)}\right)^* \mathbf{A}\, \mathbf{V}^{(1)}$$

that has a blocked structure as follows:

$$\mathbf{A}^{(1)} = \begin{bmatrix} \mathbf{A}_{11}^{(1)} & \mathbf{A}_{12}^{(1)} \\ \mathbf{0} & \mathbf{A}_{22}^{(1)} \end{bmatrix} = \quad .$$



In other words, the top left $b \times b$ block is diagonal, and the bottom left block is zero. Ideally, we would want the top right block to also be zero, and this is what we would obtain if we used the exact left and right singular

vectors in the transformation. The randomized sampling does not exactly identify the correct subspaces, but it does to high enough accuracy that the elements in the top right block have very small moduli. For purposes of low-rank approximation, such a format strikes an attractive balance between computational efficiency and close to optimal accuracy. We will demonstrate that $\|\mathbf{A}_{22}^{(1)}\| \approx \inf\{\|\mathbf{A} - \mathbf{B}\| : \mathbf{B} \text{ has rank } b\}$, and that the diagonal entries of $\mathbf{A}_{11}^{(1)}$ form accurate approximations to the first b singular values of $\mathbf{A}$.

Once the first step has been completed, the second step applies the same procedure to the remaining block $\mathbf{A}_{22}^{(1)}$, which has size $(m-b) \times (n-b)$, and then continues in the same fashion to process all remaining blocks.

3.1.4 Relationship to earlier work

The UTV factorization was introduced and popularized by G.W. Stewart in a sequence of papers, including [114, 115, 119, 120], and the text book chapter [116, p. 400]. Among these, [120] is of particular relevance as it discusses explicitly how the UTV decomposition can be used for low-rank approximation and for finding approximations to the singular values of a matrix; in Section 3.6.2 we compare the accuracy of the method in [120] to the accuracy of randUTV. Of relevance here is also [96], which describes deterministic iterative methods for driving a triangular matrix towards diagonality. Another path towards better rank revelation and more accurate rank estimation is described in [49]. A key advantage of the UTV factorization is the ease with which it can be updated, as discussed in, e.g., [7, 6, 102]. Implementation aspects are discussed in [50].

The work presented here relies crucially on previous work on randomized subspace iteration for approximating the linear spaces spanned by groups of singular vectors, including [111, 69, 91, 90, 68]. This prior body of literature focused on the task of computing *partial* factorizations of matrices. More recently, it was observed [93, 88, 44] that analogous techniques can be utilized to accelerate methods for computing *full* factorizations such as the CPQR. The gain in speed is attained from *blocking* of the algorithms and the use of dongarra1990set, rather than by reducing the asymptotic flop count. Our work is also related to a randomized algorithm for computing a UTV decomposition described in Section 5 of [35]. The algorithm in [35] relies on the observation that if an $n \times n$ matrix $\mathbf{V}$ is drawn from a Haar distribution, then the unpivoted

QR factorization of $\mathbf{AV}$ is with high probability "rank-revealing" in a certain sense. This algorithm is conceptually very similar to randUTV for the special case where no over-sampling is done, and no steps of power iteration are taken. This means that the algorithm of [35] does not provide a path to improve upon a factorization beyond what the most basic randomized sampling produces, which limits it usefulness in many practical application. (Numerical results in Section 3.6.2 illustrate that the use of power iteration is often imperative.)

3.1.5 Outline of the paper

Section 3.2 introduces notation, and lists some relevant existing results that we need. Section 3.3 provides a high-level description of the proposed algorithm. Section 3.4 describes in detail how to drive one block of columns to upper triangular form, which forms one step of the blocked algorithm. Section 3.5 describes the whole multistep algorithm, discusses some implementation issues, and provides an estimate of the asymptotic flop count. Section 3.6 gives the results of several numerical experiments that illustrate the speed and accuracy of the proposed method. Section 3.7 describes availability of software.

3.2 Preliminaries

This section introduces our notation, and reviews some established techniques that will be needed. The material in Sections 3.2.1–3.2.4 is described in any standard text on numerical linear algebra (e.g. [57, 116, 122]). The material in Section 3.2.5 on randomized algorithms is described in further detail in the survey [69] and the lecture notes [87].

3.2.1 Basic notation

Throughout this manuscript, we measure vectors in $\mathbb{R}^n$ using their Euclidean norm. The default norm for matrices will be the corresponding operator norm $\|\mathbf{A}\| = \sup\{\|\mathbf{Ax}\| : \|\mathbf{x}\| = 1\}$, although we will sometimes also use the Frobenius norm $\|\mathbf{A}\|_{\text{Fro}} = \left(\sum_{i,j} |\mathbf{A}(i,j)|^2\right)^{1/2}$. We use the notation of Golub and Van Loan [57] to specify submatrices: If $\mathbf{B}$ is an $m \times n$ matrix, and $I = [i_1, i_2, \ldots, i_k]$ and $J = [j_1, j_2, \ldots, j_\ell]$ are index vectors, then $\mathbf{B}(I, J)$ denotes the corresponding $k \times \ell$ submatrix. We let $\mathbf{B}(I, :)$

denote the matrix $\mathbf{B}(I, [1,\ 2,\ \ldots,\ n])$, and define $\mathbf{B}(:, J)$ analogously. $\mathbf{I}_n$ denotes the $n \times n$ identity matrix, and $\mathbf{0}_{m,n}$ is the $m \times n$ zero matrix. The transpose of $\mathbf{B}$ is denoted by $\mathbf{B}^*$, and we say that an $m \times n$ matrix $\mathbf{U}$ is *orthonormal* if its columns are orthonormal, so that $\mathbf{U}^*\mathbf{U} = \mathbf{I}_n$. A square orthonormal matrix is said to be *unitary.*

3.2.2 The Singular Value Decomposition (SVD)

Let $\mathbf{A}$ be an $m \times n$ matrix and set $r = \min(m, n)$. Then the SVD of $\mathbf{A}$ takes the form

$$\begin{array}{ccccc} \mathbf{A} & = & \mathbf{U} & \mathbf{D} & \mathbf{V}^*, \\ m \times n & & m \times r & r \times r & r \times n \end{array} \tag{3.2}$$

where matrices $\mathbf{U}$ and $\mathbf{V}$ are orthonormal, and $\mathbf{D}$ is diagonal. We have $\mathbf{U} = \begin{bmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \ldots & \mathbf{u}_r \end{bmatrix}$, $\mathbf{V} = \begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \ldots & \mathbf{v}_r \end{bmatrix}$, and $\mathbf{D} = \mathrm{diag}(\sigma_1, \sigma_2, \ldots, \sigma_r)$, where $\{\mathbf{u}_j\}_{j=1}^r$ and $\{\mathbf{v}_j\}_{j=1}^r$ are the left and right singular vectors of $\mathbf{A}$, respectively, and $\{\sigma_j\}_{j=1}^r$ are the singular values of $\mathbf{A}$. The singular values are ordered so that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r \geq 0$.

A principal advantage of the SVD is that it furnishes an explicit solution to the low rank approximation problem. To be precise, for $k = 1, 2, \ldots, r$ let us define the rank-k matrix

$$\mathbf{A}_k = \mathbf{U}(:, 1:k)\, \mathbf{D}(1:k, 1:k)\, \mathbf{V}(:, 1:k)^* = \sum_{j=1}^{k} \sigma_j\, \mathbf{u}_j\, \mathbf{v}_j^*.$$

Then, the Eckart-Young theorem asserts that

$$\|\mathbf{A} - \mathbf{A}_k\| = \inf\{\|\mathbf{A} - \mathbf{B}\| :\ \mathbf{B} \text{ has rank } k\}.$$

A disadvantage of the SVD is that the cost to compute the full SVD is $O(n^3)$ for a square matrix, and $O(mnr)$ for a rectangular matrix, with large pre-factors. Moreover, standard techniques for computing the SVD are challenging to parallelize, and cannot readily be modified to compute partial factorizations.

3.2.3 The Column Pivoted QR (CPQR) decomposition

Let $\mathbf{A}$ be an $m \times n$ matrix and set $r = \min(m, n)$. Then, the CPQR decomposition of $\mathbf{A}$ takes the form

$$\begin{array}{ccccc} \mathbf{A} & = & \mathbf{Q} & \mathbf{R} & \mathbf{P}^*, \\ m \times n & & m \times r & r \times n & n \times n \end{array} \tag{3.3}$$

where $\mathbf{Q}$ is orthonormal, $\mathbf{R}$ is upper triangular, and $\mathbf{P}$ is a permutation matrix. The permutation matrix $\mathbf{P}$ is typically chosen to ensure monotonic decay in magnitude of the diagonal entries of $\mathbf{R}$ so that $|\mathbf{R}(1,1)| \geq |\mathbf{R}(2,2)| \geq |\mathbf{R}(3,3)| \geq \cdots$. The factorization (3.3) is commonly computed using the *Householder QR algorithm* [57, Sec. 5.2], which is exceptionally stable.

An advantage of the CPQR is that it is computed via an incremental algorithm that can be halted to produce a partial factorization once any given tolerance has been met. A disadvantage is that it is not ideal for low-rank approximation. In the typical case, the error incurred is noticeably worse than what you get from the SVD but usually not disastrously so. However, there exist matrices for which CPQR leads to very suboptimal approximation errors [76]. Specialized pivoting strategies have been developed that can in some circumstances improve the low-rank approximation property, resulting in so called "rank-revealing QR factorizations" [24, 64].

The classical column pivoted Householder QR factorization algorithm drives the matrix $\mathbf{A}$ to upper triangular form via a sequence of $r - 1$ rank-one updates and column pivotings. Due to the column pivoting performed after each update, it is difficult to *block*, making it hard to achieve high computational efficiency on modern processors [40]. It has recently been demonstrated that randomized methods can be used to resolve this difficulty [93, 88, 44]. However, we do not yet have rigorous theory backing up such randomized techniques, and they have not yet been incorporated into standard software packages.

3.2.4 Efficient factorizations of tall and thin matrices

The algorithms proposed in this manuscript rely crucially on the fact that factorizations of "tall and thin" matrices can be computed efficiently. To be precise, suppose that we are given a matrix $\mathbf{B}$ of size

$m \times b$, where $m \gg b$, and that we seek to compute an *unpivoted* QR factorization

$$\begin{array}{ccccc} \mathbf{B} & = & \mathbb{Q} & & \mathbf{R}, \\ m \times b & & m \times m & & m \times b \end{array} \tag{3.4}$$

where $\mathbb{Q}$ is unitary, and $\mathbf{R}$ is upper triangular. When the Householder QR factorization procedure is used to compute the factorization (3.4), the matrix $\mathbb{Q}$ is formed as a product of b so called "Householder reflectors" [57, Sec. 5.2], and can be written in the form

$$\begin{array}{ccccccc} \mathbb{Q} & = & \mathbf{I} & + & \mathbf{W} & \mathbf{Y}, \\ m \times m & & m \times m & & m \times b & b \times m \end{array} \tag{3.5}$$

for some matrices $\mathbf{W}$ and $\mathbf{Y}$ that can be readily computed given the b Householder vectors that are formed in the QR factorization procedure [15, 113, 74]. As a consequence, we need only $O(mb)$ elements to store $\mathbb{Q}$, and we can apply $\mathbb{Q}$ to an $m \times n$ matrix using $\sim 2mnb$ flops. In this manuscript, the different typeface in $\mathbb{Q}$ is used as a reminder that this is a matrix that can be stored and applied efficiently, without being explicitely built.

Next, suppose that we seek to compute the SVD of the tall and thin matrix $\mathbf{B}$. This can be done efficiently in a two-step procedure, where the first step is to compute the unpivoted QR factorization (3.4). Let $\mathbf{R}_{\text{small}}$ denote the top $b \times b$ block of $\mathbf{R}$ so that

$$\mathbf{R} = \begin{bmatrix} \mathbf{R}_{\text{small}} \\ \mathbf{0}_{m-b,b} \end{bmatrix}. \tag{3.6}$$

Then, compute the SVD of $\mathbf{R}_{\text{small}}$ to obtain the factorization

$$\begin{array}{ccccccc} \mathbf{R}_{\text{small}} & = & \mathbf{U}_{\text{small}} & \mathbf{D}_{\text{small}} & \mathbf{V}^*_{\text{small}}. \\ b \times b & & b \times b & b \times b & b \times b \end{array} \tag{3.7}$$

Combining (3.4), (3.6), and (3.7), we obtain the factorization

$$\begin{array}{ccccccc} \mathbf{B} & = & \mathbb{Q} & \begin{bmatrix} \mathbf{U}_{\text{small}} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{m-b} \end{bmatrix} & \begin{bmatrix} \mathbf{D}_{\text{small}} \\ \mathbf{0}_{m-b,b} \end{bmatrix} & \mathbf{V}^*_{\text{small}}, \\ m \times b & & m \times m & m \times m & m \times b & b \times b \end{array} \tag{3.8}$$

which we recognize as a singular value decomposition of $\mathbf{B}$. Observe that the cost of computing this factorization is $O(mb^2)$, and that only $O(mb)$ stored elements are required, despite the fact that the matrix of left singular vectors is ostensibly an $m \times m$ dense matrix.

Remark 7. *We mentioned in Section 3.2.3 that it is challenging to achieve high performance when implementing column pivoted QR on modern processors. In contrast, unpivoted QR is highly efficient, since this algorithm can readily be blocked (see, e.g., Figures 4.1 and 4.2 in [88]).*

3.2.5 Randomized power iterations

This section summarizes key results of [111, 69, 87] on randomized algorithms for constructing sets of orthonormal vectors that approximately span the row or column spaces of a given matrix. To be precise, let $\mathbf{A}$ be an $m \times n$ matrix, let b be an integer such that $1 \leq b < \min(m, n)$, and suppose that we seek to find an $n \times b$ orthonormal matrix $\mathbf{Q}$ such that

$$\|\mathbf{A} - \mathbf{A}\mathbf{Q}\mathbf{Q}^*\| \approx \inf\{\|\mathbf{A} - \mathbf{B}\| : \mathbf{B} \text{ has rank } b\}.$$

Informally, the columns of $\mathbf{Q}$ should approximately span the same space as the dominant b right singular vectors of $\mathbf{A}$. This is a task that is very well suited for subspace iteration (see [36, Sec. 4.4.3] and [8]), in particular when the starting matrix is a Gaussian random matrix [111, 69]. With q a (small) integer denoting the number of steps taken in the power iteration, the following algorithm leads to close to optimal results:

1. Draw a Gaussian random matrix $\mathbf{G}$ of size $m \times b$.
2. Form a "sampling matrix" $\mathbf{Y}$ of size $n \times b$ via $\mathbf{Y} = \left(\mathbf{A}^*\mathbf{A}\right)^q \mathbf{A}^*\mathbf{G}$.
3. Orthonormalize the columns of $\mathbf{Y}$ to form the matrix $\mathbf{Q}$.

Observe that in step (2), the matrix $\mathbf{Y}$ is computed via alternating application of $\mathbf{A}^*$ and $\mathbf{A}$ to a tall thin matrix with b columns. In some situations, orthonormalization is required between applications to avoid loss of accuracy due to floating point arithmetic. In [111, 69] it is demonstrated that by using a Gaussian random matrix as the starting point, it is often sufficient to take just a few steps of power iteration, say $q = 1$ or 2, or even $q = 0$.

Remark 8 (Over-sampling)**.** *In the analysis of power iteration with a Gaussian random matrix as the starting point, it is common to draw a few "extra" samples. In other words, one picks a small integer p representing the amount of over-sampling done, say $p = 5$ or $p = 10$, and starts with a Gaussian matrix of size $m \times (b + p)$. This results in an orthonormal matrix $\mathbf{Q}$ of size $n \times (b + p)$. Then, with probability almost 1, the error $\|\mathbf{A} - \mathbf{AQQ}^*\|$ is close to the minimal error in rank-b approximation in both spectral and Frobenius norm [69, Sec. 10]. When no over-sampling is used, one risks losing some accuracy in the last couple of modes of the singular value decomposition. However, our experience shows that in the context of the present article, this loss is hardly noticeable.*

Remark 9 (RSVD)**.** *The randomized range finder described in this section is simply the first stage in the two-stage "Randomized SVD (RSVD)" procedure for computing an approximate rank-b SVD of a given matrix $\mathbf{A}$ of size $m \times n$. To wit, suppose that we have used the randomized range finder to build an orthonormal matrix $\mathbf{Q}$ of size $n \times b$ such that $\mathbf{A} = \mathbf{AQQ}^* + \mathbf{E}$, for some some error matrix $\mathbf{E}$. Then, we can compute an approximate factorization*

$$\begin{array}{ccccccc} \mathbf{A} & = & \mathbf{U} & \mathbf{D} & \mathbf{V}^* & + & \mathbf{E}, \\ m \times n & & m \times b & b \times b & b \times n & & m \times n \end{array} \tag{3.9}$$

where $\mathbf{U}$ and $\mathbf{V}$ are orthonormal, and $\mathbf{D}$ is diagonal, via the following steps (which together form the "second stage" of RSVD): (1) Set $\mathbf{B} = \mathbf{AQ}$ so that $\mathbf{AQQ}^* = \mathbf{BQ}^*$. (2) Compute a full SVD of the small matrix $\mathbf{B}$ so that

$$\begin{array}{ccccccc} \mathbf{B} & = & \mathbf{U} & \mathbf{D} & \hat{\mathbf{V}}^*. \\ m \times b & & m \times b & b \times b & b \times b \end{array} \tag{3.10}$$

(3) Set $\mathbf{V} = \mathbf{Q}\hat{\mathbf{V}}$. Observe that these last three steps are exact up to floating point arithmetic, so the error in (3.9) is exactly the same as the error in the range finder alone: $\mathbf{E} = \mathbf{A} - \mathbf{AQQ}^* = \mathbf{A} - \mathbf{UDV}^*$.

3.3 An overview of the randomized UTV factorization algorithm

This section describes at a high level the overall template of the algorithm randUTV that given an $m \times n$ matrix $\mathbf{A}$ computes its UTV decomposition (3.1). For simplicity, we assume that $m \geq n$, that an upper triangular middle factor $\mathbf{T}$ is sought, and that we work with a block size b that evenly divides n so that

the matrix $\mathbf{A}$ can be partitioned into s blocks of b columns each; in other words, we assume that $n = sb$. The algorithm randUTV iterates over s steps, where at the i'th step the i'th block of b columns is driven to upper triangular form via the application of unitary transformations from the left and the right. A cartoon of the process is given in Figure 3.2.

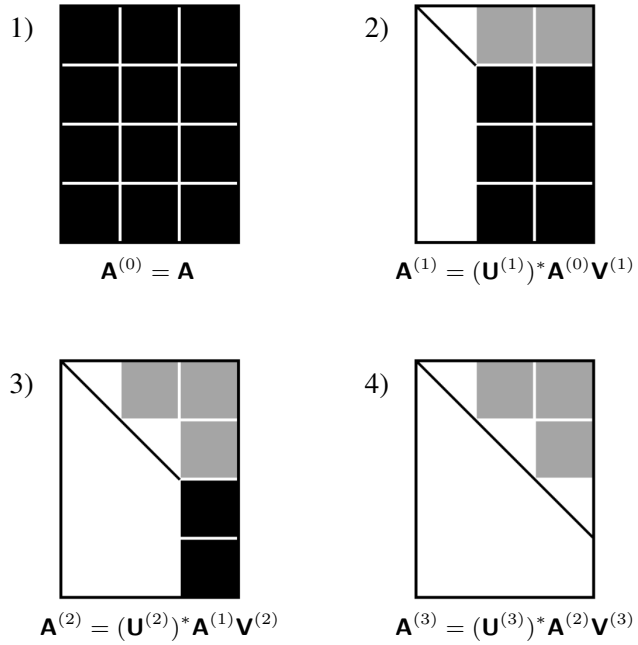


Figure 3.2: Cartoon illustrating the process by which a given matrix $\mathbf{A}$ is driven to block upper triangular form, with most of the mass concentrated into the diagonal entries. With $\mathbf{A}^{(0)} = \mathbf{A}$, we form a sequence of matrices $\mathbf{A}^{(i)} = (\mathbf{U}^{(i)})^* \mathbf{A}^{(i-1)} \mathbf{V}^{(i)}$ by applying unitary matrices $\mathbf{U}^{(i)}$ and $\mathbf{V}^{(i)}$ from the left and the right. Each $\mathbf{U}^{(i)}$ and $\mathbf{V}^{(i)}$ consists predominantly of a product of b Householder reflectors, where b is the block size. Black blocks represent non-zero entries. Grey blocks represent entries that are not necessarily zero, but are small in magnitude.

To be slightly more precise, we build at the i'th step unitary matrices $\mathbf{U}^{(i)}$ and $\mathbf{V}^{(i)}$ of sizes $m \times m$ and $n \times n$, respectively, such that

$$\mathbf{U} = \mathbf{U}^{(1)} \mathbf{U}^{(2)} \cdots \mathbf{U}^{(s)}, \qquad \text{and} \qquad \mathbf{V} = \mathbf{V}^{(1)} \mathbf{V}^{(2)} \cdots \mathbf{V}^{(s)}.$$

Using these matrices, we drive $\mathbf{A}$ towards upper triangular form through a sequence of transformations

$$\begin{aligned} \mathbf{A}^{(0)} &= \mathbf{A}, \\ \mathbf{A}^{(i)} &= \left(\mathbf{U}^{(i)}\right)^* \mathbf{A}^{(i-1)} \mathbf{V}^{(i)}, \qquad i = 1, 2, 3, \ldots, s, \\ \mathbf{T} &= \mathbf{A}^{(s)}. \end{aligned}$$

The objective at step i is to transform the i'th diagonal block to diagonal form, to zero out all blocks beneath the i'th diagonal block, and to make all blocks to the right of the i'th diagonal block as small in magnitude as possible.

Each matrix $\mathbf{U}^{(i)}$ and $\mathbf{V}^{(i)}$ consists predominantly of a product of b Householder reflectors. To be precise, each such matrix is a product of b Householder reflectors, but with the i'th block of b columns right-multiplied by a small $b \times b$ unitary matrix.

The next two sections provide additional details. Section 3.4 describes exactly how to build the transformation matrices $\mathbf{U}^{(1)}$ and $\mathbf{V}^{(1)}$ that are needed in the first step of the iteration. Then, Section 3.5 shows how to apply the techniques described in Section 3.4 repeatedly to build the full factorization.

3.4 A randomized algorithm for finding a pair of transformation matrices for the first step

3.4.1 Objectives for the construction

In this section, we describe a randomized algorithm for finding unitary matrices $\mathbf{U}$ and $\mathbf{V}$ that execute the first step of the process outlined in Section 3.3, and illustrated in Figure 3.2. To avoid notational clutter, we simplify the notation slightly, describing what we consider a basic single step of the process. Given an $m \times n$ matrix $\mathbf{A}$ and a block size b, we seek two orthonormal matrices $\mathbf{U}$ and $\mathbf{V}$ of sizes $m \times m$ and $n \times n$, respectively, such that the matrix

$$\mathbf{T} = \mathbf{U}^*\mathbf{A}\mathbf{V} \tag{3.11}$$

has a diagonal leading $b \times b$ block, and the entries beneath this block are all zeroed out. In Section 3.3, we referred to the matrices $\mathbf{U}$ and $\mathbf{V}$ as $\mathbf{U}^{(1)}$ and $\mathbf{V}^{(1)}$, respectively, and $\mathbf{T}$ as $\mathbf{A}^{(1)}$.

To make the discussion precise, let us partition $\mathbf{U}$ and $\mathbf{V}$ so that

$$\mathbf{U} = \big[\mathbf{U}_1 \mid \mathbf{U}_2\big], \qquad \text{and} \qquad \mathbf{V} = \big[\mathbf{V}_1 \mid \mathbf{V}_2\big], \tag{3.12}$$

where $\mathbf{U}_1$ and $\mathbf{V}_1$ each contain b columns. Then, set $\mathbf{T}_{ij} = \mathbf{U}_i^*\mathbf{A}\mathbf{V}_j$ for $i, j = 1, 2$ so that

$$\mathbf{U}^*\mathbf{A}\mathbf{V} = \left[\begin{array}{c|c} \mathbf{T}_{11} & \mathbf{T}_{12} \\ \hline \mathbf{T}_{21} & \mathbf{T}_{22} \end{array}\right]. \tag{3.13}$$

Our objective is now to build matrices $\mathbf{U}$ and $\mathbf{V}$ that accomplish the following:

1. $\mathbf{T}_{11}$ is diagonal, with entries that closely approximate the leading b singular values of $\mathbf{A}$.
2. $\mathbf{T}_{21} = \mathbf{0}$.
3. $\mathbf{T}_{12}$ has small magnitude.
4. The norm of $\mathbf{T}_{22}$ should be close to optimally small, so that $\|\mathbf{T}_{22}\| \approx \inf\{\|\mathbf{A}-\mathbf{C}\| : \mathbf{C} \text{ has rank } b\}$.

The purpose of condition (iv) is to minimize the error in the rank-b approximation to $\mathbf{A}$, cf. Section 3.5.3.

3.4.2 A theoretically ideal choice of transformation matrices

Suppose that we could somehow find two matrices $\mathbf{U}$ and $\mathbf{V}$ whose first b columns *exactly* span the subspaces spanned by the dominant left and right singular vectors, respectively. Finding such matrices is of course computationally hard, but *if* we could build them, then we would get the optimal result that

2. $\mathbf{T}_{21} = \mathbf{0}$.
3. $\mathbf{T}_{12} = \mathbf{0}$.
4. $\|\mathbf{T}_{22}\| = \sigma_{b+1} = \min\{\|\mathbf{A} - \mathbf{C}\| : \text{ matrix } \mathbf{C} \text{ has rank } b\}$.

Enforcing condition (i) is then very easy, since the dominant $b \times b$ block is now disconnected from the rest of the matrix. Simply executing a full SVD of this small $b \times b$ block, and then updating $\mathbf{U}$ and $\mathbf{V}$ will do the job.

3.4.3 A randomized technique for approximating the span of the dominant singular vectors

Inspired by the observation in Section 3.4.2 that a theoretically ideal right transformation matrix $\mathbf{V}$ is a matrix whose b first columns span the space spanned by the dominant b right singular vectors of $\mathbf{A}$, we use the randomized power iteration described in Section 3.2.5 to execute this task. We let q denote a small integer specifying how many steps of power iteration we take. Typically, $q = 0, 1, 2$ are good choices. Then,

take the following steps: (1) Draw a Gaussian random matrix $\mathbf{G}$ of size $m \times b$. (2) Compute a sampling matrix $\mathbf{Y} = \left(\mathbf{A}^*\mathbf{A}\right)^q \mathbf{A}^*\mathbf{G}$ of size $n \times b$. (3) Perform an unpivoted Householder QR factorization of $\mathbf{Y}$ so that

$$\begin{array}{ccccc} \mathbf{Y} & = & \mathbb{V} & & \mathbf{R}. \\ n \times b & & n \times n & & n \times b \end{array}$$

Observe that $\mathbb{V}$ will be a product of b Householder reflectors, and that the first b columns of $\mathbb{V}$ will form an orthonormal basis for the space spanned by the columns of $\mathbf{Y}$. In consequence, the first b columns of $\mathbb{V}$ form an orthonormal basis for a space that approximately spans the space spanned by the b dominant right singular vectors of $\mathbf{A}$. (The font used for $\mathbb{V}$ is a reminder that it is a product of Householder reflectors, which is exploited when it is stored and operated upon, cf. Section 3.2.4.)

3.4.4 Construction of the left transformation matrix

The process for finding the left transformation matrix $\mathbf{U}$ is deterministic, and will exactly transform the first b columns of $\mathbf{A}\mathbb{V}$ into a diagonal matrix. Observe that with $\mathbb{V}$ the unitary matrix constructed using the procedure in Section 3.4.3, we have the identity.

$$\mathbf{A} = \mathbf{A}\mathbb{V}\mathbb{V}^* = \big[\mathbf{A}\mathbb{V}_1 \mid \mathbf{A}\mathbb{V}_2\big]\mathbb{V}^*, \tag{3.14}$$

where the partitioning $\mathbb{V} = [\mathbb{V}_1\ \mathbb{V}_2]$ is such that $\mathbb{V}_1$ holds the first b columns of $\mathbb{V}$. We now perform a full SVD on the matrix $\mathbf{A}\mathbb{V}_1$, which is of size $m \times b$ so that

$$\begin{array}{ccccccc} \mathbf{A}\mathbb{V}_1 & = & \mathbf{U} & & \mathbf{D} & \mathbf{V}^*_{\text{small}}. \\ m \times b & & m \times m & & m \times b & b \times b \end{array} \tag{3.15}$$

Inserting (3.15) into (3.14) we obtain the identity

$$\mathbf{A} = \big[\mathbf{U}\mathbf{D}\mathbf{V}^*_{\text{small}} \mid \mathbf{A}\mathbb{V}_2\big]\mathbb{V}^*. \tag{3.16}$$

Factor out $\mathbf{U}$ in (3.16) to the left to get

$$\mathbf{A} = \mathbf{U}\big[\mathbf{D}\mathbf{V}^*_{\text{small}} \mid \mathbf{U}^*\mathbf{A}\mathbb{V}_2\big]\mathbb{V}^*.$$

Finally, factor out $\mathbf{V}_{\text{small}}$ to the right to yield the factorization

$$\mathbf{A} = \mathbf{U} \underbrace{\begin{bmatrix} \mathbf{D} & | & \mathbf{U}^*\mathbf{A}\mathbb{V}_2 \end{bmatrix}}_{=\mathbf{T}} \mathbf{V}^*, \qquad \text{with } \mathbf{V} = \mathbb{V} \begin{bmatrix} \mathbf{V}_{\text{small}} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{n-b} \end{bmatrix}. \tag{3.17}$$

Equation (3.17) is the factorization $\mathbf{A} = \mathbf{U}\mathbf{T}\mathbf{V}^*$ that we seek, with

$$\mathbf{T} = \begin{bmatrix} \mathbf{T}_{11} & \mathbf{T}_{12} \\ \mathbf{T}_{21} & \mathbf{T}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{D}(1:b, 1:b) & \mathbf{U}_1^*\mathbf{A}\mathbb{V}_2 \\ \mathbf{0} & \mathbf{U}_2^*\mathbf{A}\mathbb{V}_2 \end{bmatrix}.$$

Remark 10. *As we saw in (3.17), the right transformation matrix* $\mathbf{V}$ *consists of a product* $\mathbb{V}$ *of* b *Householder reflectors, with the first* b *columns rotated by a small unitary matrix* $\mathbf{V}_{\text{small}}$*. We will next demonstrate that the left transformation matrix* $\mathbf{U}$ *can be built in such a way that it can be written in an entirely analogous form. Simply observe that the matrix* $\mathbf{A}\mathbb{V}_1$ *in (3.15) is a tall thin matrix, so that the SVD in (3.15) can efficiently be computed by first performing an unpivoted QR factorization of* $\mathbf{A}\mathbb{V}_1$ *to yield a factorization*

$$\underset{m \times b}{\mathbf{A}\mathbb{V}_1} \;=\; \underset{m \times m}{\mathbb{U}} \quad \underset{m \times b}{\begin{bmatrix} \mathbf{R}_{11} \\ \mathbf{0} \end{bmatrix}},$$

where $\mathbf{R}_{11}$ *is of size* $b \times b$*, and* $\mathbb{U}$ *is a product of* b *Householder reflectors, cf. Section 3.2.4. Then, perform an SVD of* $\mathbf{R}_{11}$ *to obtain*

$$\underset{b \times b}{\mathbf{R}_{11}} \;=\; \underset{b \times b}{\mathbf{U}_{\text{small}}} \; \underset{b \times b}{\mathbf{D}_{\text{small}}} \; \underset{b \times b}{\mathbf{V}_{\text{small}}^*}.$$

The factorization (3.15) then becomes

$$\underset{m \times b}{\mathbf{A}\mathbb{V}_1} \;=\; \underset{m \times m}{\mathbf{U}} \quad \underset{m \times b}{\begin{bmatrix} \mathbf{D}_{\text{small}} \\ \mathbf{0} \end{bmatrix}} \; \underset{b \times b}{\mathbf{V}_{\text{small}}^*}, \qquad \textit{with } \mathbf{U} = \mathbb{U} \begin{bmatrix} \mathbf{U}_{\text{small}} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{bmatrix}. \tag{3.18}$$

We see that the expression for $\mathbf{U}$ *in (3.18) is exactly analogous to the expression for* $\mathbf{V}$ *in (3.17).*

3.4.5 Summary of the construction of transformation matrices

Even though the derivation of the transformation matrices got slightly long, the final algorithm is simple. It can be written down precisely with just a few lines of Matlab code, as shown in the right panel in

Figure 3.3 (the single step process described in this section is the subroutine `stepUTV`).

```
function [U,T,V] = randUTV(A,b,q)
  T = A;
  U = eye(size(A,1));
  V = eye(size(A,2));
  for i = 1:ceil(size(A,2)/b)
    I1 = 1:(b*(i-1));
    I2 = (b*(i-1)+1):size(A,1);
    J2 = (b*(i-1)+1):size(A,2);
    if (length(J2) > b)
      [UU,TT,VV] = stepUTV(T(I2,J2),b,q);
    else
      [UU,TT,VV] = svd(T(I2,J2));
    end
    U(:,I2)  = U(:,I2)*UU;
    V(:,J2)  = V(:,J2)*VV;
    T(I2,J2) = TT;
    T(I1,J2) = T(I1,J2)*VV;
  end
return
```

```
function [U,T,V] = stepUTV(A,b,q)
  G = randn(size(A,1),b);
  Y = A'*G;
  for i = 1:q
    Y = A'*(A*Y);
  end
  [V,~] = qr(Y);
  [U,D,W]  = svd(A*V(:,1:b));
  T = [D,U'*A*V(:,(b+1):end)];
  V(:,1:b) = V(:,1:b)*W;
return
```

Figure 3.3: Matlab code for the algorithm randUTV that given an $m \times n$ matrix $\mathbf{A}$ computes its UTV factorization $\mathbf{A} = \mathbf{UTV}^*$, cf. (3.1). The input parameters b and q reflect the block size and the number of steps of power iteration, respectively. This code is simplistic in that products of Householder reflectors are stored simply as dense matrices, making the overall complexity $O(n^4)$; it also assumes that $m \geq n$. An efficient implementation is described in Figure 3.4.

3.5 The algorithm randUTV

In this section, we describe the algorithm randUTV that given an $m \times n$ matrix $\mathbf{A}$ computes a UTV factorization of the form (3.1). For concreteness, we assume that $m \geq n$ and that we seek to build an *upper* triangular middle matrix $\mathbf{T}$. The modifications required for the other cases are straight-forward. Section 3.5.1 describes the most basic version of the scheme, Section 3.5.2 describes a computationally efficient version, and Section 3.5.4 provides a calculation of the asymptotic flop count of the resulting algorithm.

3.5.1 A simplistic algorithm

The algorithm randUTV is obtained by applying the single-step algorithm described in Section 3.4 repeatedly, to drive $\mathbf{A}$ to upper triangular form one block of b columns at a time. We recall that a cartoon

of the process is shown in Figure 3.2. At the start of the process, we create three arrays that hold the output matrices $\mathbf{T}$, $\mathbf{U}$, and $\mathbf{V}$, and initialize them by setting

$$\mathbf{T} = \mathbf{A}, \qquad \mathbf{U} = \mathbf{I}_m, \qquad \mathbf{V} = \mathbf{I}_n.$$

In the first step of the iteration, we use the single-step technique described in Section 3.4 to create two unitary "left and right transformation matrices" $\mathbf{U}^{(1)}$ and $\mathbf{V}^{(1)}$ and then update $\mathbf{T}$, $\mathbf{U}$, and $\mathbf{V}$ accordingly:

$$\mathbf{T} \leftarrow (\mathbf{U}^{(1)})^*\mathbf{T}\mathbf{V}^{(1)}, \qquad \mathbf{U} \leftarrow \mathbf{U}\mathbf{U}^{(1)}, \qquad \mathbf{V} \leftarrow \mathbf{V}\mathbf{V}^{(1)}.$$

This leaves us with a matrix $\mathbf{T}$ whose b leading columns are upper triangular (like matrix $\mathbf{A}^{(1)}$ in Figure 3.2). For the second step, we build transformation matrices $\mathbf{U}^{(2)}$ and $\mathbf{V}^{(2)}$ by applying the single-step algorithm described in Section 3.4 to the remainder matrix $\mathbf{T}((b+1):m, (b+1):n)$, and then updating $\mathbf{T}$, $\mathbf{U}$, and $\mathbf{V}$ accordingly.

The process then continues to drive one block of b columns at a time to upper triangular form. With $s = \lceil n/b \rceil$ denoting the total number of steps, we find that after $s-1$ steps, all that remains to process is the bottom right block of $\mathbf{T}$ (cf. the matrix $\mathbf{A}^{(2)}$ in Figure 3.2). This block consists of b columns if n is a multiple of the block size, and is otherwise even thinner. For this last block, we obtain the final left and right transformation matrices $\mathbf{U}^{(s)}$ and $\mathbf{V}^{(s)}$ by computing a full singular value decomposition of the remaining matrix $\mathbf{T}(((s-1)b+1):m, ((s-1)b+1):n)$, and updating the matrices $\mathbf{T}$, $\mathbf{U}$, and $\mathbf{V}$ accordingly. (In the cartoon in Figure 3.2, we have $s = 3$, and the matrices $\mathbf{U}^{(3)}$ and $\mathbf{V}^{(3)}$ are built by computing a full SVD of a dense matrix of size $2s \times s$.)

We call the algorithm described in this section randUTV. It can be coded using just a few lines of Matlab code, as illustrated in Figure 3.3. In this simplistic version, all unitary matrices are represented as dense matrices, which makes the overall complexity $O(n^4)$ for an $n \times n$ matrix.

3.5.2 A computationally efficient version

In this section, we describe how the basic version of randUTV, as given in Figure 3.3, can be turned into a highly efficient procedure via three simple modifications. The resulting algorithm is summarized in

Figure 3.4. We note that the two versions of randUTV described in Figures 3.3 and 3.4 are mathematically equivalent; if they were to be executed in exact arithmetic, their outputs would be identical.

The first modification is that all operations on the matrix $\mathbf{T}$ and on the unitary matrices $\mathbf{U}$ and $\mathbf{V}$ should be carried out "in place" to not unnecessarily move any data. To be precise, using the notation in Section 3.4, we generate at each iteration four unitary matrices $\mathbb{U}$, $\mathbb{V}$, $\mathbf{U}_{\text{small}}$, and $\mathbf{V}_{\text{small}}$. As soon as such a matrix is generated, it is immediately applied to $\mathbf{T}$, and then used to update either $\mathbf{U}$ or $\mathbf{V}$.

Second, we exploit that the two "large" unitary transforms $\mathbb{U}$ and $\mathbb{V}$ both consist of products of b Householder reflectors. We generate them by computing unpivoted Householder QR factorizations of tall and thin matrices, using a subroutine that outputs simply the b Householder vectors. Then, $\mathbb{U}$ and $\mathbb{V}$ can both be stored and applied efficiently, as described in Section 3.2.4.

The third and final modification pertains to the situation where the input matrix $\mathbf{A}$ is non-square. In this case, the full SVD that is computed in the last step involves a rectangular matrix. When $\mathbf{A}$ is tall ($m > n$), we find at this step that J_3 is empty, so the matrix to be processed is $\mathbf{T}([I_2, I_3], J_2)$. When computing the SVD of this matrix, we use the efficient version described in Remark 10, which outputs a factorization in which $\mathbf{U}_{\text{small}}$ consists in part of a product of Householder reflectors. (In this situation $\mathbf{U}_{\text{small}}$ is in fact not necessarily "small," but it can be stored and applied efficiently.)

Remark 11 (The case $m < n$)**.** *In a situation where the matrix has fewer rows than columns, but we still seek an upper triangular matrix* $\mathbf{T}$*, randUTV proceeds exactly as described for the first* $s - 1$ *steps. In the final step, we now find that* I_3 *is empty, but* J_3 *is not, and so we need to compute the SVD of the "fat" matrix* $\mathbf{T}(I_2, [J_2, J_3])$*. We do this in a manner entirely analogous to how we handle a "tall" matrix, by first performing an unpivoted QR factorization of the rows of* $\mathbf{T}(I_2, [J_2, J_3])$*. In this situation it is the matrix of right singular vectors at the last step that consists in part of a product of Householder reflectors.*

3.5.3 Connection between RSVD and randUTV

The proposed algorithm randUTV is directly inspired by the Randomized SVD (RSVD) algorithm described in Remark 9 (as originally described in [90, 83, 111] and later elaborated in [91, 69]). In this sec-

$[\mathbf{U},\ \mathbf{T},\ \mathbf{V}] = randUTV(\ \mathbf{A},\ b,\ q)$
% Initialize output variables:
$\mathbf{T} = \mathbf{A}$; $\mathbf{U} = \mathbf{I}_m$; $\mathbf{V} = \mathbf{I}_n$;

for $i = 1 : \min(\lceil m/b \rceil, \lceil n/b \rceil)$ **do**
% Create partitions $1 : m = [I_1, I_2, I_3]$ *and* $1 : n = [J_1, J_2, J_3]$ *so that* (I_2, J_2) *points to the "active" block that is to be diagonalized:*
$I_1 = 1 : (b(i-1))$; $I_2 = (b(i-1)+1) : \min(bi, m)$; $I_3 = (bi+1) : m$;
$J_1 = 1 : (b(i-1))$; $J_2 = (b(i-1)+1) : \min(bi, n)$; $J_3 = (bi+1) : n$;

if (I_3 and J_3 are both nonempty) **then**
% Generate the sampling matrix $\mathbf{Y}$ *whose columns approximately span the space spanned by the* b *dominant right singular vectors of the matrix* $\mathbf{X} = \mathbf{T}([I_2,\ I_3], [J_2,\ J_3])$. *We do this via randomized sampling, setting* $\mathbf{Y} = \big(\mathbf{X}^*\mathbf{X}\big)^q\mathbf{X}^*\mathbf{G}$ *where* $\mathbf{G}$ *is a Gaussian random matrix with* b *columns.*
$\mathbf{G} = \mathtt{randn}(m - b(i-1), b)$
$\mathbf{Y} = \mathbf{T}([I_2, I_3], [J_2, J_3])^*\mathbf{G}$
for $j = 1 : q$ **do**
$\mathbf{Y} = \mathbf{T}([I_2, I_3], [J_2, J_3])^*\big(\mathbf{T}([I_2, I_3], [J_2, J_3])\mathbf{Y}\big)$.
end for

% Build a unitary matrix $\mathbb{V}$ *whose first* b *columns form an orthonormal basis for the columns of* $\mathbf{Y}$. *Then, apply the transformations. (We exploit that* $\mathbb{V}$ *is a product of* b *Householder reflectors, cf. Section 3.2.4.)*
$[\mathbb{V}, \sim] = \mathtt{qr}(\mathbf{Y})$
$\mathbf{T}(:, [J_2,\ J_3]) = \mathbf{T}(:, [J_2, J_3])\mathbb{V}$
$\mathbf{V}(:, [J_2,\ J_3]) = \mathbf{V}(:, [J_2, J_3])\mathbb{V}$

% Build a unitary matrix $\mathbb{U}$ *whose first* b *columns form an orthonormal basis for the columns of* $\mathbf{T}([I_2, I_3], J_2)$. *Then,*
apply the transformations. (We exploit that $\mathbb{U}$ *is a product of* b *Householder reflectors, cf. Section 3.2.4.)*
$[\mathbb{U}, \mathbf{R}] = \mathtt{qr}(\mathbf{T}([I_2, I_3], J_2))$
$\mathbf{U}(:, [I_2, I_3]) = \mathbf{U}(:, [I_2, I_3])\mathbb{U}$
$\mathbf{T}([I_2, I_3], J_3) = \mathbb{U}^*\mathbf{T}([I_2, I_3], J_3)$
$\mathbf{T}(I_3, J_2) = \mathbf{0}$

% Perform the local SVD that diagonalizes the active diagonal block. Then, apply the transformations.
$[\mathbf{U}_{\text{small}}, \mathbf{D}_{\text{small}}, \mathbf{V}_{\text{small}}] = \mathtt{svd}(\mathbf{R}(1 : b, 1 : b))$
$\mathbf{T}(I_2, J_2) = \mathbf{D}_{\text{small}}$
$\mathbf{T}(I_2, J_3) = \mathbf{U}_{\text{small}}^*\mathbf{T}(I_2, J_3)$
$\mathbf{U}(:, I_2) = \mathbf{U}(:, I_2)\mathbf{U}_{\text{small}}$
$\mathbf{T}(I_1, J_2) = \mathbf{T}(I_1, J_2)\mathbf{V}_{\text{small}}$
$\mathbf{V}(:, J_2) = \mathbf{V}(:, J_2)\mathbf{V}_{\text{small}}$
else

% Perform the local SVD that diagonalizes the last diagonal block. Then, apply the transformations. If either I_3 *or* J_3 *is long, this should be done economically, cf. Section 3.5.2.*
$[\mathbf{U}_{\text{small}}, \mathbf{D}_{\text{small}}, \mathbf{V}_{\text{small}}] = \mathtt{svd}(\mathbf{T}([I_2, I_3], [J_2, J_3]))$
$\mathbf{U}(:, [I_2, I_3]) = \mathbf{U}(:, [I_2, I_3])\mathbf{U}_{\text{small}}$
$\mathbf{V}(:, [J_2, J_3]) = \mathbf{V}(:, [J_2, J_3])\mathbf{V}_{\text{small}}$
$\mathbf{T}([I_2, I_3], [J_2, J_3]) = \mathbf{D}_{\text{small}}$
$\mathbf{T}(I_1, [J_2, J_3]) = \mathbf{T}(I_1, [J_2, J_3])\mathbf{V}_{\text{small}}$
end if
end for
return

Figure 3.4: The algorithm randUTV that given an $m \times n$ matrix $\mathbf{A}$ computes the UTV factorization $\mathbf{A} = \mathbf{UTV}^*$, cf. (3.1). The input parameters b and q reflect the block size and the number of steps of power iteration, respectively.

tion, we explore this connection in more detail, and demonstrate that the low-rank approximation error that results from the "single-step" UTV-factorization described in Section 3.4 is identical to the error produced by the RSVD (with a twist). This means that the detailed error analysis that is available for the RSVD (see, e.g., [129, 62, 69]) immediately applies to the procedure described here. To be precise:

Theorem 1. *Let* $\mathbf{A}$ *be an* $m \times n$ *matrix, let* b *be an integer denoting step size such that* $1 \leq b < \min(m, n)$*, and let* q *denote a non-negative integer. Let* $\mathbf{G}$ *denote the* $m \times b$ *Gaussian matrix drawn in Section 3.4.3, and let* $\mathbf{U}$*,* $\mathbf{T}$*, and* $\mathbf{V}$ *be the factors in the factorization* $\mathbf{A} = \mathbf{U}\mathbf{T}\mathbf{V}^*$ *built in Sections 3.4.3 and 3.4.4, partitioned as in (3.12) and (3.13).*

1. *Let* $\mathbf{Y} = \big(\mathbf{A}^*\mathbf{A}\big)^q\mathbf{A}^*\mathbf{G}$ *denote a sampling matrix, and let* $\mathbf{Q}$ *denote an* $n \times b$ *orthonormal matrix whose columns form a basis for the column space of* $\mathbf{Y}$*. Then, the error* $\|\mathbf{A} - \mathbf{A}\mathbf{Q}\mathbf{Q}^*\|$ *precisely equals the error incurred by the RSVD with* q *steps of power iteration, as analyzed in [69, Sec. 10]. It holds that*

$$\|\mathbf{A} - \mathbf{A}\mathbf{Q}\mathbf{Q}^*\| = \|\mathbf{A} - \mathbf{U}_1\mathbf{T}_{11}\mathbf{V}_1^*\| = \left\| \begin{bmatrix} \mathbf{T}_{12} \\ \\ \mathbf{T}_{22} \end{bmatrix} \right\|. \tag{3.19}$$

2. *Let* $\mathbf{Z} = \mathbf{A}\mathbf{Y} = \big(\mathbf{A}\mathbf{A}^*\big)^{q+1}\mathbf{G}$ *denote a sampling matrix, and let* $\mathbf{W}$ *denote an* $m \times b$ *orthonormal matrix whose columns form a basis for the column space of* $\mathbf{Z}$*. If the rank of* $\mathbf{A}$ *is at least* b*, then*

$$\|\mathbf{A} - \mathbf{W}\mathbf{W}^*\mathbf{A}\| = \|\mathbf{A} - \mathbf{U}_1\big(\mathbf{T}_{11}\mathbf{V}_1^* + \mathbf{T}_{12}\mathbf{V}_2^*\big)\| = \|\mathbf{T}_{22}\|\,. \tag{3.20}$$

We observe that the term $\|\mathbf{A} - \mathbf{W}\mathbf{W}^*\mathbf{A}\|$ that arises in part (b) can informally be said to be the error resulting from RSVD with "$q + 1/2$" steps of power iteration. This conforms with what one might have optimistically hoped for, given that the RSVD involves $2q+2$ applications of either $\mathbf{A}$ or $\mathbf{A}^*$ to thin matrices with b columns, and randUTV involves $2q + 3$ such operations at each step ($2q + 1$ applications in building $\mathbf{Y}$, and then the computations of $\mathbf{A}\mathbb{V}$ and $\mathbb{U}^*\mathbf{A}$, which are in practice applications of $\mathbf{A}$ to thin matrices due to the identity (3.5)).

Proof. The proofs for the two parts rest on the fact that $\mathbf{A}$ can be decomposed as follows:

$$\mathbf{A} = \mathbf{U}\mathbf{T}\mathbf{V}^* = \begin{bmatrix}\mathbf{U}_1 & \mathbf{U}_2\end{bmatrix} \begin{bmatrix}\mathbf{T}_{11} & \mathbf{T}_{12} \\ \mathbf{0} & \mathbf{T}_{22}\end{bmatrix} \begin{bmatrix}\mathbf{V}_1^* \\ \mathbf{V}_2^*\end{bmatrix} = \mathbf{U}_1\mathbf{T}_{11}\mathbf{V}_1^* + \mathbf{U}_1\mathbf{T}_{12}\mathbf{V}_2^* + \mathbf{U}_2\mathbf{T}_{22}\mathbf{V}_2^*, \tag{3.21}$$

where $\mathbf{U}_1$ and $\mathbf{V}_1$ have b columns each, and $\mathbf{T}_{11}$ is of size $b \times b$. With the identity (3.21) in hand, the claims in (a) follow once we have established that the orthogonal projection $\mathbf{Q}\mathbf{Q}^*$ equals the projection $\mathbf{V}_1\mathbf{V}_1^*$. The claims in (b) follow once we establish that $\mathbf{W}\mathbf{W}^* = \mathbf{U}_1\mathbf{U}_1^*$.

(a) Observe that the matrix $\mathbf{Q}$ is by construction identical to the matrix $\mathbb{V}_1$ built in Sections 3.4.3 and 3.4.4. Since $\mathbb{V}_1 = \mathbf{V}_1\mathbf{V}_{\text{small}}^*$, we find that $\mathbf{Q}\mathbf{Q}^* = \mathbb{V}_1\mathbb{V}_1^* = \big(\mathbf{V}_1\mathbf{V}_{\text{small}}^*\big)\big(\mathbf{V}_1\mathbf{V}_{\text{small}}^*\big)^* = \mathbf{V}_1\big(\mathbf{V}_{\text{small}}^*\mathbf{V}_{\text{small}}\big)\mathbf{V}_1^*$. Since $\mathbf{V}_{\text{small}}^*\mathbf{V}_{\text{small}} = \mathbf{I}_b$, it follows that $\mathbf{Q}\mathbf{Q}^* = \mathbf{V}_1\mathbf{V}_1^*$. Then

$$\mathbf{A}\mathbf{Q}\mathbf{Q}^* = \mathbf{A}\mathbf{V}_1\mathbf{V}_1^* = \{\text{Use } (3.21) \text{ and that } \mathbf{V}_2^*\mathbf{V}_1 = \mathbf{0} \text{ and } \mathbf{V}_1^*\mathbf{V}_1 = \mathbf{I}.\} = \mathbf{U}_1\mathbf{T}_{11}\mathbf{V}_1^*. \tag{3.22}$$

The first identity in (3.19) follows immediately from (3.22). The second identity holds since (3.21) implies that $\mathbf{A} - \mathbf{U}_1\mathbf{T}_{11}\mathbf{V}_1^* = \mathbf{U}_1\mathbf{T}_{12}\mathbf{V}_2^* + \mathbf{U}_2\mathbf{T}_{22}\mathbf{V}_2^* = \mathbf{U}\begin{bmatrix}\mathbf{T}_{12} \\ \mathbf{T}_{22}\end{bmatrix}\mathbf{V}_2^*$, with $\mathbf{U}$ unitary and $\mathbf{V}_2$ orthonormal.

(b) We will first prove that with probability 1, the two $m \times b$ matrices $\mathbf{A}\mathbb{V}_1$ and $\mathbf{Z}$ have the same column spaces. To this end, note that since $\mathbb{V}_1$ is obtained by performing an unpivoted QR factorization of the matrix $\mathbf{Y}$ defined in (a), we know that $\mathbb{V}_1\mathbf{R} = \mathbf{Y}$ for some $b \times b$ upper triangular matrix $\mathbf{R}$. The assumption that $\mathbf{A}$ has rank at least b implies that $\mathbf{R}$ is invertible with probability 1. Consequently, $\mathbf{A}\mathbb{V}_1 = \mathbf{A}\mathbf{Y}\mathbf{R}^{-1} = \mathbf{Z}\mathbf{R}^{-1}$, since $\mathbf{Z} = \mathbf{A}\mathbf{Y}$. Since right multiplication by an invertible matrix does not change the column space of a matrix, the claim follows.

Since $\mathbf{A}\mathbb{V}_1$ and $\mathbf{Z}$ have the same column spaces, it follows from the definition of $\mathbb{U}$ that $\mathbb{U}_1\mathbb{U}_1^* = \mathbf{W}\mathbf{W}^*$. Since $\mathbf{U}_1 = \mathbb{U}_1\mathbf{U}_{\text{small}}$ where $\mathbf{U}_{\text{small}}$ is unitary, we see that $\mathbb{U}_1\mathbb{U}_1^* = \mathbf{U}_1\mathbf{U}_1^*$. Consequently,

$$\mathbf{W}\mathbf{W}^*\mathbf{A} = \mathbf{U}_1\mathbf{U}_1^*\mathbf{A} = \{\text{Use } (3.21).\} = \mathbf{U}_1\mathbf{T}_{11}\mathbf{V}_1^* + \mathbf{U}_1\mathbf{T}_{12}\mathbf{V}_2^*\,,$$

which establishes the first identity in (3.20). The second identity holds since (3.21) implies that $\mathbf{A} - \mathbf{U}_1(\mathbf{T}_{11}\mathbf{V}_1^* + \mathbf{T}_{12}\mathbf{V}_2^*) = \mathbf{U}_2\mathbf{T}_{22}\mathbf{V}_2^*$, with $\mathbf{U}_2$ and $\mathbf{V}_2$ orthonormal. □

Remark 12 (Oversampling)**.** *We recall that the accuracy of randUTV depends on how well the space col*$(\mathbb{V}_1)$ *aligns with the space spanned by the* b *dominant right singular vectors of* $\mathbf{A}$*. If these two spaces were to match exactly, then the truncated UTV factorization would achieve perfectly optimal accuracy. One way to improve the alignment is to increase the power parameter* q*. A second way to make the two spaces align better is to use oversampling, as described in Remark 8. With* p *an over-sampling parameter (say* $p = 5$ *or* $p = 10$*), we would draw a Gaussian random matrix* $\mathbf{G}$ *of size* $m \times (b + p)$*, and then compute an "extended" sampling matrix* $\mathbf{Y}' = \left(\mathbf{A}^*\mathbf{A}\right)^q \mathbf{A}^*\mathbf{G}$ *of size* $n \times (b+p)$*. The* $n \times b$ *sampling matrix* $\mathbf{Y}$ *we would use to compute* $\mathbb{V}$ *would then be formed by the* b *dominant left singular vectors of* $\mathbf{Y}'$*. Oversampling in this fashion does improve the accuracy, but in our experience, the additional computational cost is not worth it. Incorporating over-sampling would also introduce an additional tuning parameter* p*, which is in many ways undesirable.*

3.5.4 Theoretical cost of randUTV

Now we analyze the theoretical cost of the implementation of randUTV, and we compare it to those of CPQR and SVD.

The theoretical cost of the CPQR factorization and the unpivoted QR factorization of an $m \times n$ matrix is: $2mn^2 - 2n^3/3$ flops, when no orthonormal matrices are required. Although the theoretical cost of both the pivoted and the unpivoted QR is the same, other factors should be considered, being the most important one the quality of flops. In modern architectures, flops performed inside BLAS-3 operations can be about 5–10 times faster than flops performed inside BLAS-1 and BLAS-2 operations, since BLAS-3 is CPU-bound whereas BLAS-1 and BLAS-2 are memory-bound. Hence, the unpivoted QR factorization in high-performance libraries such as [4] can be much faster because most of the flops are performed inside BLAS-3 operations, whereas only half of the flops in the best implementations of CPQR (`dgeqp3`) [107] in [4] are performed inside BLAS-3 operations. In addition, this low performance of CPQR can even be smaller because of the appearance of catastrophic cancellations during the computations. The appearance of just one catastrophic cancellation will stop the building of a block Householder reflector before all of it has been built. This sudden stop forces the algorithm to work on smaller block sizes, which are suboptimal,

and hence performances are even lower.

The SVD usually comprises two steps: the reduction to bidiagonal form, and then the reduction from bidiagonal to diagonal form. The first step is a direct step, whereas the second step is an iterative one. The theoretical cost of the reduction to bidiagonal form of an $m \times n$ matrix is: $4mn^2 - 4n^3/3$ flops, when no singular vectors are needed. If $m \gg n$, the cost can be reduced to: $2mn^2 + 2n^3$ flops by performing first a QR factorization. The cost of the reduction to diagonal form depends on the number of iterations, which is unknown a priori, but it is usually small when no singular vectors are built. On the one hand, in the bidiagonalization a large share of the flops are performed inside the not-so-efficient BLAS-1 and BLAS-2 operations. Therefore, no high performances are obtained in the reduction to bidiagonal form. On the other hand, the reduction to diagonal form uses just BLAS-1 operations. These operations are memory-bound, and in addition they cannot be efficiently parallelized within BLAS, which might reduce performances on multicore machines. In conclusion, usual implementations of the SVD will render low performances on both single-core architectures and multicore architectures.

The theoretical cost of the randUTV factorization of an $m \times n$ matrix is: $(5+2q)mn^2-(3+2q)n^3/3$ flops, when no orthonormal matrices are required, and q steps of power iteration are applied. If $q = 0$, the theoretical cost of randUTV is three times as high as the theoretical cost of CPQR; if $q = 1$, it is four times as high; and if $q = 2$, it is five times as high. Although randUTV seems computationally more expensive than CPQR, the quality of flops should be considered. The share of BLAS-1 and BLAS-2 flops in randUTV is very small: BLAS-1 and BLAS-2 flops are only employed inside the CPQR factorization of the sampling matrix $\mathbf{Y}$, the QR factorization of the current column block, and the SVD of the diagonal block. As these operations only apply to blocks of dimensions $n \times b$, $m \times b$, and $b \times b$, respectively, the total amount of these types of flops is negligible, and therefore most of the flops performed in randUTV are BLAS-3 flops. Hence, the algorithm for computing the randUTV will be much faster than what the theoretical cost predicts. In conclusion, this heavy use of BLAS-3 operations will render good performances on single-core architectures, multicore architectures, GPUs, and distributed-memory architectures.

3.6 Numerical results

3.6.1 Computational speed

In this section, we investigate the speed of the proposed algorithm randUTV, and compare it to the speeds of highly optimized methods for computing the SVD and the column pivoted QR (CPQR) factorization.

All experiments reported in this article were performed on an Intel Xeon E5-2670 processor at 2.6 GHz, with 16 cores and 64 GiB of RAM. Other details of interest include that the OS used was Linux (Version 2.6.32-431.el6.x86_64), and the code was compiled with gcc (Version 4.4.7). Main routines for computing the SVD (`dgesvd` and `dgesdd`) and the CPQR (`dgeqp3`) were taken from Intel's MKL library (Version 11.1.3) since this library usually delivers much higher performances than Netlib's LAPACK codes. Our implementations were coded with `libflame` [125, 124] (Release 11104).

Each of the three algorithms we tested (randUTV, SVD, CPQR) was applied to double-precision real matrices of size $n \times n$. We report the following times:

T_{svd}	The time in seconds for the LAPACK functions `dgesvd` and `dgesdd` from Intel's MKL.
T_{cpqr}	The time in seconds for the LAPACK function `dgeqp3` from Intel's MKL.
T_{randUTV}	The time in seconds for our implementation of randUTV.

For the purpose of a fair comparison, the three implementations were linked to the BLAS library from Intel's MKL. In all cases, we used an algorithmic block size of $b = 64$. While likely not optimal for all problem sizes, this block size yields near best performance and, regardless, it allows us to easily compare and contrast the performance of the different implementations.

Table 3.1 shows the measured computational times when executed on 1, 8, and 16 cores, respectively. In these experiments, all orthonormal matrices ($\mathbf{U}$ and $\mathbf{V}$ for SVD and UTV, and $\mathbf{Q}$ for CPQR) are explicitly formed. This slightly favors CPQR since only one orthonormal matrix is required. The corresponding numbers obtained when orthonormal matrices are not built are given in an appendix of [89]. To better illustrate the relative performance of the various techniques, we plot in Figure 3.5 the computational times

measured divided by n^3. Since all techniques under consideration have asymptotic complexity $O(n^3)$ when applied to an $n \times n$ matrix, these graphs better reveal the computational efficiency. (We plot time divided by n^3 rather than the more commonly reported "normalized Gigaflops" since the algorithms we compare have different scaling factors multiplying the dominant n^3-term in the asymptotic flop count.) Figure 3.5 also shows the timings we measured when the orthonormal matrices were not formed.

The results in Figure 3.5 lead us to make several observations: (1) The algorithm randUTV is decisively faster than the SVD in almost all cases (the exceptions involve the situation where no unitary matrices are sought, and the input matrix is small). (2) Comparing the speeds of CPQR and randUTV, when both methods are executed on a single core, CPQR is similar when no orthonormal matrices are built or slightly faster when orthonormal matrices are built (because in this case randUTV must build two matrices). (3) As the matrix size grows, and as the number of cores increases, randUTV gains an edge on CPQR in terms of speed.

3.6.2 Errors

In this section, we describe the results of the numerical experiments that were conducted to investigate how good randUTV is at computing an accurate rank-k approximation to a given matrix. Specifically, we compared how well partial factorizations reveal the numerical ranks of four different test matrices:

- *Matrix 1 (Fast Decay):* This is an $n \times n$ matrix of the form $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^*$ where $\mathbf{U}$ and $\mathbf{V}$ are randomly drawn matrices with orthonormal columns (obtained by performing an unpivoted QR factorization on a random Gaussian matrix), and where $\mathbf{D}$ is diagonal with entries given by $d_j = \beta^{(j-1)/(n-1)}$ with $\beta = 10^{-5}$.

- *Matrix 2 (S-shaped Decay):* This matrix is built in the same manner as "Matrix 1", but now the diagonal entries of $\mathbf{D}$ are chosen to first hover around 1, then decay rapidly, and then level out at 10^{-2}, as shown in Figure 3.7 (black line).

- *Matrix 3 (Gap):* This matrix is built in the same manner as "Matrix 1", but now there is a sudden drop in magnitudes of the singular values so that $\sigma_{151} = 0.1\,\sigma_{150}$. Specifically, $\mathbf{D}(j, j) = 1/j$ for

Computational times when executed on a single core

n	T_{svd}		T_{cpqr}	T_{randUTV}		
	`dgesvd`	`dgesdd`		$q=0$	$q=1$	$q=2$
500	1.93e-01	1.24e-01	3.78e-02	8.43e-02	9.37e-02	1.03e-01
1000	1.22e+00	7.82e-01	2.62e-01	5.08e-01	5.74e-01	6.45e-01
2000	9.04e+00	5.88e+00	2.18e+00	3.58e+00	4.09e+00	4.61e+00
3000	2.98e+01	2.00e+01	7.59e+00	1.14e+01	1.31e+01	1.50e+01
4000	7.02e+01	4.70e+01	1.80e+01	2.71e+01	3.13e+01	3.56e+01
5000	1.35e+02	9.04e+01	3.48e+01	5.07e+01	5.90e+01	6.74e+01
6000	2.29e+02	1.54e+02	5.95e+01	8.74e+01	1.01e+02	1.16e+02
8000	5.47e+02	3.68e+02	1.43e+02	2.10e+02	2.45e+02	2.81e+02
10000	1.03e+03	6.98e+02	2.74e+02	3.95e+02	4.61e+02	5.26e+02

Computational times when executed on 8 cores

n	T_{svd}		T_{cpqr}	T_{randUTV}		
	`dgesvd`	`dgesdd`		$q=0$	$q=1$	$q=2$
500	1.12e-01	7.70e-02	2.32e-02	5.36e-02	5.64e-02	5.92e-02
1000	5.27e-01	3.23e-01	1.02e-01	2.18e-01	2.35e-01	2.51e-01
2000	2.63e+00	1.57e+00	4.92e-01	1.14e+00	1.24e+00	1.33e+00
3000	8.23e+00	5.04e+00	2.36e+00	2.92e+00	3.25e+00	3.56e+00
4000	1.94e+01	1.34e+01	6.45e+00	5.92e+00	6.57e+00	7.28e+00
5000	3.79e+01	2.71e+01	1.28e+01	1.01e+01	1.13e+01	1.25e+01
6000	6.46e+01	4.82e+01	2.24e+01	1.62e+01	1.84e+01	2.05e+01
8000	1.55e+02	1.21e+02	5.37e+01	3.70e+01	4.20e+01	4.70e+01
10000	2.90e+02	2.29e+02	1.01e+02	6.48e+01	7.43e+01	8.39e+01

Computational times when executed on 16 cores

n	T_{svd}		T_{cpqr}	T_{randUTV}		
	`dgesvd`	`dgesdd`		$q=0$	$q=1$	$q=2$
500	1.08e-01	7.74e-02	2.32e-02	5.83e-02	6.14e-02	6.44e-02
1000	4.69e-01	3.03e-01	9.06e-02	2.24e-01	2.38e-01	2.52e-01
2000	2.26e+00	1.33e+00	4.09e-01	9.51e-01	1.02e+00	1.09e+00
3000	7.05e+00	4.34e+00	2.16e+00	2.34e+00	2.57e+00	2.81e+00
4000	1.73e+01	1.21e+01	6.08e+00	4.62e+00	5.10e+00	5.57e+00
5000	3.34e+01	2.48e+01	1.22e+01	7.54e+00	8.31e+00	9.09e+00
6000	5.86e+01	4.47e+01	2.15e+01	1.24e+01	1.38e+01	1.52e+01
8000	1.38e+02	1.12e+02	5.15e+01	2.72e+01	3.02e+01	3.32e+01
10000	2.62e+02	2.14e+02	9.70e+01	4.57e+01	5.12e+01	5.68e+01

Table 3.1: Computational times of different factorizations when executed on one core (top), 8 cores (middle), and 16 cores (bottom). All orthonormal matrices are built explicitly.

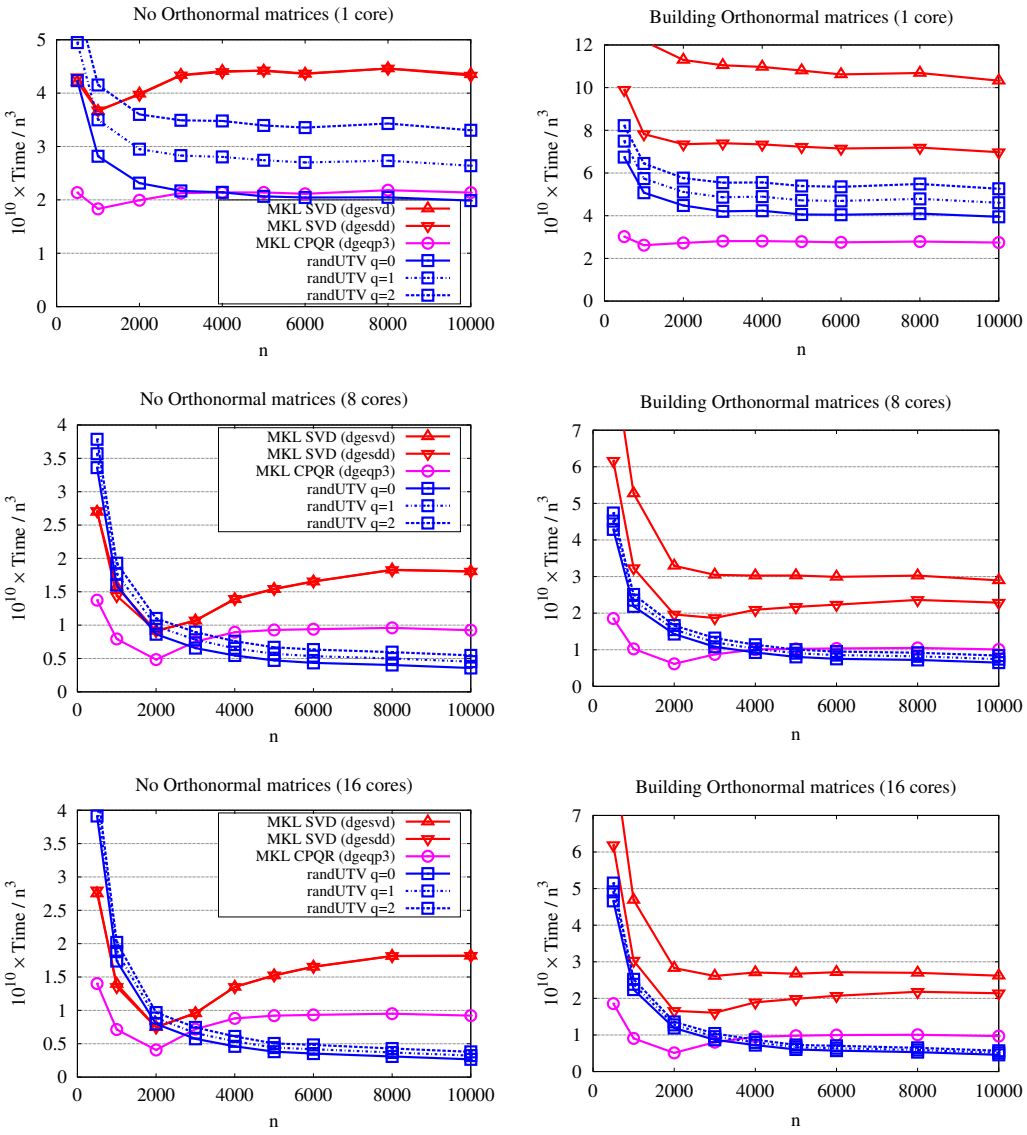


Figure 3.5: Computational cost (the lower, the better performances) of randUTV compared to the cost of the LAPACK drivers `dgesvd` and `dgesdd` (SVD), and `dgeqp3` (CPQR). The algorithms were applied to double-precision real matrices of size $n \times n$. The measured computational time divided by n^3 is plotted against n (e.g. the red lines are T_{svd}/n^3.) The three rows of plots correspond to executing on 1, 8, and 16 cores, respectively.

$j \leq 150$, and $\mathbf{D}(j,j) = 0.1/j$ for $j > 150$.

- *Matrix 4 (BIE):* This matrix is the result of discretizing a Boundary Integral Equation (BIE) defined on a smooth closed curve in the plane. To be precise, we discretized the so called "single layer" operator associated with the Laplace equation using a 6^{th} order quadrature rule designed by Alpert [2]. This operator is well-known to be ill-conditioned, which necessitates the use of a rank-revealing factorization in order to solve the corresponding linear system in as stable a manner as possible.

For each test matrix, we computed the error

$$e_k = \|\mathbf{A} - \mathbf{A}_k\| \tag{3.23}$$

where $\mathbf{A}_k$ is the rank-k approximation resulting from either of the three techniques discussed in this article:

$$\textit{SVD:} \qquad \mathbf{A}_k = \mathbf{A}_k^{\text{optimal}} = \mathbf{U}(:,1:k)\,\mathbf{D}(1:k,1:k)\,\mathbf{V}(:,1:k)^*, \tag{3.24}$$

$$\textit{CPQR:} \qquad \mathbf{A}_k = \mathbf{Q}(:,1:k)\,\mathbf{R}(1:k,:)\,\mathbf{P}^*, \tag{3.25}$$

$$\textit{randUTV:} \qquad \mathbf{A}_k = \mathbf{U}(:,1:k)\,\mathbf{T}(1:k,:)\,\mathbf{V}^*. \tag{3.26}$$

For randUTV, we ran the experiment with zero, one, and two steps of power iteration ($q = 0, 1, 2$). In addition to the direct errors defined by (3.23), we also calculated the relative errors, as defined via

$$e_k^{\text{relative}} = 100\% \times \frac{\|\mathbf{A} - \mathbf{A}_k\|}{\|\mathbf{A} - \mathbf{A}_k^{\text{optimal}}\|}. \tag{3.27}$$

The results are shown in Figures 3.6, 3.7, 3.8, and 3.9. The figures also report the errors resulting from a UTV factorization that G.W. Stewart proposed in [120], precisely for purposes of low-rank approximation and estimation of singular values. To be precise, Stewart builds a factorization $\mathbf{A} = \mathbf{U}\mathbf{L}\mathbf{V}^*$ where $\mathbf{U}$ and $\mathbf{V}$ are orthonormal, and $\mathbf{L}$ is lower triangular. The procedure is to first compute a CPQR factorization of $\mathbf{A}$ so that $\mathbf{A} = \mathbf{Q}_1\mathbf{R}_1\mathbf{P}_1^*$. Then compute a CPQR of the transpose of the upper triangular matrix $\mathbf{R}_1$ so that $\mathbf{R}_1^* = \mathbf{Q}_2\mathbf{L}^*\mathbf{P}_2^*$. Finally, set $\mathbf{U} = \mathbf{Q}_1\mathbf{P}_2$ and $\mathbf{V} = \mathbf{P}_1\mathbf{Q}_2$. The rank-$k$ approximation is then

$$\textit{QLP:} \qquad \mathbf{A}_k = \mathbf{U}\,\mathbf{L}(:,1:k)\,\mathbf{V}(:,1:k)^*. \tag{3.28}$$

Based on the errors shown in Figures 3.6–3.9, we make several empirical observations: (1) randUTV is much better than CPQR at computing low-rank approximations. Even when no power iteration ($q = 0$) is used, errors from randUTV are substantially smaller. When one or two steps of power iteration are taken ($q = 1$ or $q = 2$), the errors become close to optimal in all cases studied. (2) For the matrix with a gap in its singular values (cf. Figure 3.8), randUTV performs remarkably well in that both σ_{150} and σ_{151} are approximated to high accuracy. (3) The relative errors resulting from randUTV are consistently small, and much more reliably small than those resulting from CPQR. (4) Comparing the "QLP" factorization of Stewart [120] to randUTV, we see that Stewart's algorithm results in errors that are similar to those resulting from randUTV with $q = 0$. As soon as the power parameter is increased, randUTV tends to perform better. (We observe that in terms of speed, Stewart's QLP algorithm relies on two CPQR factorizations, which makes it much slower than randUTV.)

All error results shown in this section refer to errors measured in the spectral norm. The effects of including over-sampling in the algorithm, as described in Remark 12, is illustrated in numerical experiments given in an appendix in [89]. These experiments show that over-sampling does improve the error, but also that the improvement is almost imperceptible. For most applications, over-sampling is in our experience not worth the additional effort.

Remark 13 (Is randUTV rank-revealing?)**.** *While we do not have rigorous proofs that randUTV is rank-revealing, the numerical experiments in this section strongly indicate that it is, in the sense that a truncated UTV factorization approximates a given matrix to nearly optimal accuracy (in particular when a small number of steps of power iteration is used). We did not formally investigate whether randUTV is "strongly rank-revealing" in the sense of [64], but the numerical experiments certainly indicate that it does very well in this regard.*

3.6.3 Concentration of mass to the diagonal

In this section, we investigate our claim that randUTV produces a matrix $\mathbf{T}$ whose diagonal entries are close approximations to the singular values of the given matrix. (We recall that in the factorization

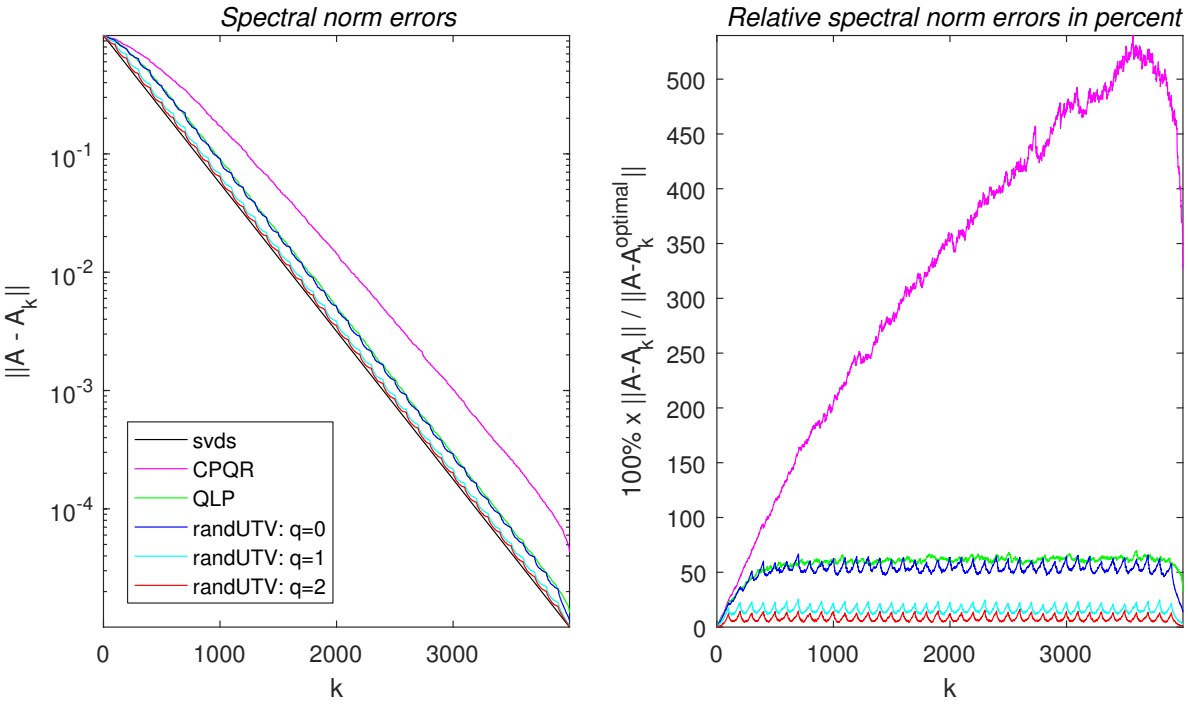


Figure 3.6: Rank-k approximation errors for the matrix "Fast Decay" (see Section 3.6.2) of size 4000×4000. The block size was $b = 100$. Left: Absolute errors in spectral norm. The black line marks the theoretically minimal errors. Right: Relative errors, as defined by (3.27).

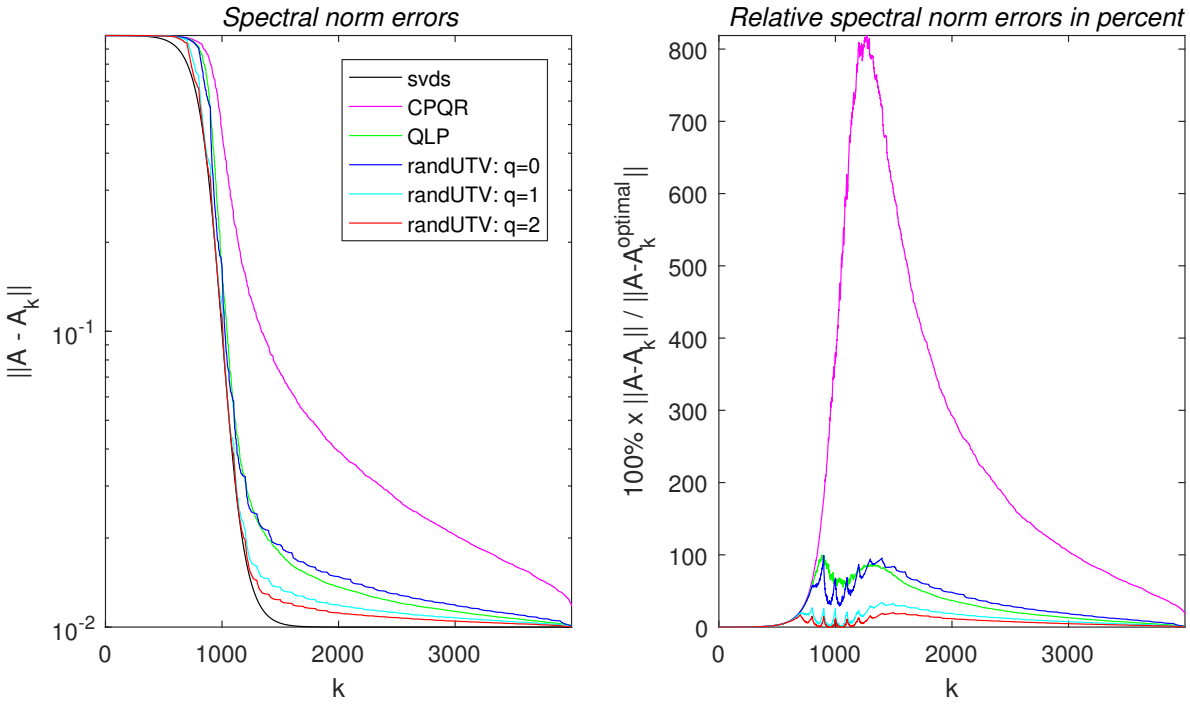


Figure 3.7: Rank-k approximation errors for the matrix "S-shaped decay" (see Section 3.6.2) of size 4000×4000. The block size was $b = 100$. Left: Absolute errors in spectral norm. The black line marks the theoretically minimal errors. Right: Relative errors, as defined by (3.27).

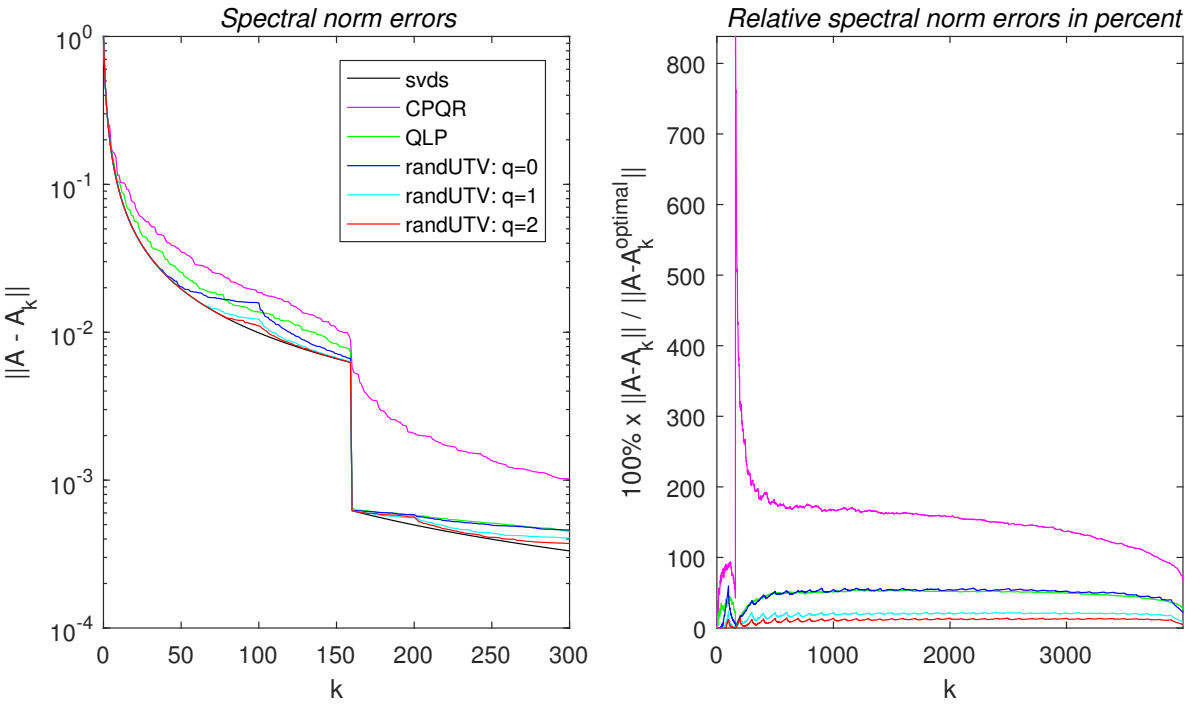


Figure 3.8: Rank-k approximation errors for the matrix "Gap" (see Section 3.6.2) of size 4000×4000. The block size was $b = 100$. Left: Absolute errors in spectral norm. The black line marks the theoretically minimal errors. (Observe that we zoomed in on the area around the gap.) Right: Relative errors, as defined by (3.27).

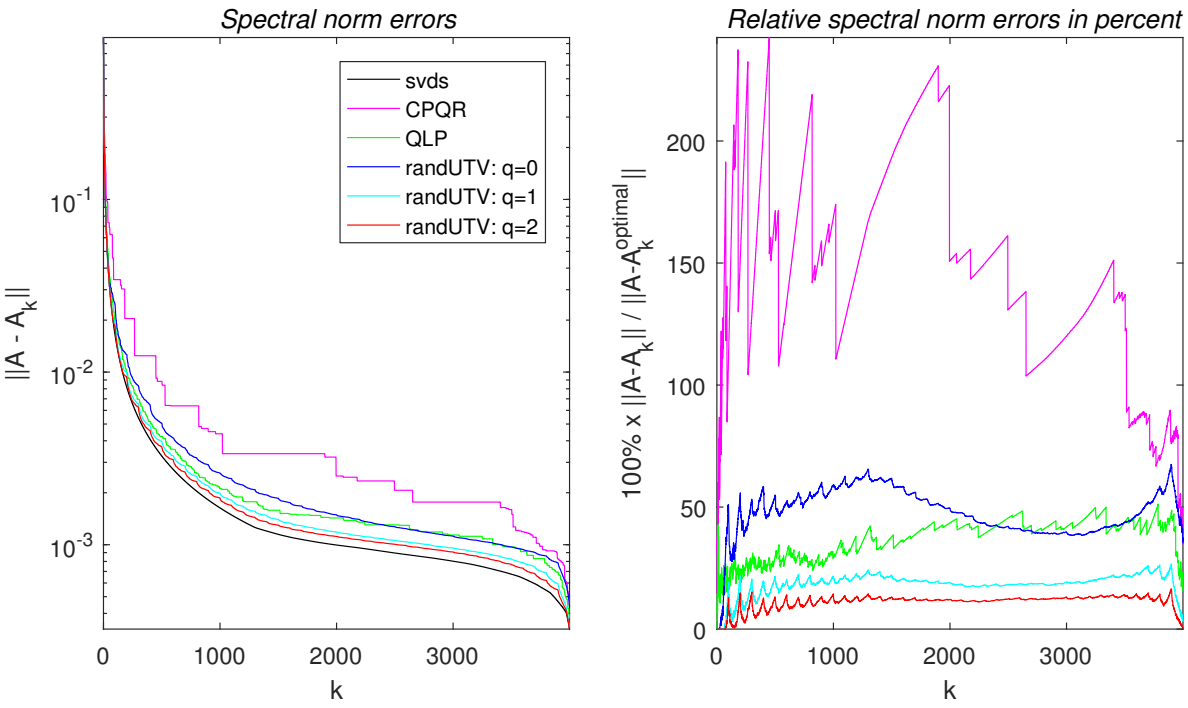


Figure 3.9: Rank-k approximation errors for the matrix "BIE" (see Section 3.6.2) of size 4000×4000. The block size was $b = 100$. Left: Absolute errors in spectral norm. The black line marks the theoretically minimal errors. Right: Relative errors, as defined by (3.27).

$\mathbf{A} = \mathbf{UTV}^*$, the matrix $\mathbf{T}$ is upper triangular, but with the entries above the diagonal very small in magnitude which forces the diagonal entries to approach the corresponding singular values.) Figure 3.10 shows the results for the four different test matrices described in Section 3.6.2, again for matrices of size $4\,000 \times 4\,000$. For reference, we also show the diagonal entries of the "R-factor" in a CPQR, and the diagonal entries of the "L-factor" in Stewart's QLP factorization [120]. We see that both Stewart's and our algorithm results in far better results than plain CPQR. randUTV roughly matches the accuracy of the QLP when $q = 0$, and does better as q is increased to one or two. (We recall that randUTV is much faster than the QLP method.)

3.7 Availability of code

Implementations of the discussed algorithm are available under 3-clause (modified) BSD license from:

`https://github.com/flame/randutv`

This repository includes two different implementations: one to be used with the LAPACK library [4], and the other one to be used with the libFLAME [104, 65] library.

3.8 Concluding remarks and future work

We have described a randomized algorithm we call "randUTV" for computing an economical alternative to the Singular Value Decomposition (SVD) of a matrix. The new method is much faster than classical algorithms for computing the SVD (cf. Section 3.6.1), and provides accuracy that is very close to optimal for tasks such as low rank approximation, or subspace identification (cf. Section 3.6.2). The new algorithm is blocked and executes well on modern communication-constrained hardware. It is incremental and can be used to compute partial factorizations.

Compared to the column pivoted QR factorization, which is commonly used as an economical alternative to the SVD for low rank approximation, randUTV is similar or better in terms of speed in modern architectures and much more accurate. Compared to methods designed specifically for low rank approximation such as the strongly rank-revealing QR factorizations [64], the QLP factorization [120], etc., randUTV

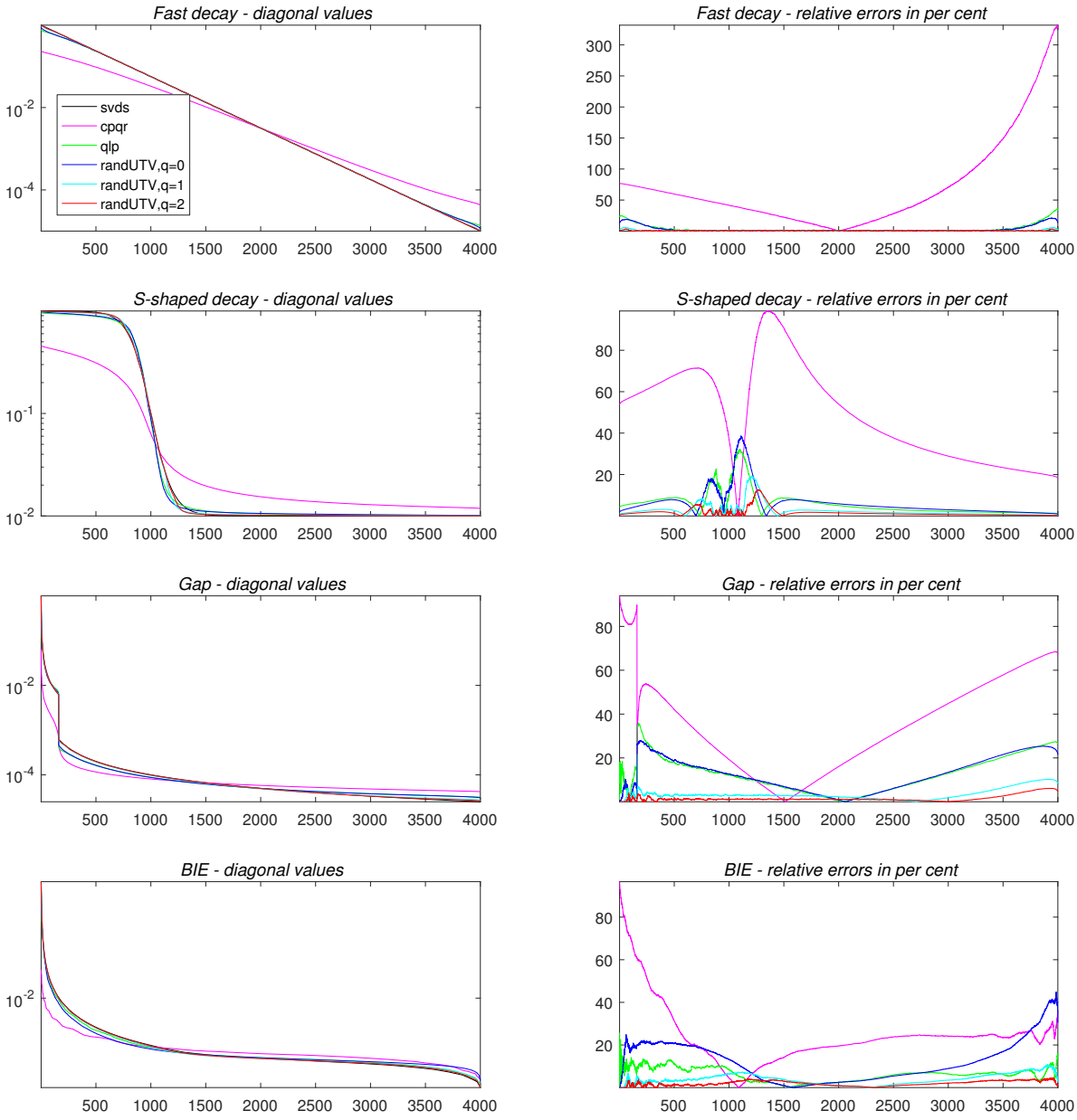


Figure 3.10: The results shown here illustrate how well randUTV approximates the singular values of four different matrices of size 4000×4000. For reference, we also include the diagonal values of the middle factors in a plain CPQR, and in Stewart's "QLP factorization" [120]. The left column shows the diagonal entries themselves. The right column shows relative errors, so that, e.g., the magenta line for CPQR plots $100\% \times |\mathbf{R}(i,i) - \sigma_i|/\sigma_i$ versus i.

provides similar or better accuracy, and is much faster.

The new method can also be viewed as an alternative to existing randomized methods for low rank approximation, as described in [69, 83, 91]. These methods excel when the numerical rank k of a matrix is much smaller than the matrix dimensions m and n. Moreover, the methods of [69, 83, 91] work best when the user has some rough idea of what k is in advance (although these requirements were relaxed substantially in [95]). In contrast, randUTV provides high speed for any rank, and does not need any a priori information about the numerical rank.

In this manuscript, we focused on the case of multicore CPUs with shared memory. We expect that the relative advantages of randUTV will be even more pronounced in more several communication-constrained environments such as GPUs, distributed memory parallel computers, or factorizations of matrices stored out of core. Work on variations of the method modified for such environments is currently under way.

Acknowledgements: The research reported was supported by DARPA, under the contract N66001-13-1-4050, and by the NSF, under the awards DMS-1407340 and DMS-1620472.

Chapter 4

Efficient algorithms for computing a rank-revealing UTV factorization on parallel computing architectures

The randomized singular value decomposition (RSVD) uses results from random linear algebra to efficiently build an approximate singular value decomposition of a matrix. The randUTV algorithm of Martinsson *et al.* builds on the ideas of the RSVD to produce a *full* factorization that provides low-rank approximations with near-optimal error. Because the bulk of randUTV is cast in terms of communication-efficient operations like matrix-matrix multiplication and unpivoted QR factorizations, it is faster than competing rank-revealing factorization methods like column pivoted QR in most high performance computational settings. In this article, optimized randUTV implementations are presented for both shared memory and distributed memory computing environments. For shared memory, randUTV is reconfigured as an algorithm-by-blocks, eliminating bottlenecks from data synchronization points to achieve acceleration over the standard "blocked algorithm" approach. The distributed memory implementation is based on the ScaLAPACK library. The performances of our new codes compares favorably with competing factorizations available on both shared memory and distributed memory architectures.

4.1 Introduction.

4.1.1 Overview.

Computational linear algebra faces significant challenges as high performance computing moves further away from the serial into the parallel. Classical algorithms were designed to minimize the number of floating point operations, and do not always lead to optimal performance on modern communication-bound

architectures. The obstacle is particularly apparent in the area of rank-revealing matrix factorizations. Traditional techniques based on column pivoted QR factorizations or Krylov methods tend to be challenging to parallelize well, as they are most naturally viewed as a sequence of matrix-vector operations.

In this paper, we describe techniques for efficiently implementing a randomized algorithm for computing a so-called rank-revealing UTV decomposition [89]. Given an input matrix A of size $m \times n$, the objective is to compute a factorization

$$\begin{array}{ccccccc} A & = & U & & T & & V^*, \\ m \times n & & m \times m & & m \times n & & n \times n \end{array} \tag{4.1}$$

where the middle factor T is upper triangular (or upper trapezoidal in the case $m < n$) and the left and right factors U, V have orthonormal columns. The factorization is rank-revealing in the sense that

$$\|A - U(:, 1:k)T(1:k, :)V^*\| \approx \inf\{\|A - B\| : B \text{ has rank } k\}. \tag{4.2}$$

The resulting factorization has entries above the diagonal in T that are very small in modulus, and in practice the diagonal entries of T are often excellent approximations to the singular values of A. A factorization of this type is useful for solving tasks such as low-rank approximation, for determined basis to approximations to the fundamental subspaces of A, for solving ill-conditioned or over/under-determined linear systems in a least-square sense, and for estimating the singular values of A.

The randomized UTV algorithm `randUTV` that we implement has several characteristics that in many environments make it preferable to classical rank-revealing factorizations like column pivoted QR (CPQR) and the singular value decomposition (SVD):

- It consistently produces matrix factors which yield low-rank approximations with accuracy comparable to the SVD. The particular use of randomization in the algorithm is essentially risk free. The reliability of the method is supported by theoretical analysis, as well as extensive numerical experiments.

- It casts most of its operations in terms of matrix-matrix multiplications, which are highly efficient in parallel computing environments. It was demonstrated in [89] that an straightforward blocked

implementation of `randUTV` executes faster than even highly optimized implementations of CPQR in symmetric multiprocessing systems. In this manuscript, we present an implementation that improves on the performances in [89] for SMP and obtain similar findings for distributed memory architectures.

- It processes the input matrix in sets of multiple columns, so it can be stopped part way through the factorization process if it is found that a requested tolerance has been met. If k columns end up having been computed, only $O(mnk)$ flops will have been expended.

In this manuscript, we present two efficient implementations of the `randUTV` algorithm: one for shared memory, and one for distributed memory. We first discuss the notation and some key mathematical tools that will be used throughout the paper in Section 4.2. In Section 4.3, we familiarize the reader with the `randUTV` algorithm that was recently described in [89]. Sections 4.4 and 4.5 describe the shared and distributed memory implementations that form the main contribution of this manuscript. In Section 4.6, we present numerical results that compare our implementations to highly optimized implementations of competing factorizations.

4.2 Preliminaries.

We use the notation $A \in \mathbb{R}^{m\times n}$ to specify that A is an $m \times n$ matrix with real entries. An *orthogonal* matrix is a square matrix whose column vectors each have unit norm and are pairwise orthogonal. $\sigma_i(A)$ represents the i-th singular value of A, and $\inf(A) = \min_i\{\sigma_i(A)\}$. The default norm $\|\cdot\|$ is the spectral norm. We also use the standard matrix indexing notation $A(c : d, e : f)$ to denote the submatrix of A consisting of the entries in the c-th through d-th rows of the e-th through f-th columns.

4.2.1 The Singular Value Decomposition (SVD)

Let $A \in \mathbb{R}^{m\times n}$ and $p = \min(m, n)$. Then, the singular value decomposition (SVD) of A is given by

$$\begin{array}{ccccccc} A & = & U & & \Sigma & & V^*, \\ m\times n & & m\times m & & m\times n & & n\times n \end{array}$$

where U and V are orthogonal and Σ is diagonal. We may also speak of the *economic* SVD of A, given by

$$\begin{array}{ccccc} A & = & U & \Sigma & V^*, \\ m \times n & & m \times p & p \times p & p \times n \end{array}$$

in which case U and V are not orthogonal (because they are not square) but their columns remain orthonormal.

A singular value decomposition exists for any real matrix and is "somewhat" unique.[1] The diagonal elements σ_i of Σ are thus called the *singular values* of A and satisfy $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_i \geq 0$, $i = 1, 2, \ldots, p$. The columns u_i and v_i of U and V are called the *left* and *right singular vectors*, respectively, of A.

A key fact about the SVD is that it provides theoretically optimal rank-k approximations to A. Specifically, the famous Eckart-Young-Mirsky Theorem [45, 98] states that given the SVD of a matrix A as described above and a fixed $1 \leq k \leq p$, we have that

$$\|A - U(:, 1:k)\Sigma(1:k, 1:k)(V(:, 1:k))^*\| = \inf\{\|A - B\| : B \text{ has rank } k\}.$$

A corollary of this result is that the subspace spanned by the leading k left and right singular vectors of A provides the optimal rank-k approximations to the column and row space, respectively, of A. That is to say, if QQ^* is the projection operator onto the subspace spanned by the left singular vectors of A, where $Q \in \mathbb{R}^{m \times k}$, and $x \in \text{Col}(A)$, then

$$\|x - QQ^*x\| = \inf\{\|x - Bx\| : B \text{ has rank } k\}.$$

The right singular vectors of A and the row space of A have the analogous result.

4.2.2 The QR decomposition

Given a matrix $A \in \mathbb{R}^{m \times n}$, let $p = \min(m, n)$. A QR decomposition of A is given by

$$\begin{array}{ccccc} A & = & Q & R, & \\ m \times n & & m \times m & m \times n & \end{array}$$

[1] For more details on the properties of the SVD, see, *e.g.* [57, 122, 116].

where Q is orthogonal and R is upper triangular. If $m > n$, then any QR can be reduced to the "economic" QR

$$\begin{array}{ccccc} A & = & Q & & R. \\ m \times n & & m \times n & n \times n & \end{array}$$

The standard algorithm for computing a QR factorization relies on Householder reflectors. We shall call this algorithm "HQR" in this article and refer the reader once again to textbooks such as [57, 122, 116] for a complete discussion. For this article, it is only necessary to note that the outputs of HQR are the following: an upper triangular matrix R of the QR factorization, and a unit lower triangular matrix $V \in \mathbb{R}^{m \times p}$ and a vector $v \in \mathbb{R}^p$ that can be used to build or to apply Q (see Section 4.2.3). In this article, we make critical use of the fact that for $m > n$, the leading p columns of Q form an orthonormal basis for the column space of A.

4.2.3 Compact WY representation of collections of Householder reflectors.

Consider a matrix $A \in \mathbb{R}^{n \times n}$, and let $H_i \in \mathbb{R}^{n \times n},\ i = 1, \ldots, b$ be Householder transformations. As a Householder transformation has the following structure: $H_i = I - \tau_i v_i v_i^*$, applying it to a matrix A requires a matrix-vector product and a rank-1 update. If all H_i are applied one after another, the computation requires $\mathcal{O}(bn^2)$ flops overall because of the special structure of the Householder transformations. Both operations are matrix-vector based, and therefore they do not render high performances on modern architectures.

If several Householder transformations must be applied, the product $H = H_b H_{b-1} \cdots H_2 H_1$ may be expressed in the form

$$H = I - WTW^*,$$

where $W \in \mathbb{R}^{n \times b}$ is lower trapezoidal and $T \in \mathbb{R}^{b \times b}$ is upper triangular. This formation of the product of Householder matrices is called the compact WY representation [113]. If the Householder transformations used to form each H_i are known, matrices W and T of the compact WY are inexpensive to compute. The above expression can be used to build the product HA:

$$HA = A - WTW^*A.$$

In this case, the cost is about the same, but only matrix-matrix operations are employed. Since on modern architectures matrix-matrix operations are usually much more efficient that matrix-vector operations, this approach will render higher performances. Recall that one flop (floating-point operation) in a matrix-matrix operation can be executed much (several times) faster than a flop in a matrix-vector operation.

4.3 The UTV factorization.

In this section, we discuss the rank-revealing UTV matrix factorization, establishing its usefulness in computational linear algebra and reviewing efficient algorithms for its computation. In Section 4.3.1, we review the classical UTV matrix decomposition, summarizing its benefits over other standard decompositions like column-pivoted QR and the SVD. In Section 4.3.2, we summarize recent work [89] that proposes a randomized blocked algorithm for computing this factorization.

4.3.1 The classical UTV factorization.

Let $A \in \mathbb{R}^{m\times n}$ and set $p = \min(m, n)$. A UTV decomposition of A is any factorization of the form

$$\begin{array}{ccccccc} A & = & U & & T & & V^*, \\ m\times n & & m\times m & & m\times n & & n\times n \end{array} \tag{4.3}$$

where T is trapezoidal and U and V are both orthogonal. T will always be upper trapezoidal in this paper. It is often desirable to compute a *rank-revealing* UTV (RRUTV) decomposition. For any $1 \le k \le p$, consider the partitioning of T

$$T \to \left(\begin{array}{c|c} T_{11} & T_{12} \\ \hline T_{21} & T_{22} \end{array}\right), \tag{4.4}$$

where T_{11} is $k \times k$. We say a UTV factorization is rank-revealing if

1. $\sigma_{\min}(T_{11}) \approx \sigma_k(A)$,
2. $\sigma_{\max}(T_{12}) \approx \sigma_{k+1}(A)$.

The flexibility of the factors in a UTV decomposition renders certain advantages over other canonical forms like CPQR and SVD (note that each of these are specific examples of UTV factorizations). Since

the right factor in CPQR is restricted to a permutation matrix, UTV has more freedom to provide better low-rank and subspace approximations. Also, since UTV does not have the SVD's restriction of diagonality on the middle factor, the UTV is less expensive to compute and has more efficient methods for updating and downdating (see, e.g. [114, 115, 6, 46, 102]).

4.3.2 The `randUTV` algorithm.

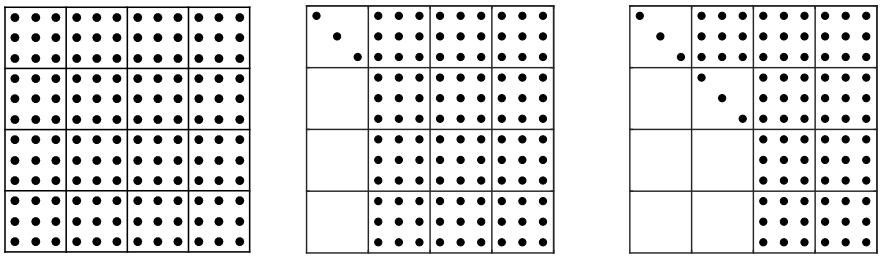
In [89], a new algorithm called `randUTV` was proposed for computing an RRUTV factorization. `randUTV` is designed to parallelize well, enhancing the RRUTV's viability as a competitor to both CPQR and the SVD for a wide class of problem types. It yields low-rank approximations comparable to the SVD at computational speeds that match, and in many cases outperform both CPQR and SVD. Unlike classical methods for building SVDs and RRUTVs, `randUTV` processes the input matrix by sets of b columns at one time and may therefore be stopped early (after processing k columns) to incur an overall cost of $\mathcal{O}(mnk)$.

The driving idea behind the structure of the `randUTV` algorithm is to build the middle factor T with a right-looking approach, that is, in each iteration multiple columns of T (a column block) are obtained simultaneously and only the right part of T is accessed. To illustrate, consider an input matrix $A \in \mathbb{R}^{m\times n}$, and let $p = \min(m, n)$. A block size parameter b with $1 \leq b \leq p$ must be chosen before `randUTV` begins. For simplicity, assume b divides p evenly. The algorithm begins by initializing $T^{(0)} := A$. Then, the bulk of the work is done in a loop requiring $\frac{p}{b}$ steps. In the i-th step, a new matrix $T^{(i+1)}$ is computed with

$$T^{(i+1)} := (U^{(i)})^* T^{(i)} V^{(i)},$$

for some orthogonal matrices $U^{(i)}$ and $V^{(i)}$. $U^{(i)}$ and $V^{(i)}$ are chosen such that:

- the leading ib columns of $T^{(i+1)}$ are upper triangular with $b \times b$ diagonal blocks on the main diagonal.
- using the partitioning in Equation 4.4 to define $T_{11}^{(i)}$ and $T_{22}^{(i)}$, we have $\sigma_{\min}(T_{11}^{(i)}) \approx \sigma_k(A)$ and $\sigma_{\max}(T_{22}^{(i)}) \approx \sigma_{k+1}(A)$ for $1 \leq k \leq ib$.
- $T_{11}^{(i)}(k, k) \approx \sigma_k(A)$ for $1 \leq k \leq ib$.



after 0 steps: $T^{(0)} := A$ | after 1 step: $T^{(1)} := (U^{(0)})^* T^{(0)} V^{(0)}$ | after 2 steps: $T^{(2)} := (U^{(1)})^* T^{(1)} V^{(1)}$

Figure 4.1: An illustration of the sparsity pattern followed by the first three $T^{(i)}$ for `randUTV` if $n = 12, b = 3$. `randUTV` continues until the entirety of $T^{(i)}$ is upper trapezoidal.

An example of the sparsity patterns for each $T^{(i)}$ is shown in Figure 4.1.

Once $T^{(i)}$ is upper triangular, U and V can then be built with

$$V := V^{(0)} V^{(1)} \cdots V^{(p/b-1)}$$

and

$$U := U^{(0)} U^{(1)} \cdots U^{(p/b-1)}.$$

In practice, $V^{(i)}$ and $U^{(i)}$ are constructed in two separate stages and applied to $T^{(i)}$ at different points in the algorithm. We will henceforth refer to these matrices as $V_\alpha^{(i)}, U_\alpha^{(i)}$ and $V_\beta^{(i)}, U_\beta^{(i)}$ for the first and second stages, respectively. Also, just one T matrix is stored, whose contents are overwritten with the new $T^{(i+1)}$ at each step. Similarly, in case the matrices U and V are required to be formed, only one matrix U and one matrix V would be stored. The outline for `randUTV` is therefore the following:

1. Initialize $T := A$, $V := I$, $U := I$.
2. for $i = 0, 1, \ldots, b/p - 1$:
   i. Build $V_\alpha^{(i)}$.
   ii. Update T and V: $T \leftarrow T V_\alpha^{(i)}$, $V \leftarrow V V_\alpha^{(i)}$.
   iii. Build $U_\alpha^{(i)}$.
   iv. Update T and U: $T \leftarrow (U_\alpha^{(i)})^* T$, $U \leftarrow U U_\alpha^{(i)}$.

v. Build $V_\beta^{(i)}$ and $U_\beta^{(i)}$ simultaneously.

vi. Update T, V, and U: $T \leftarrow (U_\beta^{(i)})^* T V_\beta^{(i)}$, $V \leftarrow V V_\beta^{(i)}$, $U \leftarrow U U_\beta^{(i)}$.

A matlab code for an easily readable (but inefficient) implementation of `randUTV` is given in Figure 4.2.

4.3.2.1 Building $V_\alpha^{(i)}$.

$V_\alpha^{(i)}$ is constructed to maximize the rank-revealing properties of the final factorization. Specifically, consider the partitioning at step i of matrices T and $V_\alpha^{(i)}$

$$T \rightarrow \left(\begin{array}{c|c} T_{11} & T_{12} \\ \hline T_{21} & T_{22} \end{array}\right), \; V_\alpha^{(i)} \rightarrow \left(\begin{array}{c|c} I & 0 \\ \hline 0 & (V_\alpha^{(i)})_{22} \end{array}\right),$$

where the top left block of each partition is $ib \times ib$. Then $V_\alpha^{(i)}$ is constructed such that the leading b columns of $(V_\alpha^{(i)})_{22}$ form an orthonormal approximate basis for the leading b right singular vectors of T_{22}. An efficient method for such a construction has been developed recently (see, *e.g.* [111, 69, 90, 91]) using ideas in random matrix theory. $V_\alpha^{(i)}$ is built as follows:

1. Draw a thin Gaussian random matrix $G^{(i)} \in \mathbb{R}^{(m-ib)\times b}$.
2. Compute $Y^{(i)} := (T_{22}^* T_{22})^q \, (T_{22})^* G^{(i)}$ for some small integer q.
3. Perform an unpivoted QR factorization on $Y^{(i)}$ to obtain an orthogonal $Q^{(i)}$ and upper triangular $R^{(i)}$ such that $Y^{(i)} = Q^{(i)} R^{(i)}$.
4. Set $(V_\alpha^{(i)})_{22} := Q^{(i)}$.

The parameter q, often called the "power iteration" parameter, determines the accuracy of the approximate basis found in $(V_\alpha^{(i)})_{22}$. Thus, raising q improves the rank-revealing properties of the resulting factorization but also increases the computational cost. For more details, see, *e.g.* [69].

4.3.2.2 Building $U_\alpha^{(i)}$.

$U_\alpha^{(i)}$ is constructed to satisfy both the rank-revealing and upper triangular requirements of the RRUTV. First, we partition $U_\alpha^{(i)}$

$$U_\alpha^{(i)} \to \left(\begin{array}{c|c} I & 0 \\ \hline 0 & (U_\alpha^{(i)})_{22} \end{array}\right).$$

To obtain $(U_\alpha^{(i)})_{22}$ such that $((U_\alpha^{(i)})_{22})^* T_{22}(:, 1:b)$ is upper triangular, we may compute the unpivoted QR factorization of $T_{22}(:, 1:b)$ to obtain $W^{(i)}, S^{(i)}$ such that $T_{22}(:, 1:b) = W^{(i)} S^{(i)}$. Next, observe that when the building of $U_\alpha^{(i)}$ occurs, the range of the leading b columns of T_{22} is approximately the same as that of the leading b left singular vectors of T_{22}. Therefore, the $W^{(i)}$ from the unpivoted QR factorization also forms an orthonormal approximate basis for the leading b left singular vectors of T_{22}, so $W^{(i)}$ is an approximately optimal choice of matrix from a rank-revealing perspective. Thus we let $(U_\alpha^{(i)})_{22} := W^{(i)}$.

4.3.2.3 Building $V_\beta^{(i)}$ and $U_\beta^{(i)}$.

$V_\beta^{(i)}$ and $U_\beta^{(i)}$ introduce more sparsity into T at low computational cost, pushing it closer to diagonality and thus decreasing $|T(k,k) - \sigma_k(A)|$ for $k = 1, \ldots, (i+1)b$. They are computed simultaneously by calculating the SVD of $T_{22}(1:b, 1:b)$ to obtain $U_{SVD}^{(i)}, V_{SVD}^{(i)}, D_{SVD}^{(i)}$ such that $T_{22}(1:b, 1:b) = U_{SVD}^{(i)} D_{SVD}^{(i)} (V_{SVD}^{(i)})^*$. Then we set

$$V_\beta^{(i)} := \left(\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & V_{SVD}^{(i)} & 0 \\ \hline 0 & 0 & I \end{array}\right), \; U_\beta^{(i)} := \left(\begin{array}{c|c|c} I & 0 & 0 \\ \hline 0 & U_{SVD}^{(i)} & 0 \\ \hline 0 & 0 & I \end{array}\right).$$

Following the update step $T \to (U_\beta^{(i)})^* T V_\beta^{(i)}$, $T_{22}(1:b, 1:b)$ is diagonal.

4.4 Efficient shared memory `randUTV` implementation.

Since the shared memory multicore computing architecture is ubiquitous in modern computing, it is therefore a prime candidate for an efficiently designed implementation of the `randUTV` algorithm presented

```
function [U,T,V] = randUTV(A,b,q)
  T = A;
  U = eye(size(A,1));
  V = eye(size(A,2));
  for i = 1:ceil(size(A,2)/b)
    I1 = 1:(b*(i-1));
    I2 = (b*(i-1)+1):size(A,1);
    J2 = (b*(i-1)+1):size(A,2);
    if (length(J2) > b)
      [UU,TT,VV] = stepUTV(T(I2,J2),b,q);
    else
      [UU,TT,VV] = svd(T(I2,J2));
    end
    U(:,I2)  = U(:,I2)*UU;
    V(:,J2)  = V(:,J2)*VV;
    T(I2,J2) = TT;
    T(I1,J2) = T(I1,J2)*VV;
  end
return
```

```
function [U,T,V] = stepUTV(A,b,q)
  G = randn(size(A,1),b);
  Y = A'*G;
  for i = 1:q
    Y = A'*(A*Y);
  end
  [V,~]    = qr(Y);
  [U,D,W]  = svd(A*V(:,1:b));
  T     = [D,U'*A*...
                V(:,(b+1):end)];
  V(:,1:b) = V(:,1:b)*W;
return
```

Figure 4.2: Matlab code for the algorithm `randUTV` that given an $m \times n$ matrix A computes its UTV factorization $A = UTV^*$. The input parameters b and q reflect the block size and the number of steps of power iteration, respectively. This code is simplistic in that products of Householder reflectors are stored simply as dense matrices, making the overall complexity $O(n^4)$. (Adapted from Figure 3 of [89].)

in Section 4.3.2. Martinsson *et al.* [89] provided an efficient blocked implementation of the randUTV factorization that was faster than competing rank-revealing factorizations, such as SVD and CPQR.

Blocked implementations for solving linear algebra problems are usually efficient since they are based on matrix-matrix operations. The ratio of flops to memory accesses in vector-vector operations and matrix-vector operations is usually very low: $\mathcal{O}(1)$ ($\mathcal{O}(n)$ flops to $\mathcal{O}(n)$ memory accesses, and $\mathcal{O}(n^2)$ flops to $\mathcal{O}(n^2)$ memory accesses, respectively). Performances are low on these types of operations since the memory becomes a significant bottleneck with such a low ratio. In contrast, the ratio of flops to memory accesses in matrix-matrix operations is much higher: $\mathcal{O}(n)$ ($\mathcal{O}(n^3)$ flops to $\mathcal{O}(n^2)$ memory accesses). This increased ratio provides much higher performances on modern computers since they require many flops per memory access.

As usual in many linear algebra codes, this blocked implementation of randUTV kept all the parallelism inside the BLAS library. However, the performances of implementations based on a parallel BLAS

are becoming increasingly inefficient as the number of cores increases in modern computers [106].

In Section 4.4.1, we discuss a scheme called algorithms-by-blocks for designing highly efficient algorithms on architectures with multiple/many cores. Section 4.4.2 explores the application of algorithms-by-blocks to `randUTV`. Finally, Sections 4.4.3 and 4.4.4 familiarize the reader with software used to implement algorithms-by-blocks and a runtime system to schedule the various matrix operations, respectively.

4.4.1 Algorithms-by-blocks: an overview.

`randUTV` is efficient in parallel computing environments mainly because it can be *blocked* easily. That is, it drives multiple columns of the input matrix A to upper triangular form in each iteration of its main loop. The design allows most of the operations to be cast in terms of the Level 3 BLAS (matrix-matrix operations), and more specifically in `xgemm` operations (matrix-matrix products). As vendor-provided and open-source multithreaded implementations of the Level 3 BLAS are highly efficient and close to the peak speed, `randUTV` renders high performances. Thus, a blocked implementation of `randUTV` relying largely on standard calls to parallel LAPACK and parallel BLAS was found to be faster than the highly optimized MKL CPQR implementation for a shared memory system, *despite `randUTV` having a much higher flop count than the CPQR algorithm* [89].

However, the benefits of pushing all parallelism into multithreaded implementations of the BLAS library are limited. Most high-performance blocked algorithms for computing factorizations (such as Cholesky, QR, LU, etc.) involve at least one task in each iteration that works on very little data, and therefore its parallelization does not render high performances. These tasks usually involve the processing of blocks with at least one small dimension b, where b is typically chosen to be 32 or 64, usually much smaller than the matrix dimensions. For instance, in the blocked Cholesky factorization this performance-limited task is the computation of the Cholesky factorization of the diagonal block, whereas in the blocked QR and LU factorizations this performance-limited part is the computation of the factorization of the current column block. Thus, since these tasks form a synchronization point, all but one core are left idle during these computations. For only four or five total cores, time lost is minimal. As the number of available cores increases, though, significant inefficiency builds up. The `randUTV` factorization is also affected by this problem, since each

iteration contains three tasks of this type: the QR factorization of matrix Y, the QR factorization of the current column block of T, and the SVD of the diagonal block of T.

We are therefore led to seek a technique other than blocking to obtain higher performances, although we will not abandon the strategy of casting most operations in terms of the Level 3 BLAS. The key lies in changing the method with which we aggregate multiple lower level BLAS flops into a single Level 3 BLAS operation. Blocked algorithms do this by raising the granularity of the algorithm's main loop. In `randUTV`, for instance, multiple columns of the input are typically processed in one iteration of the main loop. Processing one column at a time would require matrix-vector operations (Level 2 BLAS) in each iteration, but processing multiple columns at a time aggregates these into much more efficient matrix-matrix operations (Level 3 BLAS).

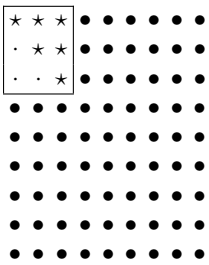
The alternative approach, called algorithms-by-blocks, is to instead raise the granularity of the *data*. With this method, the algorithm may be designed as if only scalar elements of the input are dealt with at one time. Then, the algorithm is transformed into Level 3 BLAS by conceiving of each scalar as a submatrix or block of size $b \times b$. Each scalar operation turns into a matrix-matrix operation, and operations in the algorithm will, at the finest level of detail, operate on usually a few (between one and four, but usually two or three) $b \times b$ blocks. Each operation on a few blocks is called a task. This arrangement allows more flexibility than blocking in ordering the operations, eliminating the bottleneck caused by the synchronization points in the blocking method. The performance benefits obtained by the algorithm-by-blocks approach with respect to the approach based on blocked algorithms for linear algebra problems on shared-memory architectures are usually significant [106, 22].

An algorithm-by-blocks for computing the `randUTV` requires that the QR factorization performed inside it works also on $b \times b$ blocks. In order to design this internal QR factorization process such that each unit of work requires only $b \times b$ submatrices, the algoritm-by-blocks for computing the QR factorization must employ an algorithm based on updating an existing QR factorization. We shall refer to this algorithm as `QR_AB`. We consider only the part of `QR_AB` that makes the first column of blocks upper triangular, since that is all that is required for `randUTV AB`. This work can be conceptualized as occurring in an iteration with a fixed number of steps or tasks.

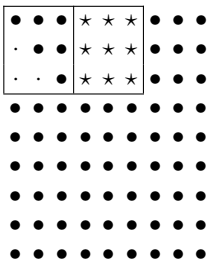
Figure 4.3 shows this process for a 9×9 matrix with block size 3. In this figure, the continuous lines show the 3×3 blocks involved in the current task, '$\bullet$' represents a non-modified element by the current task, '$\star$' represents a modified element by the current task, and '$\cdot$' represents a nullified element by the current task. The nullified elements are shown because, as usual, they store information about the Householder transformations that will be later used to apply these transformations. The first task, called *Compute_QR*, computes the QR factorization of the leading dense block A_{00}. The second task, called *Apply_left_Qt_of_dense_QR*, applies the Householder transformations obtained in the previous task (and stored in A_{00}) to block A_{01}. The third task performs the same operation onto A_{02}. The fourth task is the annhiliation of block A_{10}, which is called *Compute_QR_of_td_QR* (where 'td' means triangular-dense). The fifth task, called *Apply_left_Qt_of_td_QR*, applies the transformations of the previous task to blocks A_{01} and A_{11}. The sixth task performs the same operation onto A_{02} and A_{12}. Analogously, the seventh, eighth, and ninth tasks perform the same as tasks fourth, fifth, and sixth to the first and third row of blocks. By taking advantage of the zeros present in the factorizations for each iteration, a well-implemented `QR_AB` costs essentially no more flops than the traditional blocked unpivoted QR. The algorithm is described in greater detail in [106, 105, 22].

4.4.2 Algorithms-by-blocks for `randUTV`

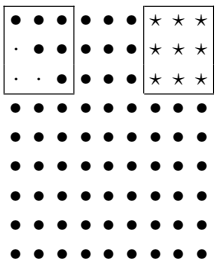
An algorithm-by-blocks for `randUTV`, which we will call `randUTV_AB`, performs mostly the same operations as the original. The key difference is that the operations' new representations allow greater flexibility in the order of completion. We will discuss in some detail how this plays out in the first step of the algorithm. First, choose a block size b (in practice, $b = 128$ or 256 works well). For simplicity, assume b divides both m and n evenly. Recall that at the beginning of `randUTV`, T is initialized with $T := A$.



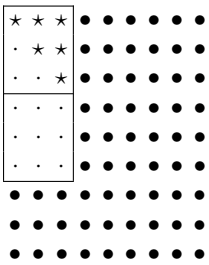
(1) After Compute_QR(A_{00})



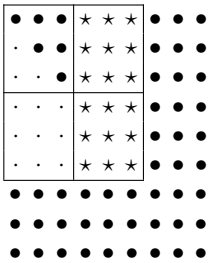
(2) After Apply_left_Qt_of_Dense_QR(A_{00}, A_{01})



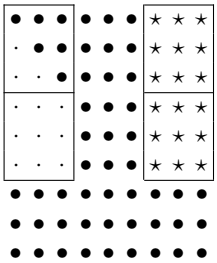
(3) After Apply_left_Qt_of_Dense_QR(A_{00}, A_{02})



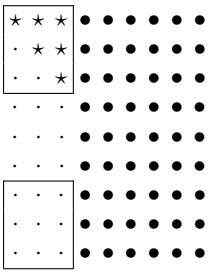
(4) After Compute_td_QR(A_{00}, A_{10})



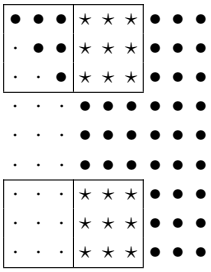
(5) After Apply_left_Qt_of_td_QR(A_{00}, A_{10}, A_{01}, A_{11})



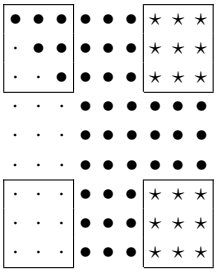
(6) After Apply_left_Qt_of_td_QR(A_{00}, A_{10}, A_{02}, A_{12})



(7) After Compute_td_QR(A_{00}, A_{20})



(8) After Apply_left_Qt_of_td_QR(A_{00}, A_{20}, A_{01}, A_{21})



(9) After Apply_left_Qt_of_td_QR(A_{00}, A_{20}, A_{02}, A_{22})

Figure 4.3: An illustration of the first tasks peformed by an algorithm-by-blocks for computing the QR factorization. The '$\bullet$' symbol represents a non-modified element by the current task, '$\star$' represents a modified element by the current task, and '$\cdot$' represents a nullified element by the current task (they are shown because they store information about the Householder transformations that will be later used to apply them). The continuous lines surround the blocks involved in the current task.

Consider a partitioning of the matrix T

$$T \rightarrow \left(\begin{array}{c|c|c|c} T_{11} & T_{12} & \cdots & T_{1N} \\ \hline T_{21} & T_{22} & \cdots & T_{2N} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline T_{M1} & T_{M2} & \cdots & T_{MN} \end{array}\right),$$

where each submatrix or block T_{ij} is $b \times b$, $N = n/b$, and $M = m/b$. Note that the rest of matrices (G, Y, U, and V) must also be accordingly partitioned. The submatrices T_{ij} (and those of the rest of matrices) are treated as the fundamental unit of data in the algorithm, so that each operation is expressed only in these terms. For the first step of the algorithm, for instance:

1. **Constructing $V^{(0)}$:** The first step, $Y^{(0)} = (T^*T)^q T^* G^{(0)}$, is broken into several tasks that each calculate the product of two blocks. In the simplified case where $q = 0$, we have $M \times N$ products of two blocks. The second step, the QR factorization of $Y^{(0)}$, uses an algorithm based on the idea of updating a QR factorization when more rows are added to the input matrix. Thus, the decomposition of each $Y_i^{(0)}$ is computed separately, and the resulting upper triangular factor $R^{(0)}$ is updated after each step. See, *e.g.* [67, 105, 106] for details on this approach to QR factorization.

2. **Constructing $U^{(0)}$:** This step requires an unpivoted QR factorization of the same size as $Y^{(0)}$, so same update-based algorithm used for $Y^{(0)}$ is used again here.

3. **Computing SVD of T_{11}:** This step is the same in `randUTV` and `randUTV_AB`. In both cases, T_{11} is interacted with as a single unit.

4. **Updating T:** The rest of `randUTV_AB` involves the updating of T, *i.e.* the computations $T \leftarrow TV^{(0)}$ and $T \leftarrow (U^{(0)})^* T$. The computations are broken down into separate stages such that the updating of each T_{ij} is a different task.

4.4.3 The FLAME abstraction for implementing algorithm-by-blocks.

A nontrivial obstacle to implementing an algorithm-by-blocks is the issue of programmability. Using the traditional approach of calls to a LAPACK implementation for the computational steps, keeping track of

indexing quickly becomes complicated and error-prone.

The FLAME (Formal Linear Algebra Methods Environment) project [65, 72] is one solution to this issue. FLAME is a framework for designing linear algebraic algorithms that departs from the traditional index-based-loop methodology. Instead, the input matrix is interacted with as a collection of submatrices, basing its loops on re-partitionings of the input.

The FLAME API [12] for the C language codifies these ideas, enabling a user of the API to code high performance implementations of linear algebra algorithms at a high level of abstraction. Furthermore, the methodology of the FLAME framework makes it a natural fit for use with an algorithm-by-blocks. Thus, the actual code for the implementation of `randUTV_AB` looks very similar to the written version of the algorithm given in Figure 4.4.

4.4.4 Scheduling the operations for an algorithm-by-blocks.

The runtime system called SuperMatrix [22] exposes and takes advantage of parallelism in `randUTV_AB`. To understand how SuperMatrix schedules and executes suboperations, consider the problem of factorizing a matrix of 2×2 blocks

$$A \leftarrow \left(\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array} \right),$$

where each block is of size $b \times b$. We will consider the case where the power iteration parameter $q = 0$ for simplicity.

Execution of the program proceeds in two phases: the analysis stage and the execution stage. In the first stage, instead of executing the code sequentially, the runtime builds a list of tasks recording the dependency information associated with each operation and placing it in a queue. An example of the queue built up by the runtime for `randUTV_AB` for the case that $A \in \mathbb{R}^{n \times n}$ and the block size is $b = n/2$ is given in Figure 4.5.

In the second stage, the scheduler/dispatcher stage, the tasks in the queue are dynamically scheduled and executed. Each task is executed as soon as its input data becomes available and a core is free to complete the work. Figure 4.6 illustrates the execution of the first half of `randUTV_AB` for a matrix $A \in \mathbb{R}^{n \times n}$ with

Algorithm: $[U, T, V] := \text{RANDUTV\_AB}(A, q, n_b)$

$V := \text{EYE}(n(A), n(A))$
$U := \text{EYE}(m(A), m(A))$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right), V \rightarrow \left(\begin{array}{c|c} V_L & V_R \end{array}\right), U \rightarrow \left(\begin{array}{c|c} U_L & U_R \end{array}\right)$

where A_{TL} is 0×0, V_L has 0 columns, U_L has 0 columns

while $m(A_{TL}) < m(A)$ **do**

Determine block size $b = \min(n_b, n(A_{BR}))$

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right),$$

$$\left(\begin{array}{c|c} V_L & V_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} V_0 & V_1 & V_2 \end{array}\right), \left(\begin{array}{c|c} U_L & U_R \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} U_0 & U_1 & U_2 \end{array}\right)$$

where A_{11} is $b \times b$, V_1 has b rows, U_1 has b rows

$$\begin{array}{ll}
G & := \text{GENERATE_IID_STDNORM_MATRIX}(m(A) - m(A_{00}), n_b) \\
Y & := \left(\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right)^* \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right)\right)^q \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right)^* G \\
[Y, T_V] & := \text{UNPIVOTED_QR}(Y) \\
\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) & := \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) - \left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) W_V T_V W_V^* \\
\left(\begin{array}{c|c} V_1 & V_2 \end{array}\right) & := \left(\begin{array}{c|c} V_1 & V_2 \end{array}\right) - \left(\begin{array}{c|c} V_1 & V_2 \end{array}\right) W_V T_V W_V^* \\
[\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right), T_U] & := \text{UNPIVOTED_QR}\left(\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)\right) \\
\left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right) & := \left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right) - W_U^* T_U^* W_U \left(\begin{array}{c} A_{12} \\ \hline A_{22} \end{array}\right) \\
\left(\begin{array}{c|c} U_1 & U_2 \end{array}\right) & := \left(\begin{array}{c|c} U_1 & U_2 \end{array}\right) - \left(\begin{array}{c|c} U_1 & U_2 \end{array}\right) W_U T_U W_U^* \\
[A_{11}, U_{SVD}, V_{SVD}] & := \text{SVD}(A_{11}) \\
A_{01} & := A_{01} V_{SVD} \\
A_{12} & := U_{SVD}^* A_{12} \\
V_1 & := V_1 V_{SVD} \\
U_1 & := U_1 U_{SVD}
\end{array}$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right),$$

$$\left(\begin{array}{c|c} V_L & V_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} V_0 & V_1 & V_2 \end{array}\right), \left(\begin{array}{c|c} U_L & U_R \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} U_0 & U_1 & U_2 \end{array}\right)$$

endwhile

Figure 4.4: The `randUTV` algorithm adapted for algorithms-by-blocks written with the FLAME methodology/notation. In this algorithm, W_V and W_U are the unit lower trapezoidal matrices stored below the diagonal of Y and $\left(\begin{array}{c} A_{11} \\ \hline A_{21} \end{array}\right)$, respectively.

block size $b = n/2$.

4.5 Efficient distributed memory `randUTV` implementation.

Distributed memory computing architectures are commonly used for solving large problems as they extend both memory and processing power over single systems. In this section, we discuss an efficient implementation of `randUTV` for distributed memory. In Section 4.5.1, we discuss the algorithmic overview and present the software used in the implementation. Section 4.5.2 familiarizes the reader with ScaLAPACK's structure and software dependencies. In Section 4.5.3 we review ScaLAPACK's data distribution scheme, and in Section 4.5.4 we describe how the building blocks of `randUTV` operate in the distributed memory environment.

4.5.1 Overview of implementation.

The distributed memory implementation of `randUTV` uses the standard blocked algorithm of [89] rather than the algorithm-by-blocks (as discussed in Section 4.4.2) since this methodology usually does not render high performances on distributed memory machines. Like in some other factorizations (QR, SVD, etc.), when applying `randUTV` to a matrix with $m \gg n$, it is best to perform an unpivoted QR factorization first and then perform the `randUTV` factorization on the resulting square triangular factor. This method is usually applied in other architectures such as shared memory.

The ScaLAPACK software library [18, 28, 30] was used in the presented implementation. This library provides much of the functionality of LAPACK for distributed memory environments. It hides most of the communication details from the developer with an object-based API, where each matrix's object information is passed to library routines. This design choice enhances the programmability of the library, enabling codes to be written similarly to a standard LAPACK implementation. However, as it is implemented in Fortran-77, its object orientation is not perfect and the programming effort is larger.

Operation	Operands		
	In		In/Out
Generate_normal_random			G_0
Generate_normal_random			G_1
Gemm_tn_oz: $C = A^*B$	A_{00}	G_0	Y_0
Gemm_tn_oz: $C = A^*B$	A_{01}	G_0	Y_1
Gemm_tn_oo: $C = C + A^*B$	A_{10}	G_1	Y_0
Gemm_tn_oo: $C = C + A^*B$	A_{11}	G_1	Y_1
Comp_dense_QR			Y_0, S_0
Copy	Y_0		E_0
Comp_td_QR			Y_0, Y_1, S_1
Apply_right_Q_of_dense_QR	E_0	S_0	A_{00}
Apply_right_Q_of_dense_QR	E_0	S_0	A_{10}
Apply_right_Q_td_QR	Y_1	S_1	A_{00}, A_{01}
Apply_right_Q_td_QR	Y_1	S_1	A_{10}, A_{11}
Comp_dense_QR			A_{00}, X_0
Copy	A_{00}		D_0
Comp_td_QR			A_{00}, A_{10}, X_1
Apply_left_Qt_of_dense_QR	D_0	X_0	A_{01}
Apply_left_Qt_of_td_QR	A_{10}	X_1	A_{01}, A_{11}
Keep_upper_triang			A_{00}
Set_to_zero			A_{10}
Svd_of_block			A_{00}, P_0, Q_0
Gemm_abta: $A = B^*A$	P_0		A_{01}
Svd_of_block			A_{11}, P_0, Q_0
Gemm_aabt: $A = AB^*$	Q_0		A_{01}

Figure 4.5: A list of the operations queued up by the runtime during the analyzer stage in the simplified case that the block size is $b = n/2$. The "In" column specifies pieces of required input data. The "In/Out" column specifies required pieces of data that will be altered upon completion of the operation. At run time, the operations may be completed in any order that does not violate the data dependencies encoded in the table.

Operation	Operands		
	In		In/Out
Generate_normal_random			G_0 ✓
Generate_normal_random			G_1 ✓
Gemm_tn_oz: $C = A^*B$	A_{00} ✓	G_0	Y_0 ✓
Gemm_tn_oz: $C = A^*B$	A_{01} ✓	G_0	Y_1 ✓
Gemm_tn_oo: $C = C + A^*B$	A_{10} ✓	G_1	Y_0
Gemm_tn_oo: $C = C + A^*B$	A_{11} ✓	G_1	Y_1
Comp_dense_QR			Y_0, S_0 ✓
Copy	Y_0		E_0 ✓
Comp_td_QR			Y_0, Y_1, S_1 ✓
Apply_right_Q_of_dense_QR	E_0	S_0	A_{00} ✓
Apply_right_Q_of_dense_QR	E_0	S_0	A_{10} ✓
Apply_right_Q_td_QR	Y_1	S_1	A_{00}, A_{01} ✓
Apply_right_Q_td_QR	Y_1	S_1	A_{10}, A_{11} ✓

(1) First half of the original table

Operands		
In		In/Out
A_{00} ✓	G_0 ✓	Y_0 ✓
A_{01} ✓	G_0 ✓	Y_1 ✓
A_{10} ✓	G_1 ✓	Y_0
A_{11} ✓	G_1 ✓	Y_1
Y_0		Y_0, S_0 ✓
		E_0 ✓
		Y_0, Y_1, S_1 ✓
E_0	S_0	A_{00} ✓
E_0	S_0	A_{10} ✓
Y_1	S_1	A_{00}, A_{01} ✓
Y_1	S_1	A_{10}, A_{11} ✓

(2) After second operation

Operands		
In		In/Out
A_{10} ✓	G_1 ✓	Y_0 ✓
A_{11} ✓	G_1 ✓	Y_1 ✓
		Y_0, S_0 ✓
Y_0		E_0 ✓
		Y_0, Y_1, S_1 ✓
E_0	S_0	A_{00} ✓
E_0	S_0	A_{10} ✓
Y_1	S_1	A_{00}, A_{01} ✓
Y_1	S_1	A_{10}, A_{11} ✓

(3) After fourth operation

Operands		
In		In/Out
		Y_0 ✓, S_0 ✓
Y_0		E_0 ✓
		Y_0, Y_1 ✓, S_1 ✓
E_0	S_0	A_{00} ✓
E_0	S_0	A_{10} ✓
Y_1	S_1	A_{00}, A_{01} ✓
Y_1	S_1	A_{10}, A_{11} ✓

(4) After sixth operation

Operands		
In		In/Out
Y_0 ✓		E_0 ✓
		Y_0, Y_1 ✓, S_1 ✓
E_0	S_0 ✓	A_{00} ✓
E_0	S_0 ✓	A_{10} ✓
Y_1	S_1	A_{00}, A_{01} ✓
Y_1	S_1	A_{10}, A_{11} ✓

(5) After seventh operation

Figure 4.6: An illustration of the execution order of the first seven steps of the first half of `randUTV_AB` for an $n \times n$ matrix using the SuperMatrix runtime system when the block size is $n/2$. A check mark '✓' indicates the value is available. The execution order may change depending on the number of available cores in the system.

4.5.2 Software dependencies.

ScaLAPACK (scalable LAPACK) was designed to be portable to a variety of computing distributed memory architectures and relies on only two external libraries (since PBLAS is considered an internal module). The first one is the sequential BLAS (Basic Linear Algebra Subroutines) [81, 39, 37], providing specifications for the most common operations involving vectors and matrices. The second one is the BLACS (Basic Linear Algebra Communication Subroutines), which, as the name suggests, is a specification for common matrix and vector communication tasks [5].

The PBLAS library is a key module inside ScaLAPACK. It comprises most of BLAS routines rewritten for use in distributed memory environments. This library is written using a combination of the sequential BLAS library and the BLACS library. Just as the BLAS library contains the primary building blocks for LAPACK routines, the PBLAS library contains the foundation for the routines in ScaLAPACK. The diagram in Figure 4.7 illustrates the dependencies of the ScaLAPACK modules. The PBLAS library serves a dual purpose in the library. First, because the PBLAS library mirrors the sequential BLAS in function, the top level of code in main ScaLAPACK routines look largely the same as the corresponding LAPACK routines. Second, the PBLAS library adds a layer of flexibility to the code regarding the mapping of operations. Traditionally, one process is assigned to each core during execution, but with a parallel BLAS implementation, a combination of processes and threads may be used. This adjustability gives more options when mapping processes onto cores just before the program execution starts.

4.5.3 ScaLAPACK data distribution scheme.

The strategy for storing data in a distributed memory computation has a significant impact on the communication cost and load balance during computation. All ScaLAPACK routines assume the so-called "block-cyclic distribution" scheme [29]. Since it involves several user-defined parameters, understanding this method is vital to building an efficient implementation.

The block-cyclic distribution scheme involves four parameters. The first two, m_b and n_b, define the block size, *i.e.* the dimensions of the submatrices used as the fundamental unit for communication among

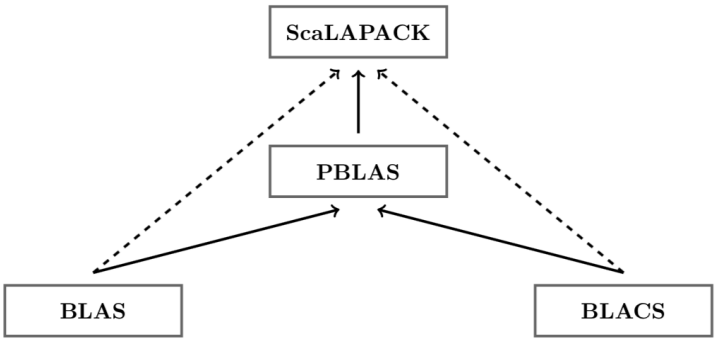


Figure 4.7: The dependencies of the modules of ScaLAPACK. A solid line means the dependence occurs in the main routines (*drivers*), and a dashed line means the dependence only occurs in auxiliary routines.

processes. Despite this flexibility, nearly all the main routines usually employ $m_b = n_b$ for the purpose of simplicity. The last two parameters, typically called P and Q, determine the shape of the logical process grid.

To understand which elements of the input matrix A are stored in which process, we may visualize the matrix as being partitioned into "tiles." In the simple case where $m_B P$ and $n_b Q$ divide m and n, respectively, every tile is of uniform size. Each tile is composed of $P \times Q$ blocks, each of size $m_b \times n_b$. Finally, every process is assigned a position on the tile grid. The block in that position on every tile is stored in the corresponding process. For example, the block in the $(0, 0)$ spot in each tile belongs with the first process P_0, the block in the $(0, 1)$ spot in each tile belongs to the second process P_1, and so on. An example is given in Figure 4.8 to demonstrate this.

4.5.4 Building blocks of `randUTV`

In this section we examine further the `randUTV` algorithm in order to understand which portions of the computation are most expensive (when no orthonormal matrices are built) and how these portions perform in the distributed memory environment. Judging by numbers of flops required, the three portions of the computation that take the most time are the following:

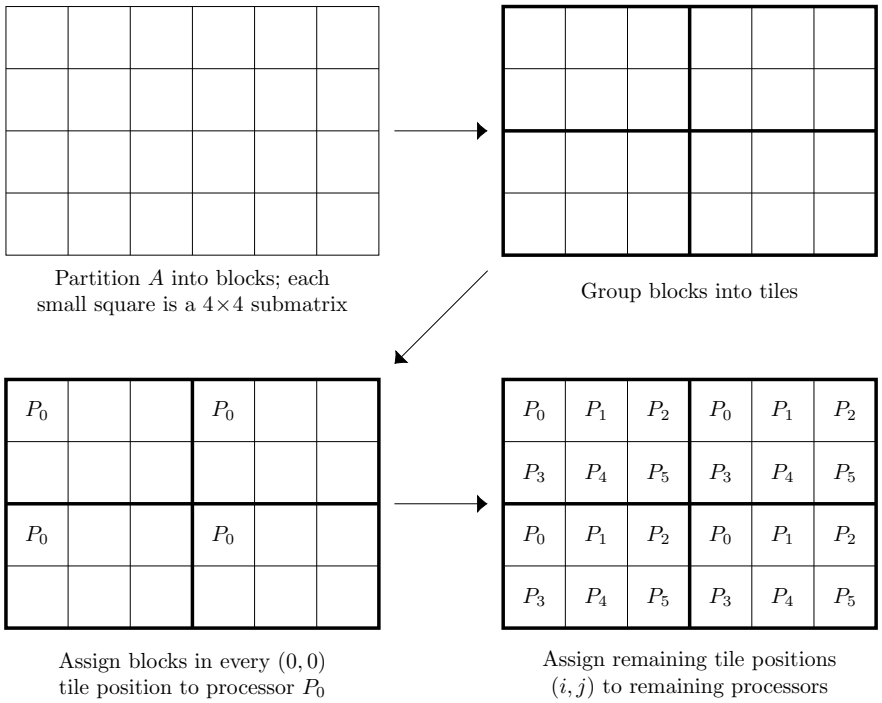


Figure 4.8: A depiction of the block-cyclic data distribution method for a matrix A with $m = 16, n = 24$. The parameters for this distribution are $m_b = 4, n_b = 4, P = 2, Q = 3$. Each tile is a grid of $P \times Q$ blocks.

1. applying $V^{(i)}$, stage α to A,

2. applying $U^{(i)}$, stage α to A,

3. building Y.

To determine the fundamental operations involved in items i and ii, first recall that $V^{(i)}$ and $U^{(i)}$ are both formed from a Householder reduction on matrices with b columns to upper trapezoidal form. As such, we may express them in the so-called compact WY form (see Section 4.2.3) as

$$V^{(i)} = I - W_V^{(i)} T_V^{(i)} (W_V^{(i)})^*,$$
$$U^{(i)} = I - W_U^{(i)} T_U^{(i)} (W_U^{(i)})^*,$$

where $T_V^{(i)}, T_U^{(i)} \in \mathbb{R}^{b\times b}$ are upper triangular, and $W_V^{(i)} \in \mathbb{R}^{n\times b}$ and $W_U^{(i)} \in \mathbb{R}^{m\times b}$ are lower trapezoidal. Thus the computations $AV^{(i)}$ and $(U^{(i)})^*A$ each require three matrix-matrix multiplications where one

dimension of the multiplication is small (recall $b \ll n$). Note that the first computation ($AV^{(i)}$) is more expensive than the second one ($(U^{(i)})^* A$) because the first one processes all the rows of A (the right part of A), whereas the second one only processes some rows of A (the bottom right part of A).

It is now evident that items i and ii use primarily `xgemm` and `xtrmm` from the BLAS. Furthermore, item iii is strictly a series of `xgemm` operations, so we see that matrix-matrix multiplications form the dominant cost within `randUTV`.

`xgemm` for distributed memory, which in the PBLAS library is titled `pxgemm`, is well-suited for efficiency in this environment. In the reference implementation of PBLAS, `pxgemm` may execute one of three different algorithms for matrix multiplication:

1. `pxgemmAB`: The outer-product algorithm is used; matrix C remains in place.
2. `pxgemmBC`: The inner-product algorithm is used; matrix A remains in place.
3. `pxgemmAC`: The inner-product algorithm is used; matrix B remains in place.

`xgemm` chooses among the algorithms by estimating the communication cost for each, depending on matrix dimensions and parameters of the storage scheme. The inherent flexibility of the matrix-matrix multiply enables good `pxgemm` implementations to overlap the communication with the processing of flops. Thus, `randUTV` for distributed memory obtains better speedups when more cores are added than competing implementations of SVD and CPQR algorithms for distributed memory.

4.6 Performance analysis

In this section, we investigate the speed of our new implementations of the algorithm for computing the randUTV factorization, and compare it to the speeds of highly optimized methods for computing the SVD and the column pivoted QR (CPQR) factorization. In all the experiments double-precision real matrices were processed.

To fairly compare the different implementations being assessed, the flop count or the usual flop rate could not be employed since the computation of the SVD, the CPQR, and the randUTV factorizations require a very different number of flops (the dominant n^3-term in the asymptotic flop count is very different).

Absolute computational times are not shown either since they vary greatly because of the large range of matrix dimensions employed in the experiments. Therefore, scaled computational times (absolute computational times divided by n^3) are employed. Hence, the lower the scaled computational times, the better the performances are. Since all the implementations being assessed have asymptotic complexity $O(n^3)$ when applied to an $n \times n$ matrix, these graphs better reveal the computational efficiency. Those scaled times are multiplied by a constant (usually 10^{10}) to make the figures in the vertical axis more readable.

Although most of the plots show scaled computational times, a few plots show speedups. The speedup is usually computed as the quotient of the time obtained by the serial implementation (on one core) and the time obtained by the parallel implementation (on many cores). Thus, this concept communicates how many times faster the parallel implementation is compared to the serial one. Hence, the higher the speedups, the better the performances of the parallel implementation are. This measure is usually very useful in checking the scalability of an implementation. Note that in this type of plots every implementation compares against itself on one core.

4.6.1 Computational speed on shared-memory architectures

The computer used in the experiments on shared-memory architectures was based on Intel processors. It featured two Intel Xeon® CPUs E5-2695 v3 (2.30 GHz), with 28 cores and 128 GiB of RAM in total. In this computer the so-called **Turbo Boost** mode of the two CPUs was turned off in our experiments.

Its OS was GNU/Linux (Version 2.6.32-504.el6.x86_64). GCC compiler (version 6.3.0 20170516) was used. Intel(R) Math Kernel Library (MKL) Version 2018.0.1 Product Build 20171007 for Intel(R) 64 architecture was employed since LAPACK routines from this library usually deliver much higher performances than LAPACK routines from the Netlib repository. When using routines of MKL's LAPACK, optimal block sizes determined by that software were employed.

In a few experiments, in addition to MKL's LAPACK routines, we also assessed Netlib's LAPACK 3.4.0 routines. In this case, the NETLIB term is used. When using routines of Netlib's LAPACK, several block sizes were employed and best results were reported. For the purpose of a fair comparison, these routines from Netlib were linked to the BLAS library from MKL.

All the matrices used in the experiments were randomly generated. Similar results were obtained on other types of matrices, such as those generated with the following singular values: $\sigma_i = 1, 1 \leq i < n, \sigma_p = 2 \cdot 10^{-12}$, and random orthonormal transformations.

Unless explicitely stated otherwise, all the experiments employed the 28 cores in the computer.

The following implementations were assessed in the experiments of this subsection:

- MKL SVD: The routine called `dgesvd` from MKL's LAPACK was used to compute the Singular Value Decomposition.

- NETLIB SVD: Same as the previous one, but the code for computing the SVD from Netlib's LAPACK was employed, instead of MKL's.

- MKL SDD: The routine called `dgesdd` from MKL's LAPACK was used to compute the Singular Value Decomposition. Unlike the previous SVD, this one uses the divide-and-conquer approach. This code is usually faster, but it requires a much larger auxiliary workspace when the orthonormal matrices are built (about four additional matrices of the same dimension as the matrix being factorized).

- NETLIB SDD: Same as the previous one, but the code for computing the SVD with the divide-and-conquer approach from Netlib's LAPACK was employed, instead of MKL's.

- MKL CPQR: The routine called `dgeqp3` from MKL's LAPACK was used to compute the column-pivoting QR factorization.

- RANDUTV PBLAS (randUTV with parallel BLAS): This is the traditional implementation for computing the randUTV factorization that relies on the parallel BLAS to take advantage of all the cores in the system. The parallel BLAS library from MKL was employed with these codes for the purpose of a fair comparison. Our implementations were coded with `libflame` [125, 124] (Release 11104).

- RANDUTV AB (randUTV with Algorithm-by-Blocks): This is the new implementation for computing the randUTV factorization by scheduling all the tasks to be computed in parallel, and

then executing them with serial BLAS. The serial BLAS library from MKL was employed with these new codes for the purpose of a fair comparison. Our implementations were coded with `libflame` [125, 124] (Release 11104).

- MKL QR: The routine called `dgeqrf` from MKL's LAPACK was used to compute the QR factorization. Although this routine does not reveal the rank, it was included in some experiments as a performance reference for the others.

For every experiment, two plots are shown. The left plot shows the performances when no orthonormal matrices are computed. In this case, just the singular values are computed for the SVD, just the upper triangular factor R is computed for the CPQR and the QR, and just the upper triangular factor T is computed for the randUTV. In contrast, the right plot shows the performances when all orthonormal matrices are explicitly formed in addition to the singular values (SVD), the upper triangular matrix R (CPQR), or the upper triangular matrix T (randUTV). In this case, matrices U and V are computed for the SVD and the randUTV, and matrix Q is computed for the CPQR and the QR. The right plot slightly favors CPQR and QR since only one orthonormal matrix is formed.

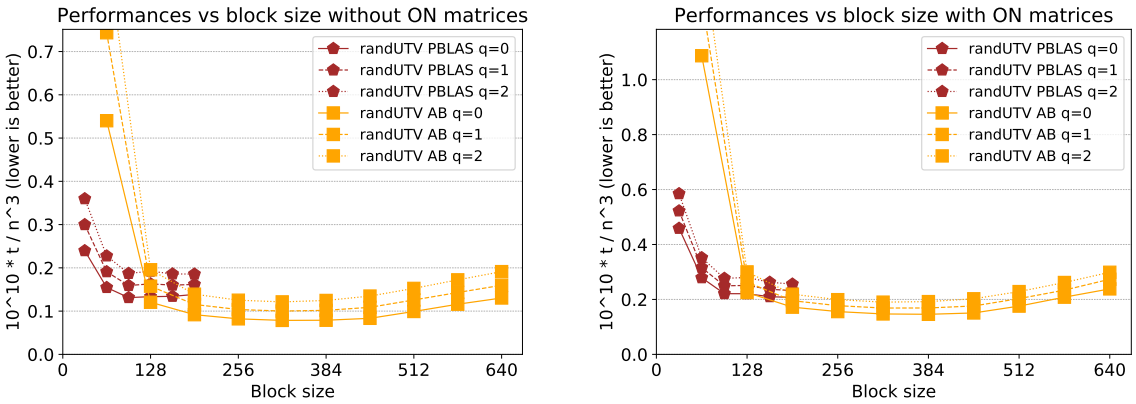


Figure 4.9: Performances of randUTV implementations versus block size on matrices of dimension 14000×14000.

Figure 4.9 shows the scaled computational times obtained by both implementations for computing the randUTV factorization (RANDUTV PBLAS and RANDUTV AB) on several block sizes when processing

matrices of dimension 14000×14000. The aim of these two plots is to determine the optimal block sizes. The other factorizations (SVD and CPQR) are not shown since in those cases we used the optimal block sizes determined by Intel's software. Optimal block sizes were around 128 for RANDUTV PBLAS; on the other hand, optimal block sizes were around 384 for RANDUTV AB.

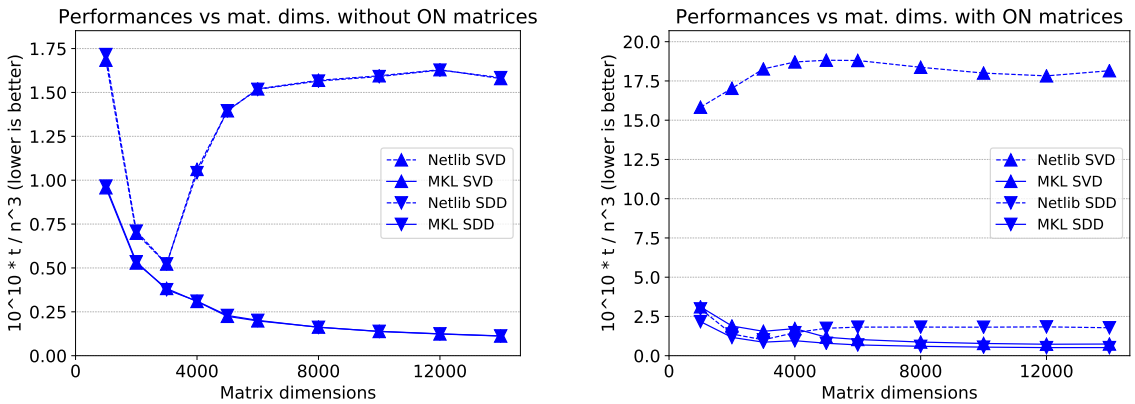


Figure 4.10: Performances versus matrix dimensions for SVD implementations for both Netlib and MKL libraries.

Figure 4.10 compares the performances of four implementations for computing the SVD factorization: MKL SVD (usual SVD from the MKL library), MKL SDD (divide-and-conquer SVD from the MKL library), NETLIB SVD (usual SVD from the Netlib library), and NETLIB SDD (divide-and-conquer SVD from the Netlib library). Performances are shown with respect to matrix dimensions. Block sizes similar to those in the previous figure were used for Netlib's routines and the best results were reported. When no orthonormal matrices are computed, both the traditional SVD and the divide-and-conquer SVD render similar performances for this matrix type. In this case, MKL routines are up to 14.1 times as fast as Netlib's routines. When orthonormal matrices are computed, the traditional SVD is much slower than the divide-and-conquer SVD. In this case, the MKL SVD routine is up to 24.4 times as fast as the NETLIB SVD, and the MKL SDD routine is up to 3.4 times as fast as the NETLIB SDD. As can be seen, MKL's codes for computing the SVD are up to more than one order of magnitude faster than Netlib's codes, thus showing the great performances achieved by Intel. This is a remarkable achievement for so complex codes. Outperforming these highly optimized codes can be really a difficult task.

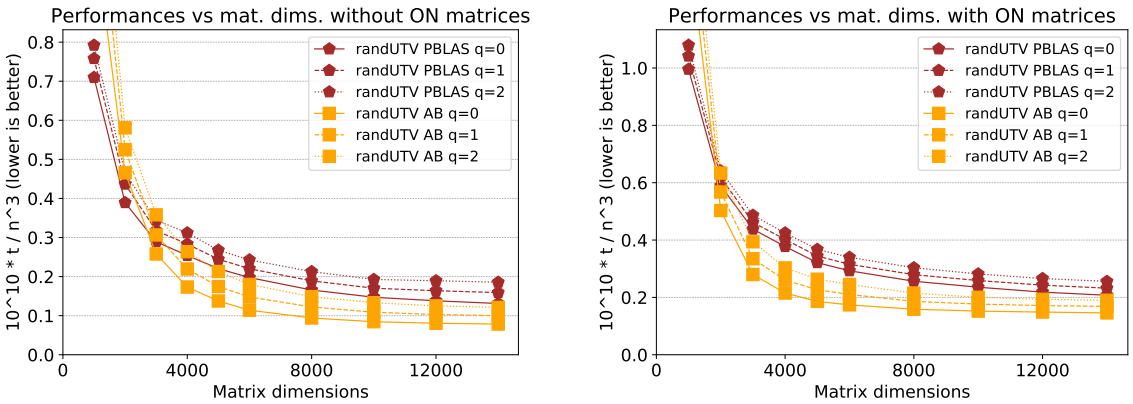


Figure 4.11: Performances versus matrix dimensions for randUTV implementations.

Figure 4.11 compares the performances of both implementations of randUTV (RANDUTV PBLAS and RANDUTV AB) as a function of matrix dimensions. In both implementations, several block sizes were tested (see above), and best results were reported. When no orthonormal matrices are built, RANDUTV AB is between 1.7 ($q = 0$) and 2.4 ($q = 2$) times as fast as RANDUTV PBLAS for the largest matrix size. When orthonormal matrices are built, RANDUTV AB is between 1.4 ($q = 0$) and 1.8 ($q = 2$) times as fast as RANDUTV PBLAS for the largest matrix size.

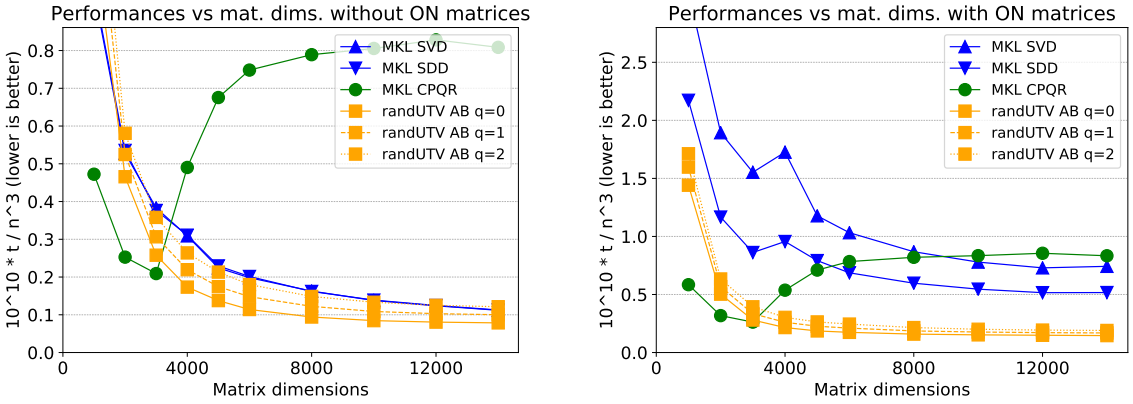


Figure 4.12: Performances versus matrix dimensions for the best implementations.

Figure 4.12 shows the performances of the best implementations as a function of the matrix dimensions. When no orthonormal matrices are built, RANDUTV AB is between 1.4 ($q = 0$) and 0.9 ($q = 2$) times as fast as MKL SDD for the largest matrix size. When orthonormal matrices are built, RANDUTV

AB is between 5.1 ($q = 0$) and 3.9 ($q = 2$) times as fast as MKL SDD for the largest matrix size. Recall that in this last case, in addition to being slower, MKL SDD requires a much larger auxiliary workspace than RANDUTV AB (about four times as large). The speeds of RANDUTV AB, MKL SVD and MKL SDD are so remarkable that they are even much faster than MKL CPQR, a factorization that requires much fewer flops.

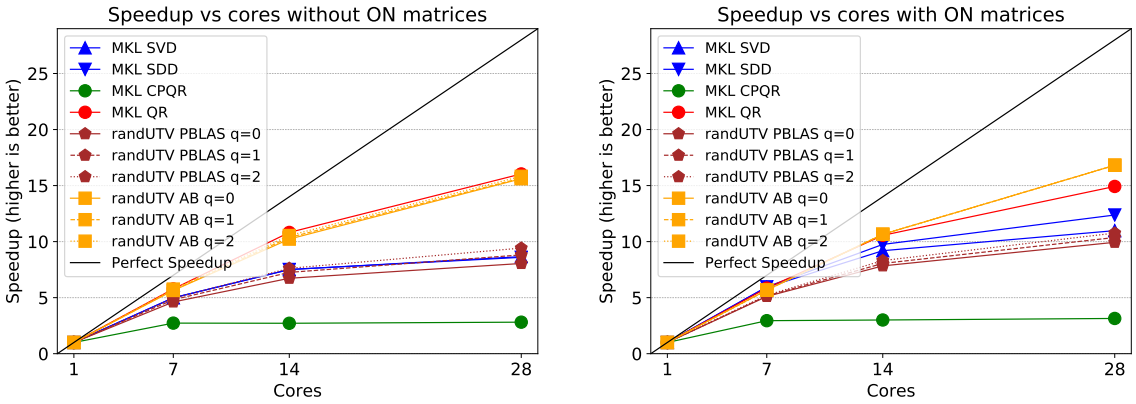


Figure 4.13: Speedups versus number of cores for the best implementations on matrices of dimension 14000×14000.

Figure 4.13 shows the speedups obtained by the best implementations on matrices of dimension 14000×14000. Recall that in this plot every implementation compares against itself on one core. When no orthonormal matrices are built, RANDUTV AB achieves a speedup between 15.6 ($q = 0$) and 15.8 ($q = 2$) on 28 cores. When orthonormal matrices are built, RANDUTV AB achieves a speedup of 16.8 on 28 cores. In both cases, its efficiency (speedup divided by the number of cores) is higher than 50 %.

The speedups of MKL SVD and MKL SDD are much lower than those of the RANDUTV AB when no orthonormal matrices are built, and a bit lower when orthonormal matrices are built. The speedups of MKL CPQR are usually much lower than those of the other factorizations in both cases, thus not being scalable at all. As can be seen, the speedups of RANDUTV AB are similar (left plot) or even higher (right plot) that those of the QR factorization. To conclude this analysis, the scalability of RANDUTV AB is similar to those of the QR factorization, and much higher than the rest of the implementations.

In conclusion, RANDUTV AB is the clear winner over competing factorization methods in terms

of raw speed when orthonormal matrices are required and the matrix is not too small ($n \gtrsim 4000$). In terms of scalability, RANDUTV AB outperforms the competition as well. Also, the algorithm-by-blocks implementation gives noticeable speedup over the blocked PBLAS version. That RandUTV AB can compete with MKL SVD at all in terms of speed is remarkable, given the large effort usually invested by Intel on its sofware. This is evidenced by the fact that the MKL CPQR is left in the dust by both MKL SVD and RANDUTV AB, each of which costs far more flops than MKL CPQR. The scalability results of RANDUTV AB and its excellent timings evince its potential as a high performance tool in shared memory computing.

4.6.2 Computational speed on distributed-memory architectures

The experiments on distributed-memory architectures reported in this subsection were performed on a cluster of HP computers. Each node of the cluster contained two Intel Xeon® CPU X5560 processors at 2.8 GHz, with 12 cores and 48 GiB of RAM in total. The nodes were connected with an Infiniband 4X QDR network. This network is capable of supporting 40 Gb/s signaling rate, with a peak data rate of 32 Gb/s in each direction.

Its OS was GNU/Linux (Version 3.10.0-514.21.1.el7.x86_64). Intel's `ifort` compiler (Version 12.0.0 20101006) was employed. LAPACK and ScaLAPACK routines were taken from the Intel(R) Math Kernel Library (MKL) Version 10.3.0 Product Build 20100927 for Intel(R) 64 architecture, since this library usually delivers much higher performances than LAPACK and ScaLAPACK codes from the Netlib repository.

All the matrices used in these experiments were randomly generated since they are much faster to be generated, and the cluster was being heavily loaded by other users.

The following implementations were assessed in the experiments of this subsection:

- SCALAPACK SVD: The routine called `pdgesvd` from MKL's ScaLAPACK is used to compute the Singular Value Decomposition (SVD).
- SCALAPACK CPQR: The routine called `pdgeqpf` from MKL's ScaLAPACK is used to compute the column-pivoted QR factorization.

- PLiC CPQR: The routine called `pdgeqp3` from the PLiC library (Parallel Library for Control) [11] is used to compute the column-pivoted QR factorization by using BLAS-3. This source code was linked to the ScaLAPACK library from MKL for the purpose of a fair comparison.

- RANDUTV: A new implementation for computing the randUTV factorization based on the ScaLAPACK infrastructure and library. This source code was linked to the ScaLAPACK library from MKL for the purpose of a fair comparison.

- SCALAPACK QR: The routine called `dgeqrf` from MKL's ScaLAPACK is used to compute the QR factorization. Although this routine does not reveal the rank, it was included in some experiments as a reference for the others.

Like in the previous subsection on shared-memory architectures, for every experiment two plots are shown. The left plot shows the performances when no orthonormal matrices are computed (the codes compute just the singular values for the SVD, the upper triangular matrix R for the CPQR and the QR factorizations, and the upper triangular matrix T for the randUTV factorization). The right plot shows the performances when, in addition to those, all orthonormal matrices are explicitly formed (matrices U and V for SVD and randUTV, and matrix Q for QR and CPQR). Recall that the right plot slightly favors CPQR and QR since only one orthonormal matrix is built.

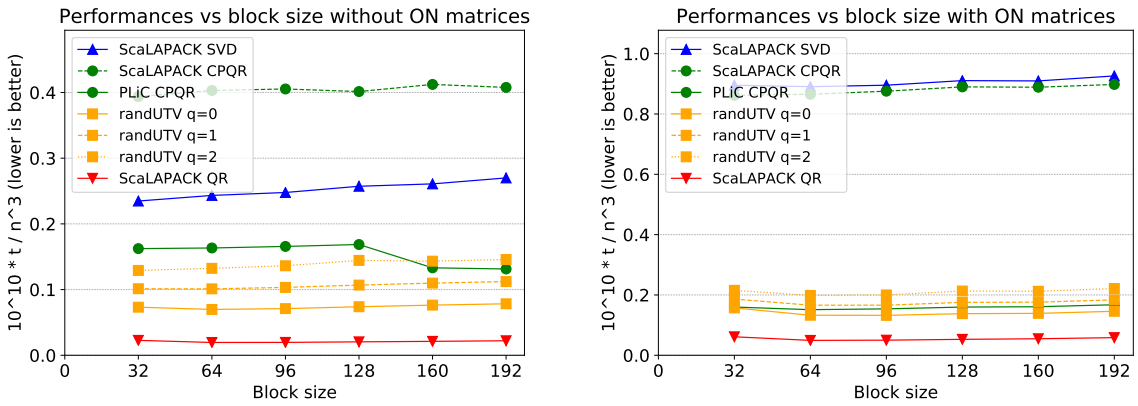


Figure 4.14: Performances versus block size on 96 cores arranged as a 6×16 mesh.

Figure 4.14 shows the performances of all the implementations described above on several block sizes

when using 96 cores arranged as a 6×16 mesh on matrices of dimension 25600×25600. As can be seen, most implementations perform slightly better on small block sizes, such as 32 and 64, the only exception being PLiC CPQR, which performs a bit better on large block sizes when no orthonormal matrices are built.

Figure 4.15 shows the performances of all the implementations for many topologies on matrices of dimension 20480×20480. The top row shows the results on one node (12 cores), the second row shows the results on two nodes (24 cores), the third row shows the results on four nodes (48 cores), and the fourth row shows the results on eight nodes (96 cores). As can be seen, best topologies are usually $p \times q$ with p slightly smaller than q.

Figure 4.16 shows the performances versus matrix dimensions on two different number of cores: 48 cores arranged as 4×12 (top row) and 96 cores arranged as 6×16 (bottom row). On the largest matrix dimension on 48 cores, when no orthonormal matrices are built, randUTV is between 5.4 ($q = 0$) and 2.8 ($q = 2$) times as fast as the SVD, whereas when orthonormal matrices are built, randUTV is between 6.6 ($q = 0$) and 4.3 ($q = 2$) times as fast as the SVD. On the largest matrix dimension on 96 cores, when no orthonormal matrices are built, randUTV is between 3.4 ($q = 0$) and 1.8 ($q = 2$) times as fast as the SVD, whereas when orthonormal matrices are built, randUTV is between 6.7 ($q = 0$) and 4.5 ($q = 2$) times as fast as the SVD. On medium and large matrices, performances of ScaLAPACK CPQR are much lower than those of randUTV, whereas performances of PLiC CPQR are more similar to those of randUTV. Nevertheless, recall that the precision of CPQR is usually much smaller than that of randUTV.

In distributed-memory applications the traditional approach creates one process per core. However, creating fewer processes and then a corresponding number of threads per process can improve performances in some cases. Obviously, the product of the number of processes and the number of threads per process must be equal to the total number of cores. The advantage of this approach is that the creation of fewer processes reduces the communication cost, which is usually the main bottleneck in distributed-memory applications. In the case of linear algebra applications, creating and using several threads per process can be easily achieved by employing shared-memory parallel LAPACK and BLAS libraries. Nevertheless, great care must be taken to ensure a proper pinning of processes to cores, since otherwise performances drop markedly. This was achieved by using the `-genv I_MPI_PIN_DOMAIN socket` flag when executing

the `mpirun`/`mpiexec` command in the machine used in the experiments.

Figure 4.17 shows the scaled timings of the factorizations of matrices of dimension 25600×25600 on 96 cores when using several configurations with different numbers of threads per process. These plots include the results on a complete set of topologies to isolate the effect of the increased number of threads. As usual, the left three plots show performances when no orthonormal matrices are built, whereas the right three plots show performances when orthonormal matrices are built. The top row shows performances when one process per core (96 processes) and then one thread per process are created ($96 \times 1 = 96$). The second row shows performances when one process per two cores (48 processes) and then two threads per process are created ($48 \times 2 = 96$). The third row shows performances when one process per three cores (32 processes) and then three threads per process are created ($32 \times 3 = 96$). As can be seen, the SVD only increases performances when orthonormal matrices are created, whereas randUTV increases performances in both cases (both with and without orthonormal matrices).

	No ON matrices Threads per process			ON matrices Threads per process		
Factorization	1	2	3	1	2	3
SVD	393.9	644.0	645.3	1494.1	1336.4	1318.0
randUTV $q = 0$	117.0	102.7	112.2	214.2	192.3	203.7
randUTV $q = 1$	168.6	142.5	160.1	272.1	232.3	251.4
randUTV $q = 2$	216.7	180.4	207.8	327.5	271.7	298.7

Table 4.1: Best timings in seconds of several topologies with 96 cores on matrices of dimension 25600×25600 considering several number of threads per process.

Table 4.1 shows the best timings (in seconds) for several topologies with 96 cores so a finer detail comparison can be achieved. Matrices being factorized are 25600×25600. As can be seen, SVD increases performances 13 % when orthonormal matrices are built, whereas randUTV with $q = 2$ improves performances 20 % in both cases. Performances usually increase when using two threads per process, but they remain similar or drop when using more than two threads per process.

Figure 4.18 shows the speedups obtained by all the implementations on matrices of dimension 20480×20480. Recall that in this plot every implementation compares against itself on one core. The best topologies

have been selected for the following number of cores: 3×4 for 12 cores, 6×4 for 24 cores, 4×12 for 48 cores, and 6×16 for 96 cores. When no orthonormal matrices are built, speedups of randUTV on the largest number of cores (96) are between 47.1 ($q = 0$) and 43.3 ($q = 2$). When orthonormal matrices are built, speedups of randUTV on the largest number of cores (96) are between 49.3 ($q = 0$) and 44.7 ($q = 2$). In both cases, the efficiency is close to 50 %. When no orthonormal matrices are built, speedups of randUTV are a bit lower than those of QR factorization; when orthonormal matrices are built, speedups of randUTV are a bit higher than those of QR factorization. In both cases, the speedups of randUTV are much higher than those obtained by the SVD and the CPQR factorization, thus showing the great scalability potential of this factorization.

In conclusion, randUTV is significantly faster than the available distributed memory implementations of SVD. It also matches the best CPQR implementation tested. randUTV is known to reveal rank far better than CPQR [89], so just matching the latter in terms of speed equates to a major gain in information at no additional computational cost. Furthermore, randUTV is faster than even CPQR in the case that orthonormal matrices are required. We finally observe that the potential for scalability of randUTV is a clear step above competing implementations for rank-revealing factorizations in distributed memory.

Acknowledgements: *The work of P.G. Martinsson was supported by the Office of Naval Research (grant N00014-18-1-2354) and by the National Science Foundation (grant DMS-1620472).*

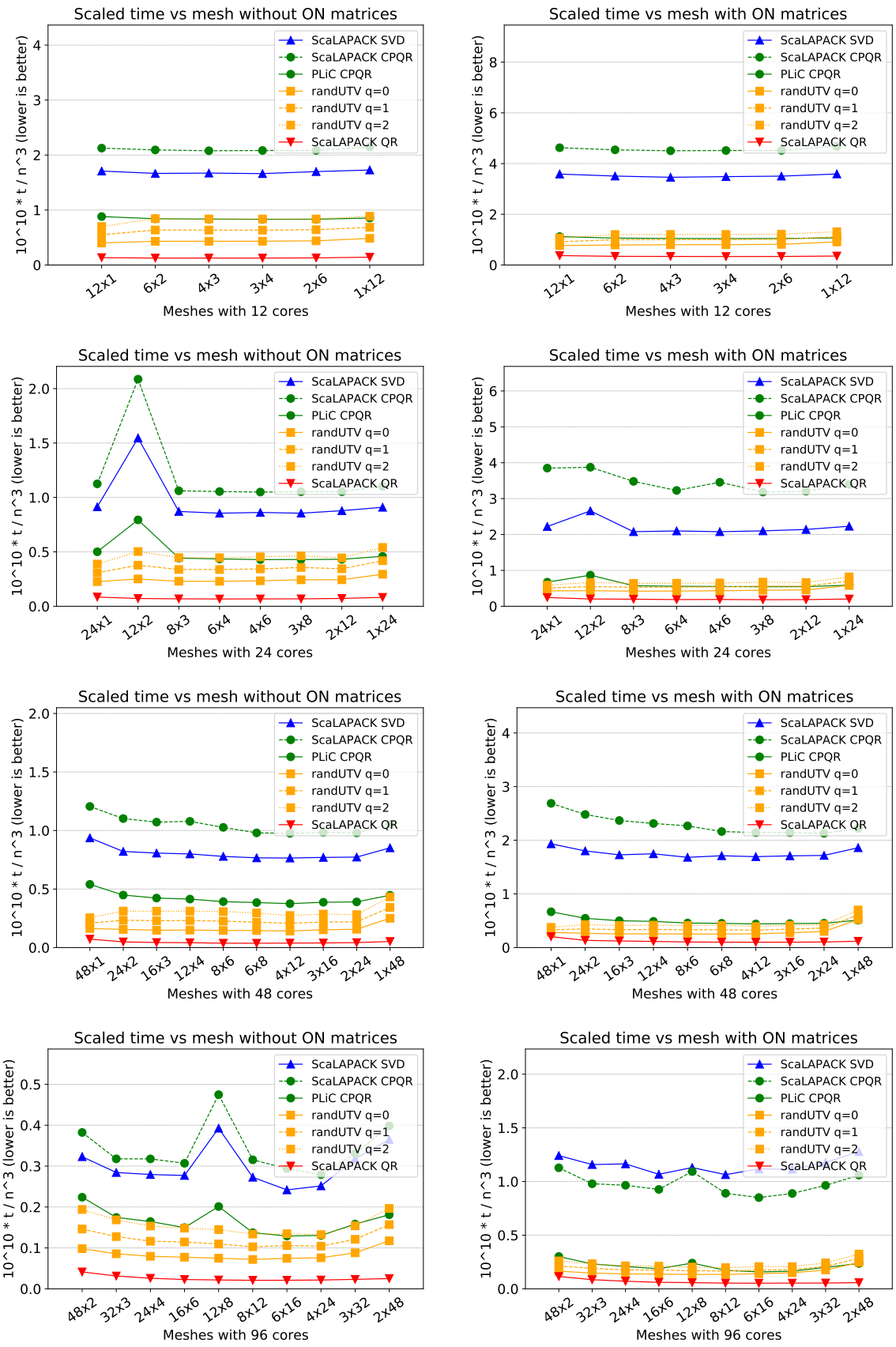


Figure 4.15: Performances on several topologies on matrices of dimension 20480×20480.

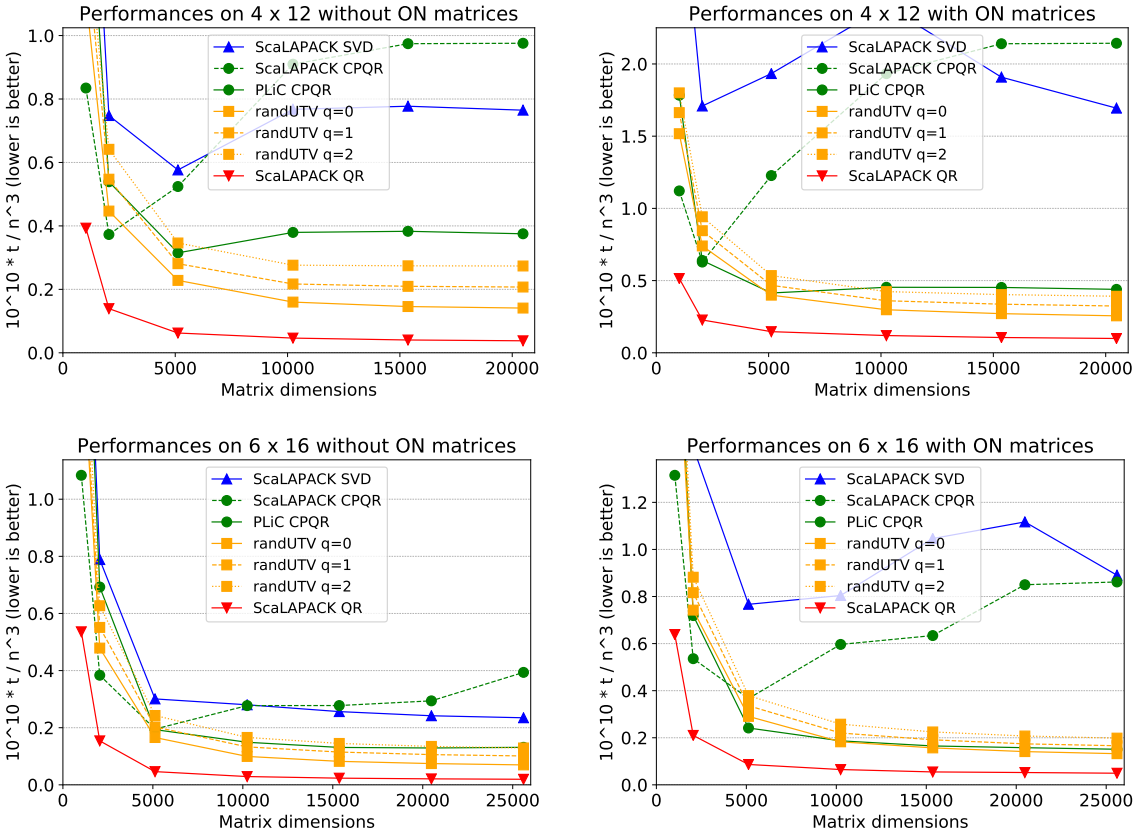


Figure 4.16: Performances versus matrix dimensions on two different number of cores. The top row shows results on 48 cores arranged as 4×12; the bottom row shows results on 96 cores arranged as 6×16.

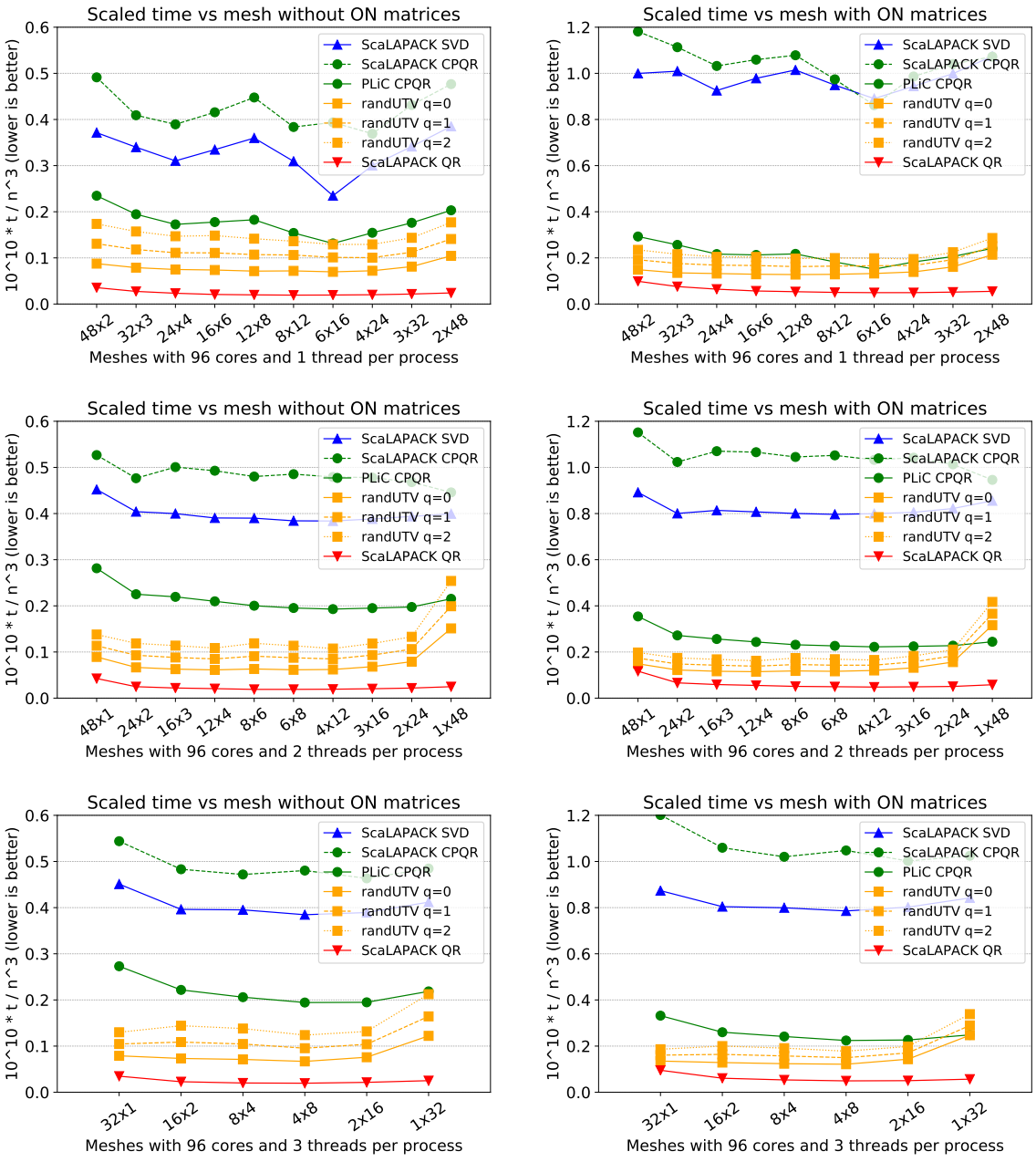


Figure 4.17: Performances on several topologies on matrices of dimension 25600×25600.

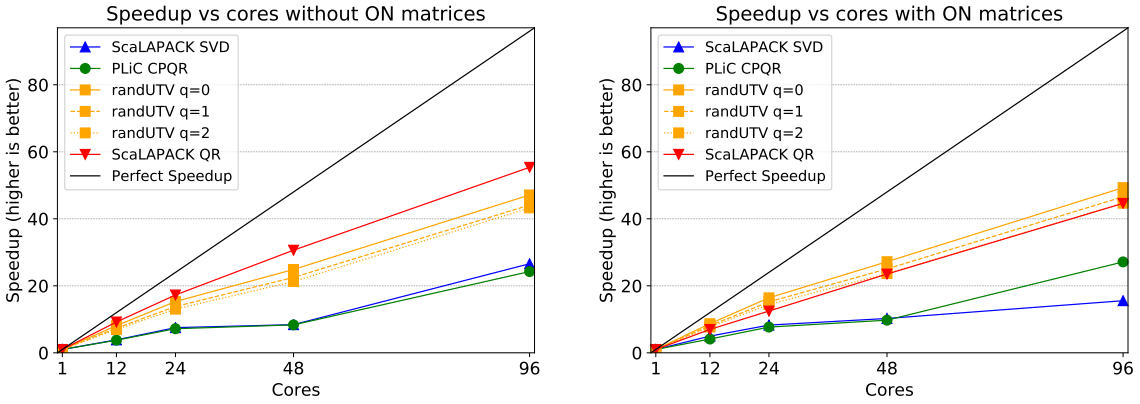


Figure 4.18: Speedups versus number of cores for all the implementations on matrices of dimension 20480×20480.

Chapter 5

Efficient algorithms for computing rank-revealing factorizations on a GPU

Standard rank-revealing factorizations such as the singular value decomposition and column pivoted QR factorization are challenging to implement efficiently on a GPU. A major difficulty in this regard is the inability of standard algorithms to cast most operations in terms of the Level-3 BLAS. This paper presents two alternative algorithms for computing a rank-revealing factorization of the form $\mathbf{A} = \mathbf{UTV}^*$, where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal and $\mathbf{T}$ is upper trapezoidal. Both algorithms use randomized projection techniques to cast most of the flops in terms of matrix-matrix multiplication, which is exceptionally efficient on the GPU. Numerical experiments illustrate that these algorithms achieve an order of magnitude acceleration over finely tuned GPU implementations of the SVD while providing low rank approximation errors close to that of the SVD.

5.1 Introduction

5.1.1 Rank-revealing factorizations

Given an $m \times n$ matrix $\mathbf{A}$, it is often desirable to compute a factorization of $\mathbf{A}$ that uncovers some of its fundamental properties. One such factorization, the *rank-revealing factorization*, is characterized as follows. Consider a matrix factorization

$$\underset{m \times n}{\mathbf{A}} = \underset{m \times m}{\mathbf{U}} \quad \underset{m \times n}{\mathbf{R}} \quad \underset{n \times n}{\mathbf{V}^*}.$$

If $\mathbf{R}$ is partitioned as

$$\mathbf{R} = \begin{bmatrix} \mathbf{R}_{11} & \mathbf{R}_{12} \\ \mathbf{R}_{21} & \mathbf{R}_{22} \end{bmatrix},$$

where $\mathbf{R}_{11}$ is $k \times k$ for some $1 \leq k \leq \min(m, n)$, we will call the factorization *rank-revealing* if it satisfies the following conditions:

1. $\sigma_{\min}(\mathbf{R}_{11}) \approx \sigma_k(\mathbf{A})$.

2. $\sigma_{\max}(\mathbf{R}_{22}) \approx \sigma_{k+1}(\mathbf{A})$.

This informal definition is a slight generalization of the usual definitions of rank revealing decompositions that appear in the literature, minor variations of which appear in, *e.g.* [27, 114, 24, 26]. Rank revealing factorizations are useful in solving problems such as least squares approximation [26, 25, 54, 80, 17, 56], rank estimation [24, 118, 116], low rank approximation [83, 42, 77, 31, 9], and subspace tracking [16, 114], among others.

Perhaps the two most commonly known and used rank revealing factorizations are the column pivoted QR decomposition (CPQR) and the singular value decomposition (SVD).[1] The SVD provides the theoretically optimal rank revealing decomposition, in that for an SVD $\mathbf{A} = \mathbf{U}\mathbf{D}\mathbf{V}^*$, we have $\sigma_{\min}(\mathbf{D}(1 : k, 1 : k)) = \sigma_i(\mathbf{A})$ and $\sigma_{\max}(\mathbf{D}(k+1 : m, k+1 : n)) = \sigma_{k+1}(\mathbf{A})$ for all $1 \leq k \leq \min(m, n)$. The SVD has high computational cost, however, and for large problems or iterative algorithms requiring such a computation during each step, this cost is often prohibitive. In this case, one can turn to the CPQR, which is faster to compute than the SVD but does not reveal rank as well.

A third choice is the rank revealing UTV factorization (RRUTV) [114, 119, 89, 49, 6]. An RRUTV can be thought of as a compromise between the SVD and CPQR that is better at revealing the numerical rank than the CPQR, and faster to compute than the SVD. Traditional algorithms for computing an RRUTV have been deterministic and guarantee revealing the rank of a matrix up to a user-defined tolerance. It is not used as widely as the aforementioned SVD and CPQR, though, except in a few settings such as subspace tracking.

[1] See Sections 5.2.3 and 5.2.2, respectively, for a brief overview of these factorizations.

5.1.2 Challenges of implementing the SVD and the CPQR on a GPU

As focus in high performance computing has shifted towards parallel environments, the use of GPUs to perform scientific computations has gained popularity and success [100, 84, 20]. The power of the GPU lies in its ability to execute many tasks in parallel extremely efficiently, and software tools have rapidly developed to allow developers to make full use of its capabilities. Algorithm design, however, is just as important. Classical algorithms for computing both the SVD and CPQR, still in use today, were designed with a heavier emphasis on reducing the number of floating point operations (flops) than on running efficiently on parallel systems. Thus, it is difficult for either factorization to maximally leverage the computing power of a GPU.

For CPQR, the limitations of the parallelism are well understood, at least relative to comparable matrix computations. The most popular algorithm for computing a CPQR uses Householder transformations and chooses the pivot columns by selecting the column with the largest norm. We will refer to this algorithm as HQRCP. See Section 5.2.3 for a brief overview of HQRCP, or, *e.g.* [21, 54] for a thorough description. Given a sequence of matrix operations, it is well known that an appropriate implementation using Level-3 BLAS, or matrix-matrix, operations will run more efficiently on modern processors than an optimal implementation using Level-2 or Level-1 BLAS [19]. This is largely due to the greater potential for the Level-3 BLAS to make more efficient use of memory caching in the processor. HQRCP as written originally in [54] uses no higher than Level-2 BLAS. Quintana-Ortí *et al.* developed HQRCP further in [107], casting about half of the flops in terms of Level-3 BLAS kernels. Additional improvement in this area, though, is difficult to find for this algorithm, as the process of selecting pivot columns inherently prevents full parallelization.

The situation for the SVD is even more bleak. It is usually computed in two stages. The first is a reduction to bidiagonal form via, *e.g.* Householder reflectors. Only about half the flops in this computation can be cast in terms of the Level-3 BLAS, similarly (and for similar reasons) to `HQRCP`. The second stage is the computation of the SVD of the bidiagonal matrix. This is usually done with either an iterative algorithm (a variant of the QR algorithm) or a recursive algorithm (divide-and-conquer) which reverts to the QR algorithm at the base layer. See [122, 57, 33, 63] for details. The recursive option inherently resists

parallelization, and the current widely-used implementations of the QR approach are cast in terms of an operation that behaves like a Level-2 BLAS.[2]

5.1.3 Proposed algorithms

In this paper, we present two randomized algorithms for computing an RRUTV. Both algorithms are designed to run efficiently on GPUs in that the majority of their flops are cast in terms of matrix-matrix multiplication. We show through extensive numerical experiments in Section 5.5 that each reveals rank nearly as well as the SVD but often costs less than HQRCP to compute on a GPU. For matrices with uncertain or high rank, then, these algorithms warrant strong consideration for this computing environment.

The first algorithm POWERURV, discussed in Section 5.3, was first introduced in the tech report [58]. POWERURV is built on another randomized RRUTV algorithm developed by Demmel *et al.* in [35], adding better rank revelation at a tolerable increase in computational cost. The algorithm itself is quite simple, capable of description with just a few lines of code. The simplicity of its implementation is a significant asset to developers, and it has just one input parameter, whose effect on the resulting calculation can easily be understood.

The second algorithm, RANDUTV, was first presented in [89]. RANDUTV is a *blocked* algorithm, meaning it operates largely inside a loop, "processing" multiple columns of the input matrix during each iteration. Specifically, for an input matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ with $m \geq n$,[3] a block size b is chosen, and the bulk of RANDUTV's work occurs in a loop of $p = \lceil n/b \rceil$ steps. During step i, orthogonal matrices $\mathbf{U}^{(i)}$ and $\mathbf{V}^{(i)}$ are computed which approximate singular vector subspaces of a relevant block of $\mathbf{T}^{(i-1)}$. Then, $\mathbf{T}^{(i)}$ is formed with

$$\mathbf{T}^{(i)} := (\mathbf{U}^{(i)})^* \mathbf{T}^{(i-1)} \mathbf{V}^{(i)}.$$

The leading ib columns of $\mathbf{T}^{(i)}$ are upper triangular (see Figure 5.1 for an illustration of the sparsity pattern), so we say that RANDUTV drives $\mathbf{A}$ to upper triangular form b columns at a time. After the final step in the

[2] The QR algorithm can be cast in terms of Level-3 BLAS, but for reasons not discussed here, this approach has not yet been adopted in most software. See [127] for details.

[3] if $m < n$, we may simply factorize the transpose and take the transpose of the result.

loop, we obtain the final $\mathbf{T}$, $\mathbf{U}$, and $\mathbf{V}$ factors with

$$\mathbf{T} := \mathbf{T}^{(p)},$$
$$\mathbf{U} := \mathbf{U}^{(1)}\mathbf{U}^{(2)}\cdots\mathbf{U}^{(p)},$$
$$\mathbf{V} := \mathbf{V}^{(1)}\mathbf{V}^{(2)}\cdots\mathbf{V}^{(p)}.$$

See Section 5.4 for the full algorithm. A major strength of RANDUTV is that it may be adaptively stopped at any point in the computation, for instance when the singular value estimates on the diagonal of the $\mathbf{T}^{(i)}$ matrices drop below a certain threshold. If stopped early, the algorithm incurs only a cost of $\mathcal{O}(mnk)$ for an $m \times n$ input matrix. Each matrix $\mathbf{U}^{(i)}$ and $\mathbf{V}^{(i)}$ is computed using techniques similar to that of the randomized SVD [69], which spends most of its flops in matrix multiplication and therefore makes efficient use of GPU capabilities.

In this paper, we propose several modifications to the RANDUTV algorithm given in [89]. These changes are made with the titular computing environment in mind, knowing that matrix-matrix multiplication is far more efficient on a GPU than unpivoted QR, RANDUTV's other building block. The revised algorithm enhances the rank revealing properties of the resulting RRUTV factorization while keeping the computational cost relatively low.

Remark 14 (RAM limitations)**.** *In this manuscript, we restrict attention to the case where all the data used in the computation fits in RAM on the GPU, which somewhat limits the size of matrices that can be handled. (For instance, in the numerical experiments reported in Section 5.5, the largest problem size we could handle involved matrices of size about* $15\,000 \times 15\,000$*.) The techniques can be modified to allow larger matrices to be handled and for multiple GPUs to be deployed, but we leave this extension for future work.*

5.1.4 Outline of paper

In Section 5.2, we review previous work in rank-revealing factorizations, discussing competing methods as well as previous work in randomized linear algebra that is foundational for the methods presented in this article. Section 5.3 presents the first algorithmic contribution of this article, POWERURV. In Section 5.4, we discuss and build on the recently developed RANDUTV algorithm, culminating in a modification

of the algorithm RANDUTV_BOOSTED with greater potential for low rank estimation. Finally, Section 5.5 presents numerical experiments which demonstrate the computational efficiency of POWERURV and RANDUTV_BOOSTED as well as their effectiveness in low rank estimation.

5.2 Preliminaries

5.2.1 Basic notation

In this manuscript, we write $\mathbf{A} \in \mathbb{R}^{m \times n}$ to denote a real-valued matrix with m rows and n columns, and $\mathbf{A}(i, j)$ refers to the element in the i-th row and j-th column of $\mathbf{A}$. The indexing notation $\mathbf{A}(i : j, k : l)$ is used to reference the submatrix of $\mathbf{A}$ consisting of the entries in the i-th through j-th rows of the k-th through l-th columns. $\sigma_i(\mathbf{A})$ is the i-th singular value of $\mathbf{A}$, and $\mathbf{A}^*$ is the transpose. An orthonormal matrix is a matrix whose columns have unit norm and are pairwise orthogonal, and an orthogonal matrix is a square orthonormal matrix. The default norm $\|\cdot\|$ is the spectral norm.

5.2.2 The Singular Value Decomposition (SVD)

Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $p = \min(m, n)$. Then the singular value decomposition (SVD) of $\mathbf{A}$ [57, 122, 116] is given by

$$\begin{array}{ccccc} \mathbf{A} & = & \mathbf{U} & \mathbf{\Sigma} & \mathbf{V}^*, \\ m \times n & & m \times m & m \times n & n \times n \end{array}$$

where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal and $\mathbf{\Sigma}$ is diagonal. When $m \neq n$, we may also speak of the *economic* SVD of $\mathbf{A}$, given by

$$\begin{array}{ccccc} \mathbf{A} & = & \mathbf{U} & \mathbf{\Sigma} & \mathbf{V}^*, \\ m \times n & & m \times p & p \times p & p \times n \end{array}$$

in which case either $\mathbf{U}$ or $\mathbf{V}$ is orthonormal instead of orthogonal. The diagonal elements $\{\sigma_i\}_{i=1}^{p}$ of $\mathbf{\Sigma}$ are the *singular values* of $\mathbf{A}$ and satisfy $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_p \geq 0$. The columns $\mathbf{u}_i$ and $\mathbf{v}_i$ of $\mathbf{U}$ and $\mathbf{V}$ are called the *left* and *right singular vectors*, respectively, of $\mathbf{A}$. Importantly, the SVD provides theoretically optimal rank-k approximations to $\mathbf{A}$. Specifically, the Eckart-Young-Mirsky Theorem [45, 98] states that given the

SVD of a matrix $\mathbf{A}$ and a fixed $k \in \{1, 2, \ldots, p\}$, we have that

$$\|\mathbf{A} - \mathbf{U}(:, 1:k)\mathbf{\Sigma}(1:k, 1:k)(\mathbf{V}(:, 1:k))^*\| = \inf\{\|\mathbf{A} - \mathbf{B}\| : \mathbf{B} \text{ has rank } k\}.$$

5.2.3 The Column Pivoted QR (CPQR) decomposition

Given a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, a column pivoted QR decomposition (CPQR) is given by

$$\begin{array}{ccccc} \mathbf{A} & = & \mathbf{Q} & \mathbf{R} & \mathbf{P}^*, \\ m \times n & & m \times m & m \times n & n \times n \end{array}$$

where $\mathbf{Q}$ is orthogonal, $\mathbf{R}$ is upper triangular, and $\mathbf{P}$ is a permutation matrix. If $m > n$, then any CPQR can be reduced to the "economic" CPQR

$$\begin{array}{ccccc} \mathbf{A} & = & \mathbf{Q} & \mathbf{R} & \mathbf{P}^*. \\ m \times n & & m \times n & n \times n & n \times n \end{array}$$

The details of the most popular algorithm for computing such a factorization, called HQRCP hereafter, are not essential to this article, but they may be explored by the reader in, *e.g.* [24, 57, 122, 116].

The rank-k approximations $\mathbf{Q}(:, 1:k)\mathbf{R}(1:k, 1:k)(\mathbf{P}(:, 1:k))^*$ provided by HQRCP are usually reasonable, though in some cases they can be alarmingly suboptimal [76]. These cases are rare, particularly in practice, and HQRCP is so much faster than computing an SVD that HQRCP is used ubiquitously for low rank approximation.

The special case when $\mathbf{P} = \mathbf{I}$ is called a(n) (unpivoted) QR decomposition. The standard algorithm for computing a QR factorization relies on Householder reflectors. We shall call this algorithm as HQR in this article and refer the reader once again to textbooks such as [57, 122, 116] for a complete discussion. For this article, it is only necessary to note that the outputs of HQR are the matrix $\mathbf{R}$ of the QR factorization and a unit lower triangular matrix $\mathbf{V} \in \mathbb{R}^{m \times p}$ that can be used to build $\mathbf{Q}$. QR decompositions have no rank revealing properties, but in this article we make critical use of the fact that the leading p columns of $\mathbf{Q}$ form an orthonormal basis for the column space of $\mathbf{A}$. The lack of the pivoting matrix $\mathbf{P}$ also allows the standard algorithm for computing a QR decomposition to rely more heavily on the level-3 BLAS than HQRCP, translating to better performance in parallel environments.

5.2.4 Efficiently applying products of Householder vectors to a matrix

Consider a matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ and a block size b such that $1 \leq b \leq n$. Suppose that we have determined b Householder matrices $\mathbf{H}$ such that

$$\mathbf{H}_b^* \mathbf{H}_{b-1}^* \cdots \mathbf{H}_1^* \mathbf{A}(:, 1:b)$$

is upper trapezoidal. We have that

$$\mathbf{H}_i = \mathbf{I} - 2\mathbf{v}_i \mathbf{v}_i^*, \ 1 \leq i \leq b,$$

where $\mathbf{v}_i$ is the Householder vector associated with the transformation. Then the matrix $\mathbf{H} = \mathbf{H}_1 \mathbf{H}_2 \cdots \mathbf{H}_b$ can be represented as

$$\mathbf{H} = \mathbf{I} - \mathbf{V}\mathbf{T}\mathbf{V}^*,$$

where $\mathbf{T} \in \mathbb{R}^{b \times b}$ is upper triangular and $\mathbf{V} \in \mathbb{R}^{n \times b}$ is lower trapezoidal with columns containing the $\mathbf{v}_i$ [113]. The form $\mathbf{I} - \mathbf{V}\mathbf{T}\mathbf{V}^*$ of $\mathbf{H}$ is called the *compact-WY* representation of a product of Householder reflectors. This representation reduces the storage requirement of $\mathbf{H}$ from $\mathcal{O}(n^2)$ to $\mathcal{O}(nb + b^2)$. More importantly, when applying the matrices to $\mathbf{A}$, the compact-WY form recasts the matrix-vector operations at the core of

$$\mathbf{H}_b \mathbf{H}_{b-1} \cdots \mathbf{H}_1 \mathbf{A} = (\mathbf{I} - 2\mathbf{v}_b \mathbf{v}_b^*)(\mathbf{I} - 2\mathbf{v}_{b-1} \mathbf{v}_{b-1}^*) \cdots (\mathbf{I} - 2\mathbf{v}_1 \mathbf{v}_1^*)\mathbf{A}$$

into matrix-matrix multiplications as in

$$\mathbf{H}\mathbf{A} = (\mathbf{I} - \mathbf{V}\mathbf{T}\mathbf{V}^*)\mathbf{A}.$$

Taking full advantage of this representation is crucial for efficiently building factorizations on a GPU.

5.2.5 The UTV decomposition

Consider a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$. A factorization

$$\begin{array}{ccccc} \mathbf{A} & = & \mathbf{U} & \mathbf{T} & \mathbf{V}^*. \\ m \times n & & m \times m & m \times n & n \times n \end{array}$$

where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal and $\mathbf{T}$ is upper trapezoidal is called a *UTV factorization*. Note that the SVD and CPQR are technically both examples of UTV factorizations. In the literature, however, a decomposition is generally given the UTV designation only if that is its most specific label; we will continue this practice in the current article. Thus, it is implied that $\mathbf{T}$ is upper trapezoidal but not diagonal, and $\mathbf{V}$ is orthogonal but not a permutation.

The flexibility of the factors of a UTV decomposition allow it to act as a compromise between the SVD and CPQR in that it is not too expensive to compute but can reveal rank quite well. A rank-revealing UTV can also be updated and downdated easily; see [116, Ch. 5, Sec. 4] and [50, 102, 114] for details.

5.2.6 The randomized singular value decomposition

The randomized SVD (RSVD) [69] provides an approximate SVD of a matrix with low rank. Specifically, given a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ with rank $k \ll \min(m,n)$, the RSVD produces orthonormal matrices $\mathbf{U} \in \mathbb{R}^{m\times k}$ and $\mathbf{V} \in \mathbb{R}^{n\times k}$ and diagonal matrix $\mathbf{D} \in \mathbb{R}^{k\times k}$ such that

$$\mathbf{A} \approx \mathbf{U}\mathbf{D}\mathbf{V}^*.$$

The key to the algorithm is efficiently computing an orthonormal basis $\mathbf{Q} \in \mathbb{R}^{m\times k}$ for the column space of $\mathbf{A}$. This can be accomplished using randomized projections. An extremely simple way of building $\mathbf{Q}$ is the following:

1. Draw a matrix $\mathbf{G} \in \mathbb{R}^{n\times k}$ with i.i.d. entries drawn from the standard normal distribution.
2. Compute $\mathbf{Y} = \mathbf{A}\mathbf{G}$.
3. Calculate an orthonormal basis $\mathbf{Q}$ for Col($\mathbf{Y}$) via, *e.g.* HQR.

This method will yield a reasonably good approximation to the column space of $\mathbf{A}$. However, for certain matrices, particularly when the decay in singular values is slow, an improved approximation will be desired. We may improve the approximation provided by $\mathbf{Q}$ in two ways:

(i) **Oversampling:** We may interpret $\mathbf{Y}$ as k different random projections onto the column space of $\mathbf{A}$. As shown in [69], the approximation of Col($\mathbf{Y}$) to Col($\mathbf{A}$) may be improved by gathering a few,

say p, extra projections. In practice $p = 5$ or $p = 10$ is sufficient, adding very little additional computational cost to the algorithm. Using a small amount of oversampling also improves the expected error bounds and vastly decreases the probability of deviating from those bounds. Thus, this technique makes the uses of randomization in the algorithm safe and reliable.

(ii) **Power iteration:** We may replace the sampling matrix $\mathbf{A}$ with $(\mathbf{A}\mathbf{A}^*)^q\mathbf{A}$ for $q \in \mathbb{N}$, a matrix with the same column space as $\mathbf{A}$ but whose singular values are $(\sigma_i(\mathbf{A}))^{2q+1}$. The contributions of singular vectors that correspond to smaller singular values will be diminished using this new sampling matrix, thus increasing the alignment of Col($\mathbf{Y}$) with the optimal approximation of Col($\mathbf{A}$). In practice, choosing q to be 0, 1 or 2 gives excellent results. When using a power iteration scheme, numerical linear dependence of the samples becomes a concern since the information will be lost from singular vectors corresponding to singular values less than $\epsilon_{\text{machine}}^{2q+1}$. To stabilize the algorithm, we build $\mathbf{Y}$ incrementally, orthonormalizing the columns of the intermediate matrix in between applications of $\mathbf{A}$ and $\mathbf{A}^*$.

These ideas are combined to produce the RANGE_FINDER method of Algorithm 3.

Algorithm 3 $\mathbf{Q} =$ RANGE_FINDER($\mathbf{A}, k, p$)

```
G =GENERATE_NORM_RAND_IID(n(A), k + p)
Y = AG
[Q, ∼] = HQR(Y)
for i = 1: q do
    Y = A*Q
    [Q, ∼] = HQR(Y)
    Y = AQ
    [Q, ∼] = HQR(Y)
end for
```

Once $\mathbf{Q}$ is built, the rest of the RSVD algorithm uses only the standard techniques of matrix multiplication, unpivoted QR factorization, and an SVD on a matrix of small dimensions. The full RSVD algorithm is given in Algorithm 4.

Observe that once $\mathbf{Q}$ is found, we have that $\mathbf{A} \approx \mathbf{Q}\mathbf{Q}^*\mathbf{A}$. Therefore

$$\mathbf{A} \approx \mathbf{Q}\mathbf{Q}^*\mathbf{A} = (\mathbf{Q}\hat{\mathbf{U}})\mathbf{D}\mathbf{V}^* = \mathbf{U}\mathbf{D}\mathbf{V}^*.$$

Algorithm 4 $[\mathbf{U}, \mathbf{D}, \mathbf{V}] = \text{RSVD}(\mathbf{A}, k)$

1: $\mathbf{Q}$ =RANGE_FINDER($\mathbf{A}, k$)
2: $\mathbf{B} = \mathbf{Q}^*\mathbf{A}$
3: $[\hat{\mathbf{U}}, \mathbf{D}, \mathbf{V}] = \text{SVD}(\mathbf{B})$
4: $\mathbf{U} = \mathbf{Q}\hat{\mathbf{U}}$

Also note that though the SVD step of $\mathbf{B}$ may look expensive, it can be done quite inexpensively. We may first perform an unpivoted QR factorization ($\mathcal{O}(nk^2)$) followed by an SVD for a $k \times k$ matrix ($\mathcal{O}(k^3)$) and a matrix multiplication to update the right singular vectors ($\mathcal{O}(nk^2)$).

5.3 The POWERURV algorithm.

The POWERURV algorithm is a slight variation of a randomized algorithm proposed by Demmel *et al.* [35] for computing a rank-revealing URV factorization of a given matrix. The modification we propose makes the method far more accurate in revealing the numerical rank, at only a marginal increase in computational time in a GPU environment. POWERURV is remarkably simple to implement. In Section 5.3.1, we review previously published work from Demmel *et al.* which forms the conceptual foundation of POWERURV. Section 5.3 lays out the basic version of the POWERURV algorithm, and Section 5.3.3 discusses an adjustment to the algorithm in Section 5.3 necessary for maintaining the rank-revealing properties of the factorization.

5.3.1 A randomized algorithm for computing a UTV factorization proposed by Demmel, Dumitriu, and Holtz

In [35], Demmel *et al.* give a randomized algorithm RURV (randomized URV) for computing a rank revealing factorization of a matrix. The algorithm can be written quite simply as follows:

1. Generate a matrix $\mathbf{B} \in \mathbb{R}^{n \times n}$ whose entries are independent, identically distributed standard Gaussian variables, *i.e.* $b_{ij} \sim \mathcal{N}(0, 1)$.

2. Compute the unpivoted QR factorization of $\mathbf{B}$ to obtain $\mathbf{V}, \mathbf{R}_B$ such that $\mathbf{B} = \mathbf{V}\mathbf{R}_B$.

3. Compute $\hat{\mathbf{A}} = \mathbf{A}\mathbf{V}^*$.

4. Perform an unpivoted QR factorization on $\hat{\mathbf{A}}$ to obtain $\mathbf{U}, \mathbf{R}$ such that $\hat{\mathbf{A}} = \mathbf{UR}$.

Note that after step (4), we have

$$\mathbf{AV}^* = \mathbf{UR} \qquad \Rightarrow \qquad \mathbf{A} = \mathbf{URV},$$

a URV decomposition of $\mathbf{A}$. With high probability, this factorization satisfies the rank-revealing properties

(a) $\sigma_{\min}(\mathbf{R}(1:r, 1:r)) \approx \sigma_r(\mathbf{A})$, and

(b) if $\sigma_{r+1}(\mathbf{A}) \ll \sigma_r(\mathbf{A})$ and $\mathbf{R}(1:r, 1:r)$ is not too ill-conditioned, then $\sigma_{\max}(\mathbf{R}(r+1:n, r+1:n)) \approx \sigma_{r+1}(\mathbf{A})$.

A key advantage of this algorithm is its simplicity. Since it relies only on unpivoted QR and matrix multiplication computations, it can easily be shown to be stable (see [35] for the proof). Furthermore, both of these operations are relatively well-suited for parallel computing environments like GPUs. Since they are extremely common building blocks for problems involving matrices, highly optimized implementations for both are readily available. Thus, a highly effective implementation of RURV may be quickly assembled by a non-expert.

Demmel *et al.* find a use for RURV as a fundamental component of a fast, stable solution to the eigenvalue problem. For this application, RURV is used iteratively, so not much is required of the accuracy of the rank-revealing aspect of the resulting factorization. For other problems types such as low rank approximation, though, the output of RURV is markedly suboptimal. This is because the matrix $\mathbf{V}$, used as a sort of "pre-conditioner," does not incorporate any information from the row of space of the input $\mathbf{A}$. This observation leads to the key idea of the powerURV algorithm, discussed in Section 5.3.

5.3.2 powerURV: A randomized algorithm enhanced by power iterations

The powerURV algorithm is inspired by the RURV of Section 5.3.1 combined with the observation that the optimal matrix $\mathbf{V}$ to use for rank revealing purposes is the matrix whose columns are the right singular vectors of the input matrix $\mathbf{A}$. If such a $\mathbf{V}$ were used, then the columns of $\hat{\mathbf{A}} = \mathbf{AV}$ would be the left singular vectors of $\mathbf{A}$ scaled by its singular values. Thus finding the right singular vectors of $\mathbf{A}$ yields

theoretically optimal low rank approximations. This subproblem is also as difficult as computing the SVD of $\mathbf{A}$, though, so we settle for choosing $\mathbf{V}$ as an efficiently computed approximation to the right singular vectors of $\mathbf{A}$.

A technique for efficiently computing a good approximation to the row space of $\mathbf{A}$ is used in the previously discussed randomized SVD (see Section 5.2.6). Computation of $\mathbf{V}$ consists of three steps:

1. Generate a matrix $\mathbf{G} \in \mathbb{R}^{n \times n}$ whose entries are independent, identically distributed standard Gaussian variables, *i.e.* $g_{ij} \sim \mathcal{N}(0, 1)$.

2. Sample the row space of $\mathbf{A}$ by calculating $\mathbf{Y} = (\mathbf{A}^*\mathbf{A})^q\mathbf{G}$, where q is some small non-negative integer.

3. Compute an orthonormal basis for the column space of $\mathbf{Y}$ with an unpivoted QR decomposition, obtaining an orthogonal matrix $\mathbf{V}$ and upper triangular matrix $\mathbf{S}$ such that $\mathbf{Y} = \mathbf{VS}$.

The matrix $\mathbf{Y}$ can be thought of as a random projection onto the row space of $\mathbf{A}$. Specifically, the columns of $\mathbf{Y}$ are random linear combinations of the columns of $(\mathbf{A}^*\mathbf{A})^q\mathbf{A}^*$. To examine the effect of q on the accuracy of the approximation provided by $\mathbf{V}$, let the SVD of $\mathbf{A}$ be given by $\mathbf{A} = \mathbf{QDX}^*$. Then we have

$$(\mathbf{A}^*\mathbf{A})^q = \mathbf{XD}^{2q}\mathbf{X}^*.$$

Therefore, the larger the value of q, the faster the singular value decay of the sampled matrix, and the lesser the contribution of smaller, noisier singular values to the columns of $\mathbf{Y}$. Therefore, choosing q to be large increases the quality of the low rank approximations at the cost of increasing the number of matrix multiplications required. In practice, choosing $q = 0, 1,$ or 2 produces high quality approximations while maintaining good computational efficiency.

The complete instruction set for powerURV is given in Algorithm 5.

5.3.3 Maintaining orthonormality numerically

Unfortunately, Algorithm 5 is vulnerable to accumulation of roundoff errors. In particular, the columns of $\mathbf{Y}$ will lose the information provided by singular vectors with corresponding singular value

Algorithm 5 [**U**,**R**,**V**] = POWERURV($\mathbf{A}, n$)

1: $\mathbf{G}$ =GENERATE_NORM_RAND_IID(n, n)
2: $\mathbf{Y} = (\mathbf{A}^*\mathbf{A})^q\mathbf{A}^*\mathbf{G}$
3: $[\mathbf{V}, \mathbf{S}] = \text{HQR}(\mathbf{Y})$
4: $\hat{\mathbf{A}} = \mathbf{AV}$
5: $[\mathbf{U}, \mathbf{R}] = \text{HQR}(\hat{\mathbf{A}})$

smaller than $\epsilon_{\text{mach}}^{1/(2q+1)}$. This problem can be remedied by orthonormalizing the columns of the sampling matrix in between each application of $\mathbf{A}$ or $\mathbf{A}^*$. The result, Algorithm 6, yields precisely the same result as Algorithm 5 when executed in exact arithmetic. Algorithm 6 is also more computationally expensive, requiring $2q + 1$ more unpivoted QR factorizations. In the absence of prior knowledge of the spectrum of $\mathbf{A}$, however, the orthonormalization is required to reliably obtain a high quality rank revealing factorization.

Algorithm 6 [**U**,**R**,**V**] = POWERURV_STABLE($\mathbf{A}, n$)

1: $\mathbf{G}$ =GENERATE_NORM_RAND_IID(n, n)
2: $\mathbf{Y} = \mathbf{A}^*\mathbf{G}$
3: $[\mathbf{V}, \mathbf{S}] = \text{HQR}(\mathbf{Y})$
4: **for** $i = 1$: q **do**
5: $\quad\mathbf{Y} = \mathbf{AV}$
6: $\quad[\mathbf{V}, \mathbf{S}] = \text{HQR}(\mathbf{Y})$
7: $\quad\mathbf{Y} = \mathbf{A}^*\mathbf{V}$
8: $\quad[\mathbf{V}, \mathbf{S}] = \text{HQR}(\mathbf{Y})$
9: **end for**
10: $\hat{\mathbf{A}} = \mathbf{AV}$
11: $[\mathbf{U}, \mathbf{R}] = \text{HQR}(\mathbf{Y})$

5.3.4 Relationship with RSVD

The POWERURV algorithm is closely connected with the standard randomized singular value decomposition algorithm (RSVD). To describe the connection, let us briefly review the steps in the RSVD. Let $\mathbf{A} \in \mathbb{R}^{m\times n}$ with $m \geq n$. Given an integer $\ell < n$, the RSVD builds an approximation to a truncated SVD via the following steps: a random matrix (typically Gaussian) $\mathbf{G}_{\text{rsvd}} \in \mathbb{R}^{n\times\ell}$ is drawn and the product

$$\mathbf{Y}_{\text{rsvd}} = (\mathbf{AA}^*)^q\mathbf{AG}_{\text{rsvd}} \in \mathbb{R}^{m\times\ell}, \tag{5.1}$$

is evaluated. The columns $\mathbf{Y}_{\text{rsvd}}$ are orthogonalized via an unpivoted QR factorization

$$\mathbf{Y}_{\text{rsvd}} = \mathbf{Q}_{\text{rsvd}}\mathbf{R}_{\text{rsvd}}. \tag{5.2}$$

In other words, the columns of the $m \times \ell$ matrix $\mathbf{Q}_{\text{rsvd}}$ form an orthogonal basis for the range of $\mathbf{Y}_{\text{rsvd}}$. Next, a deterministic SVD of $\mathbf{Q}_{\text{rsvd}}^{*}\mathbf{A}$ is computed to obtain

$$\mathbf{Q}_{\text{rsvd}}^{*}\mathbf{A} = \mathbf{W}_{\text{rsvd}}\mathbf{\Sigma}_{\text{rsvd}}(\mathbf{V}_{\text{rsvd}})^{*}, \tag{5.3}$$

where $\mathbf{W}_{\text{rsvd}} \in \mathbb{R}^{\ell \times \ell}$ is orthogonal, where $\mathbf{V}_{\text{rsvd}} \in \mathbb{R}^{n \times \ell}$ is orthonormal, and $\mathbf{\Sigma}_{\text{rsvd}} \in \mathbb{R}^{\ell \times \ell}$ is diagonal with non-negative entries. The final step is to define the $m \times \ell$ matrix

$$\mathbf{U}_{\text{rsvd}} = \mathbf{Q}_{\text{rsvd}}\mathbf{W}_{\text{rsvd}}. \tag{5.4}$$

The end result is an approximate singular value decomposition

$$\mathbf{A} \approx \mathbf{U}_{\text{rsvd}}\mathbf{\Sigma}_{\text{rsvd}}\mathbf{V}_{\text{rsvd}}^{*}.$$

To compare this process with POWERURV, we may summarize POWERURV with two steps. First, the QR factorization of $(\mathbf{A}\mathbf{A}^{*})^{q}\mathbf{G}$:

$$(\mathbf{A}\mathbf{A}^{*})^{q}\mathbf{G} = \mathbf{V}\mathbf{Z}. \tag{5.5}$$

Second, the QR factorization of $\mathbf{A}\mathbf{V}$:

$$\mathbf{A}\mathbf{V} = \mathbf{U}\mathbf{R}. \tag{5.6}$$

The key claim in this section is that the first ℓ columns of the matrix $\mathbf{U}$ resulting form powerURV have exactly the same approximation accuracy as the columns of the matrix $\mathbf{U}_{\text{rsvd}}$ resulting from the RSVD, provided that the same random matrix is used. To be precise, we have:

Lemma: *Let $\mathbf{A}$ be an $m \times n$ matrix, let ℓ be a positive integer such that $\ell < \min(m, n)$, let q be a positive integer, and let $\mathbf{G}$ be a matrix of size $n \times n$. Let $\mathbf{A} = \mathbf{U}\mathbf{R}\mathbf{V}^{*}$ be the factorization resulting from the powerURV algorithm, as defined by (5.5) and (5.6) and using $\mathbf{G}$ as the starting point. Let $\mathbf{A} \approx \mathbf{U}_{\text{rsvd}}\mathbf{\Sigma}_{\text{rsvd}}\mathbf{V}_{\text{rsvd}}^{*}$ be the approximate factorization resulting from RSVD, as defined by (5.1)– (5.4)), starting with $\mathbf{G}_{\text{rsvd}} = \mathbf{G}(:, 1:\ell)$. Suppose that the rank of $(\mathbf{A}\mathbf{A}^{*})^{q}\mathbf{A}\mathbf{G}_{\text{rsvd}}$ is no lower than the rank of $\mathbf{A}$ (this holds with probability 1 when $\mathbf{G}_{\text{rsvd}}$ is Gaussian). Then*

$$\mathbf{U}(:, 1:\ell)\mathbf{U}(:, 1:\ell)^{*}\mathbf{A} = \mathbf{U}_{\text{rsvd}}\mathbf{U}_{\text{rsvd}}^{*}\mathbf{A}.$$

Proof. We can without loss of accuracy assume that the matrix $\mathbf{A}$ has rank at least ℓ. (If it is rank deficient, then the proof we give will apply for a modified $\ell' = \text{rank}(\mathbf{A})$, and it is easy to see that adding additional columns to the basis matrices will make no difference since in this case $\mathbf{U}(:, 1:\ell)\mathbf{U}(:, 1:\ell)^*\mathbf{A} = \mathbf{U}_{\text{rsvd}}\mathbf{U}_{\text{rsvd}}^*\mathbf{A} = \mathbf{A}$.)

We will prove that $\text{Ran}(\mathbf{U}(:, \ell)) = \text{Ran}(\mathbf{U}_{\text{rsvd}})$, which immediately implies that the projectors $\mathbf{U}(:, 1:\ell)\mathbf{U}(:, 1:\ell)^*$ and $\mathbf{U}_{\text{rsvd}}\mathbf{U}_{\text{rsvd}}^*$ are identical. Let us first observe that restricting (5.5) to the first ℓ columns, we obtain

$$(\mathbf{A}^*\mathbf{A})^q\mathbf{G}_{\text{rsvd}} = \mathbf{V}(:, 1:\ell)\mathbf{Z}(1:\ell, 1:\ell). \tag{5.7}$$

We can then connect $\mathbf{Y}_{\text{rsvd}}$ and $\mathbf{U}(:, 1:\ell)$ via a simple computation

$$\begin{aligned}
\mathbf{Y}_{\text{rsvd}} &\overset{(5.1)}{=} (\mathbf{A}\mathbf{A}^*)^q\mathbf{A}\mathbf{G}_{\text{rsvd}} \\
&= \mathbf{A}(\mathbf{A}^*\mathbf{A})^q\mathbf{G}_{\text{rsvd}} \\
&\overset{(5.7)}{=} \mathbf{A}\mathbf{V}(:, 1:\ell)\mathbf{Z}(1:\ell, 1:\ell) \\
&\overset{(5.6)}{=} \mathbf{U}(:, 1:\ell)\mathbf{R}(1:\ell, 1:\ell)\mathbf{Z}(1:\ell, 1:\ell).
\end{aligned} \tag{5.8}$$

Next we link $\mathbf{U}_{\text{rsvd}}$ and $\mathbf{Y}_{\text{rsvd}}$ via

$$\mathbf{U}_{\text{rsvd}} \overset{(5.4)}{=} \mathbf{Q}_{\text{rsvd}}\mathbf{W}_{\text{rsvd}} \overset{(5.2)}{=} \mathbf{Y}_{\text{rsvd}}\mathbf{R}_{\text{rsvd}}^{-1}\mathbf{W}_{\text{rsvd}}. \tag{5.9}$$

Combining (5.8) and (5.9), we find that

$$\mathbf{U}_{\text{rsvd}} = \mathbf{U}(:, 1:\ell)\mathbf{R}(1:\ell, 1:\ell)\mathbf{Z}(1:\ell, 1:\ell)\mathbf{R}_{\text{rsvd}}^{-1}\mathbf{W}_{\text{rsvd}}. \tag{5.10}$$

The rank assumption implies that the $\ell \times \ell$ matrix $\mathbf{R}(1:\ell, 1:\ell)\mathbf{Z}(1:\ell, 1:\ell)\mathbf{R}_{\text{rsvd}}^{-1}\mathbf{W}_{\text{rsvd}}$ is non-singular, which establishes that the matrices $\mathbf{U}_{\text{rsvd}}$ and $\mathbf{U}(:, 1:\ell)$ have the same range. □

The equivalency established in the Lemma between RSVD and POWERURV allows for much of the theory for analyzing the RSVD in [69] to directly apply to the powerURV algorithm.

It is important to note that while there is a close connection between RSVD and POWERURV, the RSVD is able to attain substantially higher overall accuracy than powerURV since it can take advantage of one additional application of $\mathbf{A}$. (To wit, RSVD requires $2q + 2$ applications of either $\mathbf{A}$ or $\mathbf{A}^*$, while

POWERURV requires only $2q + 1$.) This additional application makes the first k columns of $\mathbf{U}_{\text{rsvd}}$ a much better basis than the first k columns of $\mathbf{U}$, as long as k does not get close to ℓ. (One might say that the matrix $\mathbf{W}_{\text{rsvd}}$ rearranges the columns inside $\mathbf{Q}_{\text{rsvd}}$ to make the leading columns much better aligned with the corresponding singular vectors.) An additional way that the RSVD benefits from the additional application of $\mathbf{A}$ is that the columns of $\mathbf{V}_{\text{rsvd}}$ end up being a far more accurate basis for the row space of $\mathbf{A}$ than the columns of the matrix $\mathbf{V}$ resulting from the powerURV. This was of course expected since for $q = 0$, the matrix $\mathbf{V}$ incorporates no information from $\mathbf{A}$ at all.

5.4 The RANDUTV algorithm.

RANDUTV is a blocked algorithm for computing a rank revealing UTV decomposition of a matrix. It is tailored to run particularly efficiently on the GPU due to the fact that the bulk of its flops are cast in terms of matrix-matrix multiplication. Throughout this section, $\mathbf{A} \in \mathbb{R}^{m \times n}$ is the input matrix. Without loss of generality, we assume $m \geq n$ (if $m < n$, we may operate on $\mathbf{A}^*$ instead). The block size b is a user-defined parameter which in practice is usually a number such as 128 or 256. For simplicity, we assume in this section that b divides n evenly.

In Section 5.4.1, we present the RANDUTV algorithm of [89]. Then, Sections 5.4.2 and 5.4.3 present methods of modifying RANDUTV to enhance the rank-revealing properties of the resulting factorization. Finally, in Section 5.4.4 we combine these methods with a bootstrapping technique to reduce the cost of adding them to the algorithm, resulting in the revised algorithm RANDUTV_BOOSTED.

5.4.1 The RANDUTV algorithm for computing a UTV decomposition

Given an input matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, randUTV calculates a UTV factorization of $\mathbf{A}$ that is rank revealing with high probability. The algorithm performs the bulk of its work in a loop that executes n/b iterations. In the i-th iteration, a matrix $\mathbf{T}^{(i)}$ is formed by the computation

$$\mathbf{T}^{(i)} := (\mathbf{U}^{(i)})^* \mathbf{T}^{(i-1)} \mathbf{V}^{(i)}, \tag{5.11}$$

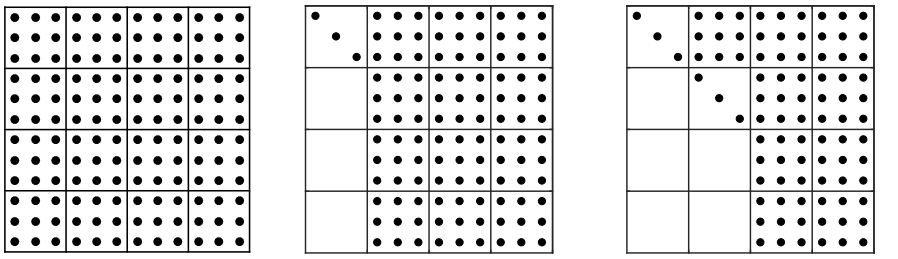


Figure 5.1: An illustration of the sparsity pattern followed by the first three $\mathbf{T}^{(i)}$ for RANDUTV if $n = 12, b = 3$. RANDUTV continues until the entirety of $\mathbf{T}^{(i)}$ is upper triangular. This figure was adapted from [95].

where $\mathbf{U}^{(i)}$ and $\mathbf{V}^{(i)}$ are orthogonal matrices chosen to effect the desired sparsity and rank revealing properties of the final factorization. Similarly to other blocked factorization algorithms, the i-th step is meant to "process" a set of b columns of $\mathbf{T}^{(i-1)}$, so that after step i, $\mathbf{T}^{(i)}$ satisfies the following sparsity requirements:

- $\mathbf{T}^{(i)}(:, 1 : ib)$ is upper triangular, and
- the $b \times b$ blocks on the main diagonal of $\mathbf{T}^{(i)}(:, 1 : ib)$ are themselves diagonal.

After the n/b-th iteration, $\mathbf{T} := \mathbf{T}^{(n/b)}$ is upper triangular, and the factorization is complete. The sparsity pattern followed by the $\mathbf{T}^{(i)}$ matrices is demonstrated in Figure 5.1. When the $\mathbf{U}$ and $\mathbf{V}$ matrices are desired, they can be built with the computations

$$\mathbf{U} := \mathbf{U}^{(1)}\mathbf{U}^{(2)} \cdots \mathbf{U}^{(n/b)},$$

$$\mathbf{V} := \mathbf{V}^{(1)}\mathbf{V}^{(2)} \cdots \mathbf{V}^{(n/b)}.$$

In practice, the $\mathbf{T}^{(i)}$, $\mathbf{U}^{(i)}$ and $\mathbf{V}^{(i)}$ are not stored separately to save memory. Instead, the space for $\mathbf{T}$, $\mathbf{U}$, and $\mathbf{V}$ is allocated at the beginning of randUTV, and at iteration i each is updated with

$$\mathbf{T} \leftarrow (\mathbf{U}^{(i)})^*\mathbf{T}\mathbf{V}^{(i)},$$

$$\mathbf{V} \leftarrow \mathbf{V}\mathbf{V}^{(i)},$$

$$\mathbf{U} \leftarrow \mathbf{U}\mathbf{U}^{(i)},$$

where $\mathbf{U}^{(i)}$ and $\mathbf{V}^{(i)}$ may be discarded or overwritten after the iteration completes.

5.4.1.1 Optimal choices for $\mathbf{U}^{(i)}$ and $\mathbf{V}^{(i)}$.

To motivate the process of building $\mathbf{U}^{(i)}$ and $\mathbf{V}^{(i)}$, consider the first step of randUTV. We begin by initializing $\mathbf{T} := \mathbf{A}$ and creating a helpful partition

$$\mathbf{T} = \begin{bmatrix} \mathbf{T}_{11} & \mathbf{T}_{12} \\ \mathbf{T}_{21} & \mathbf{T}_{22} \end{bmatrix},$$

where $\mathbf{T}_{11}$ is $b \times b$. The goal is to construct $\mathbf{V}^{(1)}$ and $\mathbf{U}^{(1)}$ such that, after the update $\mathbf{T} \leftarrow (\mathbf{U}^{(1)})^* \mathbf{T} \mathbf{V}^{(1)}$,

1. $\mathbf{T}_{11}$ is upper triangular,
2. $\mathbf{T}_{21} = \mathbf{0}$,
3. $\sigma_{\min}(\mathbf{T}_{11}) \approx \sigma_b(\mathbf{A})$,
4. $\sigma_{\max}(\mathbf{T}_{22}) \approx \sigma_{b+1}(\mathbf{A})$,
5. $\mathbf{T}_{11}(k,k) \approx \sigma_k(\mathbf{A})$, $k = 1, 2, \ldots, b$.

Items (1) and (2) in the list are basic requirements for any UTV factorization, and the rest of the items relate to the decomposition's rank revealing properties.

The key observation is that if $\mathbf{V}^{(1)}$ and $\mathbf{U}^{(1)}$ were orthogonal matrices whose leading b columns spanned the same subspace as the leading b right and left singular vectors, respectively, of $\mathbf{A}$, items (2)-(4) in the previous list would be satisfied perfectly. Items (1) and (5) could then be satisfied with an inexpensive post-processing step: compute the SVD of $\mathbf{T}_{11}$ and update $\mathbf{V}^{(1)}$ and $\mathbf{U}^{(1)}$ accordingly.[4] However, determining the singular vector subspaces is as difficult as computing a partial SVD of $\mathbf{A}$. We therefore content ourselves with the goal of building $\mathbf{V}^{(1)}$ and $\mathbf{U}^{(1)}$ such that the leading b columns are *approximations* of the subspace spanned by the leading b right and left singular subspaces, respectively, of $\mathbf{A}$.

5.4.1.2 Building $\mathbf{V}^{(i)}$ and $\mathbf{U}^{(i)}$.

To maintain simplicity in matrix dimensions, we will continue the thought of Section 5.4.1.1 of considering only the transformations in the first step, $\mathbf{U}^{(1)}$ and $\mathbf{V}^{(1)}$. Thanks to the observation of Section

[4] Recall that $\mathbf{T}_{11}$ is $b \times b$, where $b \ll n$, so this SVD is relatively cheap. See Remark 15 for details.

5.4.1.1, our goal for building $\mathbf{V}^{(1)}$ is to compute an orthogonal matrix such that the leading b columns of $\mathbf{V}^{(1)}$ span approximately the same subspace as the leading b right singular vectors of $\mathbf{A}$. We can achieve this goal efficiently by borrowing a key technique of the randomized SVD discussed in Section 5.2.6. Specifically, we build $\mathbf{V}^{(1)}$ as follows:

1. Generate a matrix $\mathbf{G} \in \mathbb{R}^{n \times b}$ whose entries are independent, identically distributed standard Gaussian variables, *i.e.* $g_{ij} \sim \mathcal{N}(0, 1)$.

2. Sample the row space of $\mathbf{A}$ by calculating $\mathbf{Y} = (\mathbf{A}^*\mathbf{A})^q\mathbf{A}^*\mathbf{G}$, where q is some small non-negative integer.

3. Compute an orthonormal basis for the column space of $\mathbf{Y}$ with an unpivoted QR decomposition, obtaining an orthogonal matrix $\mathbf{V}^{(1)}$ and upper triangular matrix $\mathbf{S}$ such that $\mathbf{Y} = \mathbf{V}^{(1)}\mathbf{S}_V$.

Once $\mathbf{V}^{(1)}$ is built, the path to $\mathbf{U}^{(1)}$ becomes clear after one observation. If the leading b columns of $\mathbf{V}^{(1)}$ span exactly the same subspace as the leading b right singular vectors of $\mathbf{A}$, then the leading b columns of $\mathbf{AV}^{(1)}$ span exactly the same subspace as the leading b *left* singular vectors of $\mathbf{A}$. Therefore, after computing $\mathbf{V}^{(1)}$, we build $\mathbf{U}^{(1)}$ as follows:

1. Perform the matrix multiplication $\mathbf{B} = \mathbf{AV}^{(1)}$.

2. Compute an orthonormal basis for the column space of $\mathbf{B}$ with an unpivoted QR decomposition to obtain $\mathbf{U}^{(1)}$ and $\mathbf{S}_U$ such that $\mathbf{B} = \mathbf{U}^{(1)}\mathbf{S}_U$.

5.4.1.3 Putting it all together.

We are now ready to combine the ideas of Sections 5.4.1.1 and 5.4.1.2 to write the basic version of randUTV. This is Algorithm 7, adapted from [95]. Note that we have omitted some details important to allowing the algorithm to run efficiently, such as fast application of the transformation matrices $\mathbf{U}^{(i)}$ and $\mathbf{V}^{(i)}$ to the $\mathbf{T}^{(i)}$ matrices. For a more in-depth discussion, see [95]. The result of randUTV is a decomposition $\mathbf{A} = \mathbf{UTV}^*$ such that

- $\mathbf{T}$ is upper triangular, and the $b \times b$ blocks on its main diagonal are themselves diagonal,

- $\mathbf{U}$ and $\mathbf{V}$ are orthogonal.
- $\|\mathbf{A} - \mathbf{U}(:, 1:k)\mathbf{T}(1:k, 1:k)\,(\mathbf{V}(:, 1:k)))^*\| \approx \sigma_{k+1}(\mathbf{A}), k = 1, \ldots, n$. That is, the error in the low rank approximations provided by $\mathbf{U}, \mathbf{T}, \mathbf{V}$ are close to optimal.
- $\mathbf{T}(i, i) \approx \sigma_i(\mathbf{A}), i = 1, \ldots, n$.

5.4.2 Using oversampling in the RANDUTV algorithm

In randUTV, the computation of matrix $\mathbf{V}^{(i)}$ is modeled after the approximation of a matrix's column space in the randomized SVD discussed in Section 5.2.6. Just as the randomized SVD uses an oversampling parameter p to improve the probability that the column space approximation succeeds, we may add a similar parameter p to the construction of $\mathbf{V}^{(i)}$ in Algorithm 7 to improve the rank revealing properties of the resulting factorization.

To do so, we first change the size of the random matrix $\mathbf{G}$ from $m \times b$ to $m \times (b + p)$ (we once again consider only the building of $\mathbf{V}^{(1)}$ to simplify matrix dimensions). This effectively increases the number of times we sample the column space of $\mathbf{A}$ by p, providing a "correction" to the information in the first b samples.

Next, we must modify how we obtain the orthonormal vectors that form $\mathbf{V}^{(1)}$. Recall that *the first b columns* of $\mathbf{V}^{(1)}$ must contain the approximation to the leading b right singular vectors of $\mathbf{A}$. If we merely orthonormalized the columns of $\mathbf{Y}$ with HQR again, the first b columns of $\mathbf{V}^{(1)}$ would only contain information from the first b columns of $\mathbf{Y}$. We must therefore choose a method for building $\mathbf{V}^{(1)}$ such that its first b columns contain a good approximation of the column space of $\mathbf{Y}$. The following approaches use two of the most common rank revealing factorizations to solve this subproblem:

a) Perform b steps of HQRCP on $\mathbf{Y}$ to obtain the factorization $\mathbf{Y} = \mathbf{V}^{(1)}\mathbf{S}_V\mathbf{P}^*$, where $\mathbf{S}_V(1:b, :)$ is upper triangular and $\mathbf{P}$ is a permutation matrix. The approximation provided by $\mathbf{V}^{(1)}(:, 1:b)$ to the column space of $\mathbf{Y}$ in this case will be suboptimal.

b) Perform an SVD on $\mathbf{Y}$ to obtain the factorization $\mathbf{Y} = \mathbf{W}\mathbf{D}\mathbf{X}^*$. Then $\mathbf{W}(:, 1:b)$ contains the

Algorithm 7 [**U**,**T**,**V**] = RANDUTV_BASIC(**A**,b,q)

$\mathbf{T} = \mathbf{A}; \mathbf{U} = \mathbf{I}_m; \mathbf{V} = \mathbf{I}_n;$
for $i = 1$: $\min(\lceil m/b, \rceil, \lceil n/b \rceil)$ **do**
  $I_1 = 1 : (b(i-1)); I_2 = (b(i-1)+1) : \min(bi, m); I_3 = (bi+1) : m;$
  $J_1 = 1 : (b(i-1)); J_2 = (b(i-1)+1) : \min(bi, n); J_3 = (bi+1) : n;$
  if (I_3 and J_3 are both nonempty) **then**
    $\mathbf{G} = \text{GENERATE\_NORM\_RAND\_IID}(m - b(i-1), b)$
    $\mathbf{Y} = \mathbf{T}([I_2, I_3], [J_2, J_3])^* \mathbf{G}$
    for $j = 1 : q$ **do**
      $\mathbf{Y} = \mathbf{T}([I_2, I_3], [J_2, J_3])^* (\mathbf{T}([I_2, I_3], [J_2, J_3]) \mathbf{Y})$
    end for
    $[\mathbf{V}_l, \sim] = \text{HQR}(\mathbf{Y})$
    $\mathbf{T}(:, [J_2, J_3]) = \mathbf{T}(:, [J_2, J_3]) \mathbf{V}_l$
    $\mathbf{V}(:, [J_2, J_3]) = \mathbf{V}(:, [J_2, J_3]) \mathbf{V}_l$

    $[\mathbf{U}_l, \mathbf{R}] = \text{HQR}(\mathbf{T}([I_2, I_3], J_2))$
    $\mathbf{U}(:, [I_2, I_3]) = \mathbf{U}(:, [I_2, I_3]) \mathbf{U}_l$
    $\mathbf{T}([I_2, I_3], J_3) = \mathbf{U}_l^* \mathbf{T}([I_2, I_3], J_3)$
    $\mathbf{T}(I_3, J_2) = \text{ZEROS}(m - bi, b)$

    $[\mathbf{U}_{\text{small}}, \mathbf{D}_{\text{small}}, \mathbf{V}_{\text{small}}] = \text{SVD}(\mathbf{R}(1 : b, 1 : b))$
    $\mathbf{T}(I_2, J_2) = \mathbf{D}_{\text{small}}$
    $\mathbf{T}(I_2, J_3) = \mathbf{U}_{\text{small}}^* \mathbf{T}(I_2, J_3)$
    $\mathbf{U}(:, I_2) = \mathbf{U}(:, I_2) \mathbf{U}_{\text{small}}$
    $\mathbf{T}(I_1, J_2) = \mathbf{T}(I_1, J_2) \mathbf{V}_{\text{small}}$
    $\mathbf{V}(:, J_2) = \mathbf{V}(:, J_2) \mathbf{V}_{\text{small}}$
  else
    $[\mathbf{U}_{\text{small}}, \mathbf{D}_{\text{small}}, \mathbf{V}_{\text{small}}] = \text{SVD}(\mathbf{T}([I_2, I_3], [J_2, J_3]))$
    $\mathbf{U}(:, [I_2, I_3]) = \mathbf{U}(:, [I_2, I_3]) \mathbf{U}_{\text{small}}$
    $\mathbf{V}(:, [J_2, J_3]) = \mathbf{V}(:, [J_2, J_3]) \mathbf{V}_{\text{small}}$
    $\mathbf{T}([I_2, I_3], [J_2, J_3]) = \mathbf{D}_{\text{small}}$
    $\mathbf{T}([I_1, [J_2, J_3]) = \mathbf{T}(I_1, [J_2, J_3]) \mathbf{V}_{\text{small}}$
  end if
end for

optimal rank-b approximation of the column space of $\mathbf{Y}$. However, this method requires one more step since $\mathbf{V}^{(1)}$ must be represented as a product of Householder vectors in order to compute $\mathbf{AV}^{(1)}$ in the following step of randUTV. After computing the SVD, therefore, we must perform HQR on $\mathbf{W}(:, 1:b)$, resulting in $\mathbf{W}(:, 1:b) = \mathbf{V}^{(1)}\mathbf{S}_W$. This method yields the optimal approximation of $\mathbf{V}^{(1)}(:, 1:b)$ to the subspace of $\mathbf{Y}$.

Though method b) provides better approximations, method a) is more efficient. The only additional computational expense for a) is that of adding pivoting HQR, which is relatively inexpensive for thin matrices like $\mathbf{Y}$. As discussed in Remark 15, method b) requires an HQR of size $n \times b+p$, an SVD of size $b+p \times b+p$, and an HQR of size $n \times b$. While the SVD is small and therefore quite cheap, the first HQR is an extra expense which is nontrivial when aggregated across every iteration in randUTV. This extra cost is addressed and mitigated in Section 5.4.4.

Remark 15. *The SVD of method b) above may look expensive at first glance. However, recall that* $\mathbf{Y}$ *is of size* $n \times (b+p)$*, where* $n \gg b+p$*. For tall, thin matrices like this, the economic SVD may be inexpensively computed as follows:*

1. *Obtain an economic QR of* $\mathbf{Y}$*, yielding* $\mathbf{Y} = \mathbf{QR}$*;* $\mathbf{Q} \in \mathbb{R}^{n \times b+p}, \mathbf{R} \in \mathbb{R}^{b+p \times b+p}$.
2. *Compute SVD of* $\mathbf{R}$ *to obtain* $\mathbf{R} = \hat{\mathbf{U}}\mathbf{D}\mathbf{V}^*$*;* $\hat{\mathbf{U}}, \mathbf{D}, \mathbf{V} \in \mathbb{R}^{b+p \times b+p}$.
3. *Compute* $\mathbf{U} = \mathbf{Q}\hat{\mathbf{U}}$.

After these steps, the result is

$$\mathbf{Y} = \mathbf{QR} = (\mathbf{Q}\hat{\mathbf{U}})\mathbf{D}\mathbf{V}^* = \mathbf{U}\mathbf{D}\mathbf{V}^*,$$

the economic SVD of $\mathbf{Y}$.

5.4.3 Orthonormalization to enhance accuracy

For subspace iteration techniques like the one used to build $\mathbf{V}^{(i)}$, the stability of the iteration is often a concern. As mentioned in Section 5.3.3, the information from any singular values less than $\epsilon_{\text{mach}}^{1/(2q+1)}$ will be lost unless an orthonormalization procedure is used during intermediate steps of the computation

$\mathbf{Y} = (\mathbf{A}^*\mathbf{A})^q\,\mathbf{A}^*\mathbf{G}$. For randUTV, only b singular vectors are in view in each iteration, so orthonormalization is not required as often as it is for powerURV.

However, if oversampling is used (*i.e.* $p > 0$), performing one orthonormalization before the final application of $\mathbf{A}^*$ markedly benefits the approximation of the column space of $\mathbf{Y}$ to the desired singular vector subspace. Numerical experiments show that this improvement occurs even when the sampling matrix is not in danger of loss of information from roundoff error. Instead, we may intuitively consider that using orthonormal columns to sample $\mathbf{A}^*$ ensures that the "extra" p columns of $\mathbf{Y}$ contain information not already accounted for in the first b columns.

5.4.4 Reducing the cost of oversampling and orthonormalization

Adding oversampling to RANDUTV_BASIC, as discussed in Sections 5.4.2 and 5.4.3, adds noticeable cost to the algorithm. First, it requires a costlier method of building $\mathbf{V}^{(i)}$. Second, it increases the dimension of the random matrix $\mathbf{G}$, increasing the cost of all the operations involving $\mathbf{G}$ and, therefore, $\mathbf{Y}$. By reusing information from a previous iteration, however, we can essentially eliminate the cost of the second item.

To demonstrate, consider the state of randUTV for input matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ with $q = 2, p = b$ after one step of the main iteration. Let $\mathbf{A}_q = (\mathbf{A}^*\mathbf{A})^q\mathbf{A}^*$ denote the sampling matrix for power iteration. At this point, we have computed and stored the following relevant matrices:

- $\mathbf{Y}^{(1)} \in \mathbb{R}^{n\times(b+p)}$: The matrix containing random samples of the row space of $\mathbf{A}$, computed with $\mathbf{Y}^{(1)} = \mathbf{A}_q\mathbf{G}^{(1)}$, where $\mathbf{G}^{(1)}$ is a random matrix.
- $\mathbf{W}^{(1)} \in \mathbb{R}^{n\times(b+p)}$: The matrix whose columns are the left singular vectors of $\mathbf{Y}^{(1)}$.
- $\mathbf{V}^{(1)} \in \mathbb{R}^{n\times n}$: The right transformation in the main step of randUTV, computed with $[\mathbf{V}^{(1)}, \sim] = \mathrm{HQR}(\mathbf{W}^{(1)}(:, 1:b))$.
- $\mathbf{U}^{(1)} \in \mathbb{R}^{m\times m}$: The left transformation in the main step of randUTV, computed with $[\mathbf{U}^{(1)}, \sim] = \mathrm{HQR}(\mathbf{A}\mathbf{V}^{(1)})$.
- $\mathbf{T} \in \mathbb{R}^{m\times n}$: The matrix being driven to upper triangular form. At this stage in the algorithm,

$$\mathbf{T} = (\mathbf{U}^{(1)})^* \mathbf{A} \mathbf{V}^{(1)}.$$

Finally, consider the partitions

$$\mathbf{W}^{(1)} = \begin{bmatrix} \mathbf{W}_1^{(1)} & \mathbf{W}_2^{(1)} \end{bmatrix},$$
$$\mathbf{V}^{(1)} = \begin{bmatrix} \mathbf{V}_1^{(1)} & \mathbf{V}_2^{(1)} \end{bmatrix},$$
$$\mathbf{U}^{(1)} = \begin{bmatrix} \mathbf{U}_1^{(1)} & \mathbf{U}_2^{(1)} \end{bmatrix},$$
$$\mathbf{T} = \begin{bmatrix} \mathbf{T}_{11} & \mathbf{T}_{12} \\ \mathbf{T}_{21} & \mathbf{T}_{22} \end{bmatrix},$$

where $\mathbf{W}_1^{(1)}$ is $n \times b$, $\mathbf{V}_1^{(1)}$ and $\mathbf{U}^{(1)}$ are $n \times b$, and $\mathbf{T}_{11}$ is $b \times b$.

In the second iteration of randUTV, the first step is to approximate the subspace spanned by the column space of $\mathbf{T}_{22}^*$ with RANGE_FINDER (Algorithm 3 with $p = 0$, and orthonormalization is usually not required). Since $\mathbf{T} = (\mathbf{U}^{(1)})^* \mathbf{T} \mathbf{V}^{(1)}$, we have

$$\mathbf{T}_{22}^* = (\mathbf{V}_2^{(1)})^* \mathbf{A}^* \mathbf{U}_2^{(1)}.$$

Therefore, the subspace we must approximate is the column space of $\mathbf{A}^*$ after being mapped into the span of the columns of $(\mathbf{V}_2^{(1)})^*$ by matrix multiplication. Next, we make the observation that, just as $\mathbf{W}_1^{(1)}$ approximates the subspace spanned by the leading b right singular vectors of $\mathbf{A}$, the columns of $\mathbf{W}_2^{(1)}$ approximate the subspace spanned by the leading b through p-th right singular vectors of $\mathbf{A}$. Thus, $(\mathbf{V}_2^{(1)})^* \mathbf{W}_2^{(1)}$ is an approximation of the column space of $\mathbf{A}^*$ after being mapped into Span$((\mathbf{V}_2^{(1)})^*)$. This multiplication involving two small matrix dimensions is much cheaper than carrying out the full power iteration process.

Putting it all together, on every iteration of randUTV after the first, we build the sampling matrix $\mathbf{Y}^{(i)}$ in two stages. First, we build $\mathbf{Y}^{(i)}(:, 1:b)$ with the standard RANGE_FINDER described in Section 3 (with $p = 0$, and orthonormalization, again, is usually not required). Second, we build compute $\mathbf{Y}^{(i)}(:, (b+1):(b+p))$ by reusing the samples from the last iteration with $\mathbf{Y}^{(i)}(:, (b+1):(b+p)) = (\mathbf{V}^{(i-1)})^* \mathbf{W}^{(i-1)}(:, (b+1):(b+p))$. The complete algorithm, adjusted to improve upon the low rank estimation properties RANDUTV_BASIC, is given in 8.

Algorithm 8 [**U**,**T**,**V**] = RANDUTV_BOOSTED(**A**,b,q,p)

$\mathbf{T} = \mathbf{A}; \mathbf{U} = \mathbf{I}_m; \mathbf{V} = \mathbf{I}_n;$
for $i = 1$:min($\lceil m/b, \rceil, \lceil n/b \rceil$) **do**
  $I_1 = 1 : (b(i-1)); I_2 = (b(i-1)+1) : \min(bi, m); I_3 = (bi+1) : m;$
  $J_1 = 1 : (b(i-1)); J_2 = (b(i-1)+1) : \min(bi, n); J_3 = (bi+1) : n;$
  if (I_3 and J_3 are both nonempty) **then**
    if ($i == 1$) **then**
      $\mathbf{G} = \text{GENERATE\_NORM\_RAND\_IID}(m - b(i-1), b+p)$
    else
      $\mathbf{G} = \text{GENERATE\_NORM\_RAND\_IID}(m - b(i-1), b)$
    end if
    $\mathbf{Y}(:, 1:b) = \mathbf{T}([I_2, I_3], [J_2, J_3])^* \mathbf{G}$
    for $j = 1 : q$ **do**
      if $j < q$ **then**
        $\mathbf{Y}(:, 1:b) = \mathbf{T}([I_2, I_3], [J_2, J_3])^* (\mathbf{T}([I_2, I_3], [J_2, J_3])\mathbf{Y})$
      else
        $\mathbf{Y}(:, 1:b) = \mathbf{T}([I_2, I_3], [J_2, J_3])\mathbf{Y}(:, 1:b)$
        $\mathbf{Y}(:, 1:b) = \mathbf{Y}(:, 1:b) - \mathbf{W}_{\text{next}}(:, (b+1):(b+p))\mathbf{W}_{\text{next}}(:, (b+1):(b+p))^* \mathbf{Y}(:, 1:b)$
        $[\mathbf{Y}(:, 1:b), \sim] = \text{HQR}(\mathbf{Y}(:, 1:b))$
        $\mathbf{Y}(:, (b+1):(b+p)) = \mathbf{W}_{\text{next}}(:, (b+1):(b+p))$
        $\mathbf{Y} = \mathbf{T}([I_2, I_3], [J_2, J_3])^* \mathbf{Y}$
      end if
    end for
    $[\mathbf{W}_V, \mathbf{D}_V, \mathbf{X}_V] = \text{SVD}(\mathbf{Y})$
    $[\mathbf{V}_l, \ ] = \text{HQR}(\mathbf{W}_V(:, 1:b))$
    $\mathbf{T}(:, [J_2, J_3]) = \mathbf{T}(:, [J_2, J_3])\mathbf{V}_l$
    $\mathbf{V}(:, [J_2, J_3]) = \mathbf{V}(:, [J_2, J_3])\mathbf{V}_l$
    $\mathbf{W}_{\text{next}} = \mathbf{V}_l^* \mathbf{W}_V(:, (b+1):(b+p))$

    $[\mathbf{U}_l, \mathbf{R}] = \text{HQR}(\mathbf{T}([I_2, I_3], J_2))$
    $\mathbf{U}(:, [I_2, I_3]) = \mathbf{U}(:, [I_2, I_3])\mathbf{U}_l$
    $\mathbf{T}([I_2, I_3], J_3) = \mathbf{U}_l^* \mathbf{T}([I_2, I_3, J_3])$
    $\mathbf{T}(I_3, J_2) = \text{ZEROS}(m - bi, b)$

    $[\mathbf{U}_{\text{small}}, \mathbf{D}_{\text{small}}, \mathbf{V}_{\text{small}}] = \text{SVD}(\mathbf{R}(1:b, 1:b))$
    $\mathbf{T}(I_2, J_2) = \mathbf{D}_{\text{small}}$
    $\mathbf{T}(I_2, J_3) = \mathbf{U}_{\text{small}}^* \mathbf{T}(I_2, J_3)$
    $\mathbf{U}(:, I_2) = \mathbf{U}(:, I_2)\mathbf{U}_{\text{small}}$
    $\mathbf{T}(I_1, J_2) = \mathbf{T}(I_1, J_2)\mathbf{V}_{\text{small}}$
    $\mathbf{V}(:, J_2) = \mathbf{V}(:, J_2)\mathbf{V}_{\text{small}}$
  else
    $[\mathbf{U}_{\text{small}}, \mathbf{D}_{\text{small}}, \mathbf{V}_{\text{small}}] = \text{SVD}(\mathbf{T}([I_2, I_3], [J_2, J_3]))$
    $\mathbf{U}(:, [I_2, I_3]) = \mathbf{U}(:, [I_2, I_3])\mathbf{U}_{\text{small}}$
    $\mathbf{V}(:, [J_2, J_3]) = \mathbf{V}(:, [J_2, J_3])\mathbf{V}_{\text{small}}$
    $\mathbf{T}([I_2, I_3], [J_2, J_3]) = \mathbf{D}_{\text{small}}$
    $\mathbf{T}([I_1, [J_2, J_3]) = \mathbf{T}(I_1, [J_2, J_3])\mathbf{V}_{\text{small}}$
  end if
end for

5.5 Numerical results

Experiments in this article were performed on an NVIDIA Tesla K40c graphics card (2880 cores). The code makes extensive use of the MAGMA (Version 2.2.0) library linked with Intel's MKL (Version 2017.0.3). It was compiled with the NVIDIA compiler NVCC (Version 8.0.61) on the Linux OS (4.15.0-45-generic.x86_64).

5.5.1 Computational speed

In this section, we investigate the speed of RANDUTV on the GPU and compare it to highly optimized implementations of competing rank-revealing factorizations for the GPU. In Figures 5.2 and 5.3, every factorization is completed on a random matrix $\mathbf{A}$ with entries drawn independently from the standard normal distribution. We compare the following times (all in seconds):

- T_{svd}: The time to compute the SVD of $\mathbf{A}$ using the MAGMA routine MAGMA_DGESDD, where all orthogonal matrices are calculated. The code is linked with the MKL implementation of the BLAS.
- T_{cpqr}: The time to execute HQRCP for $\mathbf{A}$ using the MAGMA routine MAGMA_DGEQP3, where the orthogonal matrix $\mathbf{Q}$ is calculated. The code is linked with the MKL implementation of the BLAS.
- T_{randUTV}: The time to execute RANDUTV for $\mathbf{A}$ for varying parameter choices, described in the figure captions. The code uses the MAGMA library linked with the MKL implementation of the BLAS.
- T_{powerURV}: The time to execute POWERURV for $\mathbf{A}$ for varying parameter choices, described in the figure captions. The code uses the MAGMA library linked with the MKL implementation of the BLAS.

In each plot, we divide the given time by n^3 to make the asymptotic relationships between each experiment more clear.

We observe in Figure 5.2 that RANDUTV_BASIC handily outperforms the competition in terms of speed. The cost of increasing the parameter q is also quite small due to the high efficiency of matrix

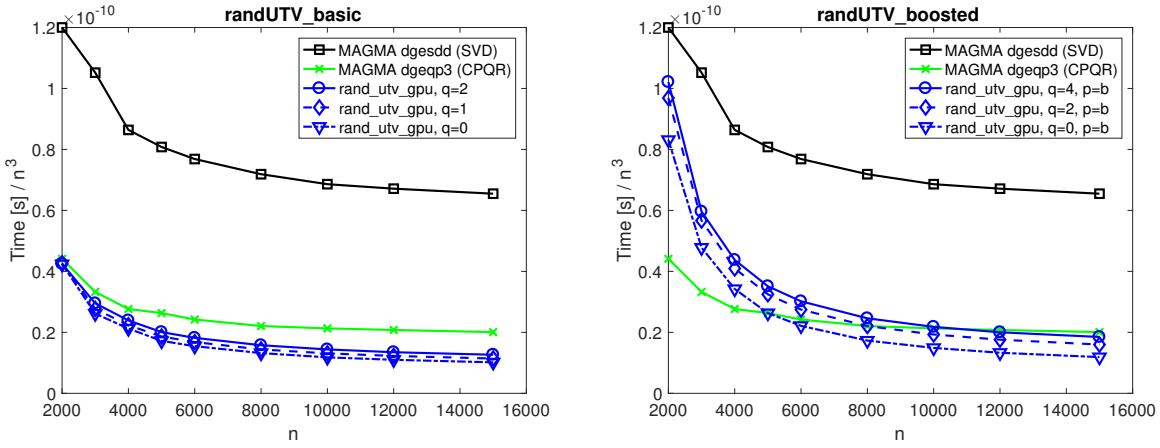


Figure 5.2: Computation times for RANDUTV plotted against the computation time for HQRCP and the SVD on the GPU. Left: The most basic, fastest RANDUTV_BASIC algorithm ($p = 0$). The block size was $b = 128$. Right: The accuracy-enhanced RANDUTV_BOOSTED with $b = 128, p = b$.

multiplication on the GPU. We see that it takes parameters of $p = b, q = 4$ of the highly rank-revealing RANDUTV_BOOSTED before the execution time even matches HQRCP. Finally, observe that the distance between the lines for RANDUTV_BOOSTED with $q = 4, p = b$ and $q = 2, p = b$ is less than the distance between the lines for $q = 0, p = b$ and $q = 2, p = b$. This difference is representative of the savings gained with bootstrapping technique whereby extra samples from one iteration are carried over to the next iteration.

5.5.2 Approximation error

In this section, we compare the errors in the low rank approximations produced by RANDUTV_BOOSTED and competing rank-revealing factorizations. Each rank-revealing factorization in this section produces a decomposition

$$\mathbf{A} = \mathbf{U}\mathbf{R}\mathbf{V}^*,$$

where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal and $\mathbf{R}$ is upper triangular. Given this factorization, a natural rank-k approximation to $\mathbf{A}$ is

$$\mathbf{A}_k = \mathbf{U}(:, 1:k)\mathbf{R}(1:k, :)\mathbf{V}^*.$$

For each of the factorizations that we study, we evaluated the spectral norm errors

$$e_k = \|\mathbf{A} - \mathbf{A}_k\|$$

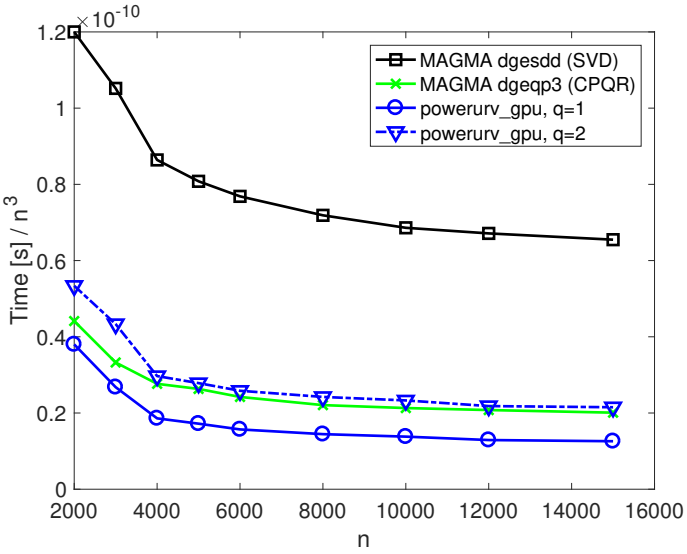


Figure 5.3: Computation times for the POWERURV computed on the GPU. These times are plotted against the computation time for HQRCP and the SVD on the GPU.

as a function of k. Specifically, the factorization techniques that we investigated were:

1. DGESDD: One of two common algorithms for computing an SVD.

2. DGEQP3: The HQRCP algorithm for computing a rank-revealing QR factorization.

3. RANDUTV_BASIC: The original RANDUTV algorithm introduced in [89], with no rank-revealing enhancement from oversampling or orthonormalization.

4. RANDUTV_BOOSTED: The RANDUTV algorithm developed in this article, modified to benefit maximally from oversampling ($p > 0$).

We also examine the relative error in the approximations produced by the factorizations

$$\frac{\|\mathbf{A} - \mathbf{A}_k\|}{\|\mathbf{A} - \mathbf{A}_k^{\text{optimal}}\|},$$

where $\mathbf{A}_k^{\text{optimal}}$ is the $\mathbf{A}_k$ produced by the SVD.

We compare the errors for several input matrices $\mathbf{A}$:

- **Fast decay:** This matrix is generated by first creating random orthogonal matrices $\mathbf{U}$ and $\mathbf{V}$ by performing unpivoted QR factorizations on two random matrices with i.i.d entries drawn according

to the standard normal distribution. Then, $\mathbf{A}_{\text{fast}}$ is computed with $\mathbf{A}_{\text{fast}} = \mathbf{UDV}^*$, where $\mathbf{D}$ is a diagonal matrix with diagonal entries $d_i i = \beta^{(i-1)/(n-1)}$, where $\beta = 10^{-5}$.

- **S-shaped decay:** This matrix is generated in the same manner as "fast decay," but the diagonal entries are chosen to first hover around 1, then quickly decay before leveling out at 10^{-2}, as shown in Figure 5.5.

- **BIE:** This matrix is the result of discretizing a Boundary Integral Equation (BIE) defined on a smooth closed curve in the plane. To be precise, we discretized the so called "single layer" operator associated with the Laplace equation using a sixth order quadrature rule designed by Alpert [2]. This operator is well-known to be ill-conditioned, which necessitates the use of a rank-revealing factorization in order to solve the corresponding linear system in as stable a manner as possible.

From these experiments, we make a few observations. First, when there is fast singular value decay in $\mathbf{A}$, RANDUTV_BOOSTED provides a significant boost in low-rank approximations over RANDUTV_BASIC. It also succeeds in decreasing the variance across approximations for different values of k. Second, RANDUTV_BOOSTED still provides noticeable correction to RANDUTV_BASIC when there is slow singular decay in $\mathbf{A}$, though the difference is not as pronounced in the relative error measurement.

Acknowledgements: The research reported was supported by the National Science Foundation, under the awards DMS-1407340 and DMS-1620472, by the Office of Naval Research, under the grant N00014-18-1-2354, and by DARPA under the grant N00014-18-1-2354. The authors also wish to express their appreciation to Nvidia for the donation of the GPUs used for the development work.

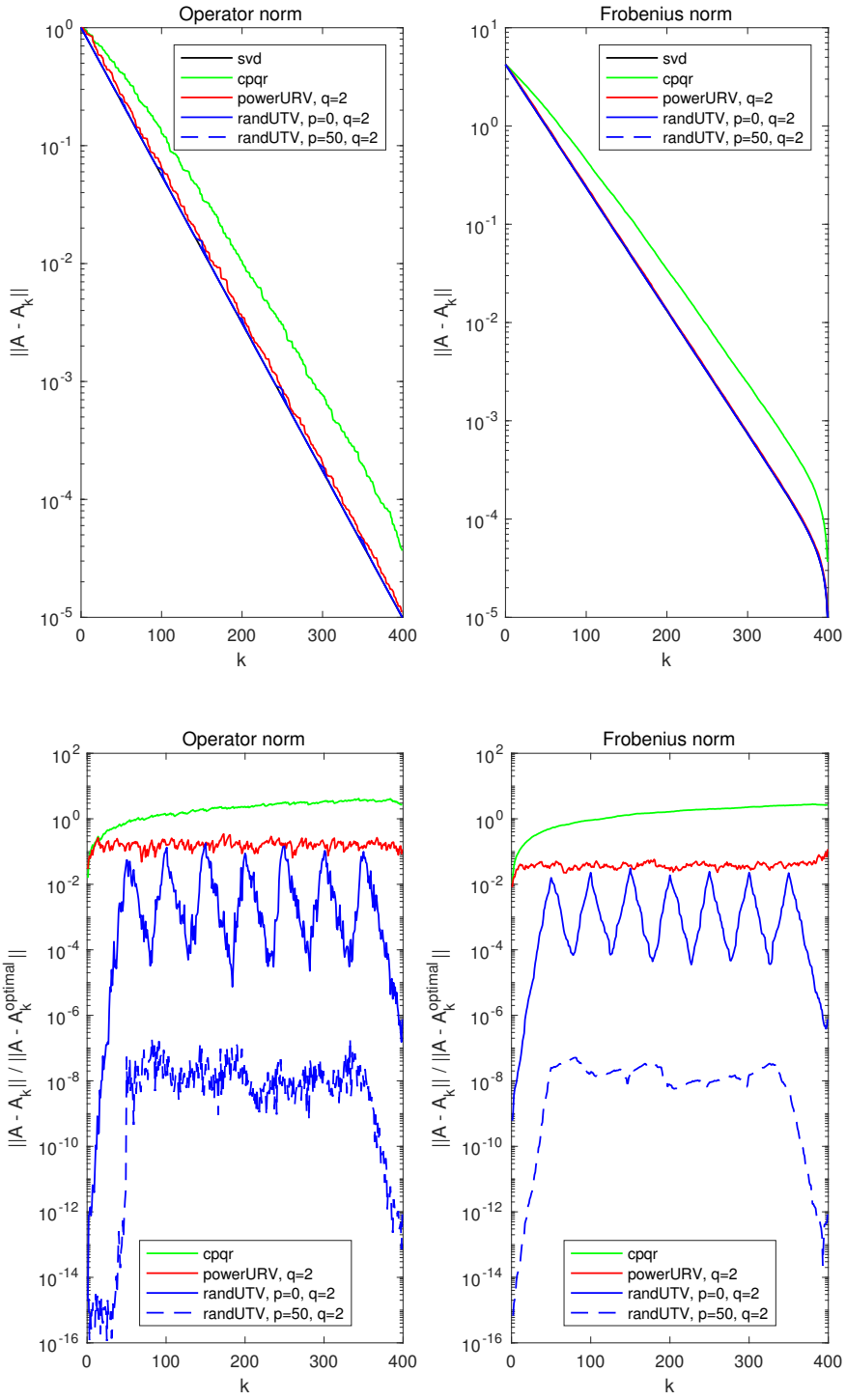


Figure 5.4: Errors in low rank approximation for the matrix "fast decay" with $n = 400$. For the RANDUTV factorizations, the block size was $b = 50$.

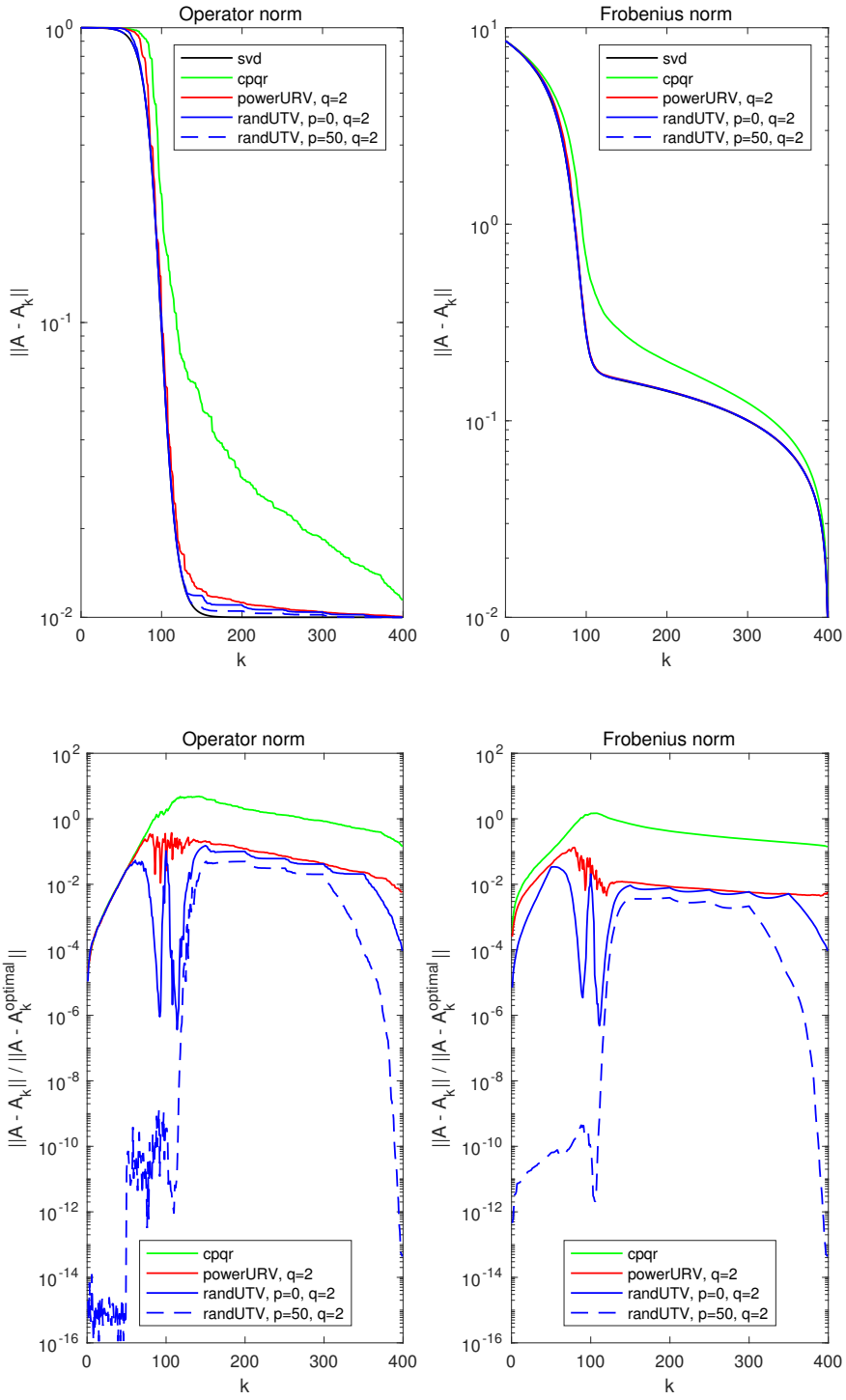


Figure 5.5: Errors in low rank approximation for the matrix "S-shaped decay" with $n = 400$. For the RANDUTV factorizations, the block size was $b = 50$.

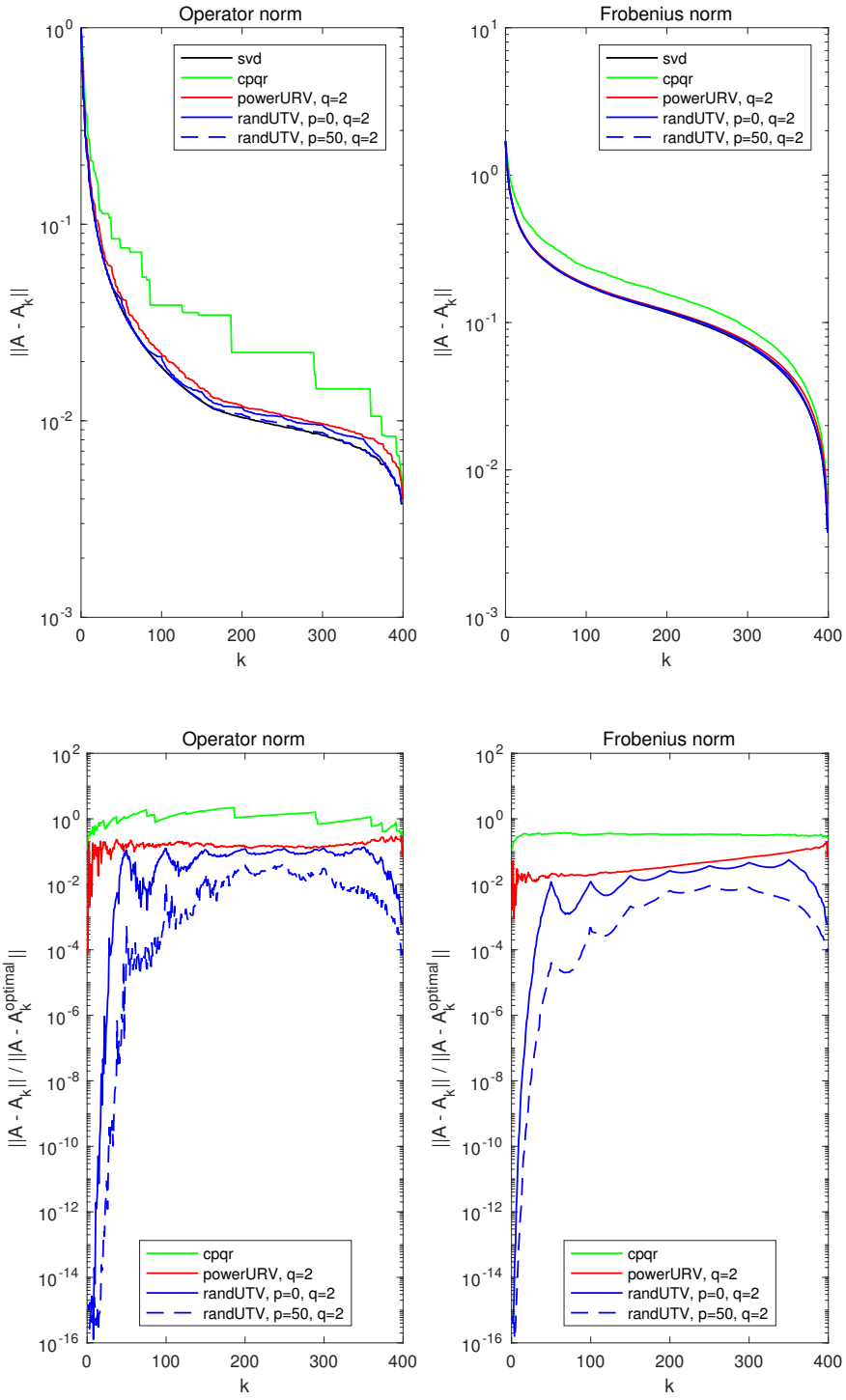


Figure 5.6: Errors in low rank approximation for the matrix "BIE" with $n = 400$. For the RANDUTV factorizations, the block size was $b = 50$.

Chapter 6

Computing rank-revealing factorizations of matrices stored out-of-core

It is often necessary to compute a rank revealing factorization of a matrix that is so large that it must be stored on a slow memory device such as a solid state or spinning hard drive. For a matrix stored in RAM, there are several well-established techniques to choose from, including computing a full singular value decomposition (SVD), Krylov techniques, and column pivoted QR (CPQR). However, none of these techniques are easily implemented when the matrix is stored out-of-core, as they typically are made up of a sequence of matrix-vector operators, which necessitates reading (and sometimes writing) large amounts of data at each step. This chapter demonstrates how randomization can be used to compute rank-revealing matrix factorizations by processing large blocks of the matrix at a time, which reduces the amount of data movement required. Techniques are presented for reorganizing randomized rank-revealing factorizations methods as either a left-looking algorithm or as an algorithm-by-blocks, which further reduces communications. Numerical experiments demonstrate that the new methods allow factorizations of matrices whose size make them inaccessible to classical deterministic methods.

6.1 Introduction

Consider an $m \times n$ matrix $\mathbf{A}$ that is so large it must be stored on an external memory device such as a spinning or solid disk drive. Often, a matrix decomposition

$$\begin{array}{ccccccc} \mathbf{A} & = & \mathbf{U} & & \mathbf{R} & \mathbf{V}^*, & \\ m \times n & & m \times m & m \times n & n \times n & & \end{array} \tag{6.1}$$

may be desired. For any k with $1 \leq k \leq \min(m, n)$, there is a natural rank-k approximation associated with the factorization, specifically $\mathbf{A}_k = \mathbf{U}(:, 1:k)\mathbf{R}(1:k, :)\mathbf{V}^*$. In many scenarios, obtaining a decomposition such that $\|\mathbf{A} - \mathbf{A}_k\|$ is small is vital. Problems which make use of such a factorization include rank-deficient and total least squares problems [80, 56, 123, 80], subset selection [97], and matrix approximation [45, 85], among others. The goal of minimizing $\|\mathbf{A} - \mathbf{A}_k\|$ is closely related to the idea of obtaining a *rank-revealing* factorization. While the definition of what constitutes a "rank-revealing factorization" varies slightly in the literature [27, 114, 24, 26], a recurring theme is that the approximation errors $\|\mathbf{A} - \mathbf{A}_k\|$ should be close to their theoretically minimal values in some sense. Therefore, in this chapter we will take a factorization that "is rank-revealing" and that "provides good rank-k approximations" to mean the same thing. The lack of specificity in our use of the term "rank-revealing" will also allow us to make comparisons. We will say that one factorization reveals rank better than another if it provides rank-k approximations with lower error than its competitor.

There are several well established options for computing a rank-revealing factorization. If a full factorization is required, the singular value decomposition (SVD) and QR decomposition with column pivoting (CPQR) are two popular options. The SVD provides theoretically optimal low rank approximations but can be prohibitively expensive to compute. The CPQR usually reveals rank reasonably well (see, *e.g.* [76] for a notable exception) and requires much less work than the SVD to compute. Neither factorization is easily implemented when the matrix is stored out-of-core, however. The traditional algorithms for computing each require mostly matrix-vector operations, which in turn necessitates many reads, and sometimes writes, to and from the main matrix in external memory. See, *e.g.* [122, 57, 78] for information on standard algorithms for computing the SVD, and [54, 116, 107] for information on the most common CPQR algorithm and its restriction to matrix-vector operations.

A truncated SVD algorithm that is effective for out-of-core matrices was introduced in [68]. This technique, however, only yields a *partial* factorization, and it requires that an upper bound for the target rank k is known in advance. Other software efforts in implementing matrix factorizations using external storage include the POOCLAPACK [67, 110, 66] and SOLAR [121] libraries, as well as an out-of-core extension to ScaLAPACK [18] that, while described in a published paper [34], does not appear in the current version

of the library. Even for these libraries, factorization routines are apparently limited to unpivoted QR and LU decompositions. To our knowledge, there are currently no widely used software options for computing a full rank-revealing factorization out-of-core, nor even publications describing an implementation.

This article describes two implementations for obtaining rank-revealing factorizations of matrices stored out-of-core. The first, `HQRRP_left` is based on the `HQRRP` algorithm of [88], which uses randomization techniques to build a fully blocked column pivoted QR factorization. `HQRRP` relies largely on matrix-matrix operations, reducing the number of reads and writes that must occur between external memory and RAM. `HQRRP_left` further reduces the number of write operations required by reimagining `HQRRP` as a left-looking algorithm. Numerical experiments reveal that this reduction in writing time is critical to the algorithm's performance, particularly when the data is stored on a spinning disk.

The second contribution of this article is an out-of-core implementation of the `randUTV` algorithm of [89], which uses randomization to build a rank-revealing UTV factorization (see, *e.g.* [119, 114] for a good introduction to this factorization). The out-of-core implementation, `randUTV_AB`, modifies `randUTV` in such a way as to achieve overlap of the communication and floating point operations. The result, demonstrated with numerical experiments, is that `randUTV_AB` suffers only a small extra cost by storing the matrix in external memory.

6.1.1 Outline of paper

In Section 6.2, we discuss the first out-of-core factorization, a rank-revealing QR factorization `HQRRP` which can be stopped early to yield a partial decomposition. Section 6.3 explores an out-of-core implementation of `randUTV`, an efficient algorithm for obtaining a rank-revealing orthogonal matrix decomposition. Finally, in Section 6.4 we present numerical experiments which demonstrate the performance of both algorithms.

6.2 Partial rank-revealing QR factorization

In this section, we develop an out-of-core implementation of a rank-revealing column pivoted QR factorization. In section 6.2.1, we review a fully blocked algorithm for computing a column pivoted QR

factorization, HQRRP. In Section 6.2.3, we discuss modifications of HQRRP which enhance its efficiency when the matrix is stored out-of-core.

6.2.1 Overview of HQRRP

Consider an input matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, and assume $m \geq n$. HQRRP [88] is a *blocked* algorithm for computing a column-pivoted QR factorization

$$\begin{array}{ccccc} \mathbf{A} & = & \mathbf{Q} & \mathbf{R} & \mathbf{P}^*, \\ m \times n & & m \times m & m \times n & n \times n \end{array}$$

where $\mathbf{Q}$ is orthogonal, $\mathbf{R}$ is upper trapezoidal, and $\mathbf{P}$ is a permutation matrix.

The bulk of the algorithm's work is executed in a loop with $\lceil n/b \rceil$, where $1 \leq b \leq n$ is the *block size* parameter. For notational simplicity, we assume that b divides n evenly for the remaining discussion, so that $\lceil n/b \rceil = p$ for some $p \in \mathbb{N}$. At the start HQRRP are the initializations

$$\mathbf{R}^{(0)} = \mathbf{A}, \quad \mathbf{Q}^{(0)} = \mathbf{I}, \quad \mathbf{P}^{(0)} = \mathbf{I}.$$

During iteration i, matrices $\mathbf{Q}^{(i)}$ and $\mathbf{P}^{(i)}$ are constructed such that for

$$\mathbf{R}^{(i)} = (\mathbf{Q}^{(i)})^* \mathbf{R}^{(i-1)} \mathbf{P}^{(i)},$$

$\mathbf{R}^{(i)}(:, 1 : ib)$ is upper trapezoidal. After p steps, $\mathbf{R}^{(p)}$ is upper trapezoidal, and the final factorization can be written as

$$\begin{aligned} \mathbf{R} &= \mathbf{R}^{(p)} \\ \mathbf{P} &= \mathbf{P}^{(0)} \mathbf{P}^{(1)} \cdots \mathbf{P}^{(p)} \\ \mathbf{Q} &= \mathbf{Q}^{(0)} \mathbf{Q}^{(1)} \cdots \mathbf{Q}^{(p)}. \end{aligned}$$

6.2.2 Choosing the orthogonal matrices

At step i of HQRRP, consider the partitioning of $\mathbf{R}^{(i)}$

$$\mathbf{R}^{(i)} \rightarrow \begin{bmatrix} \mathbf{R}_{11}^{(i)} & \mathbf{R}_{12}^{(i)} \\ \mathbf{R}_{11}^{(i)} & \mathbf{R}_{22}^{(i)} \end{bmatrix},$$

where $\mathbf{R}^{(i)}$ is $ib \times ib$. The goal of choosing the permutation matrix $\mathbf{P}^{(i)}$ is to find b columns of $\mathbf{R}_{22}^{(i-1)}$ with a large spanning volume, in the sense that the spanning volume is close to maximal. Finding the *best* selection of b columns is currently impossible without trying every combination, but we can find a *good* selection by projecting the columns of $\mathbf{R}_{22}^{(i-1)}$ into a lower dimensional space and cheaply finding b good pivot columns there. The steps for computing $\mathbf{P}^{(i)}$ are thus as follows:

1. Draw a random matrix $\mathbf{G}^{(i)} \in \mathbb{R}^{b \times m-ib}$, with i.i.d. entries drawn from the standard normal distribution.

2. Compute $\mathbf{Y}^{(i)} = \mathbf{G}^{(i)}\mathbf{R}_{22}^{(i-1)}$.

3. Compute b steps of the traditional CPQR of $\mathbf{Y}^{(i)}$ to obtain $\mathbf{Y}^{(i)}\mathbf{P}_{22}^{(i)} = \mathbf{W}_{\text{trash}}\mathbf{S}_{\text{trash}}$.

4. Set
$$\mathbf{P}^{(i)} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{P}_{22}^{(i)} \end{bmatrix}.$$

This method for selecting multiple pivot columns has shown itself effective and reliable, consistently producing factorizations that reveal rank as well as traditional CPQR [88].

Remark 16. *There is an alternate "downdating" method for computing $\mathbf{Y}^{(i)}$ during each step that reduces the asymptotic flop count of the* `HQRRP` *algorithm. With this technique* `HQRRP` *has the same asymptotic flop count as unpivoted QR factorization (though the actual flop count of* `HQRRP` *is unavoidably higher); the reader may see [88] for details. This downdating method will not be used in this article's primary implementation, however, as the communication restrictions imposed by the storage of very large matrices make the basic method of this section more practical.*

Once $\mathbf{P}^{(i)}$ has been computed, $\mathbf{Q}^{(i)}$ is built with well-established techniques:

1. Perform unpivoted QR factorization on $\mathbf{A}_{22}^{(i-1)}\mathbf{P}_{22}(:, 1:b)$ to obtain
$$\mathbf{A}_{22}^{(i-1)}\mathbf{P}_{22} = \mathbf{Q}_{22}^{(i)}\mathbf{A}_{22}^{(i)}(:, 1:b).$$

2. Set

$$\mathbf{Q}^{(i)} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{Q}_{22}^{(i)} \end{bmatrix}.$$

6.2.3 Executing `HQRRP` out of core

The `HQRRP` can be executed for a matrix stored out of core without many modifications. The only requirement is that any time $\mathbf{A}$ is altered, the relevant portion of $\mathbf{A}$ must be read off the hard drive and written back out after operated on. While this is simple in concept, however, the associated communication cost is high. Writing to hard drives, especially drives with spinning disks, is much more expensive than reading from them. It is therefore helpful to reorganize `HQRRP` as a *left-looking algorithm* [109, 110, 121, 79]. Left-looking algorithms differ from their traditional right-looking counterparts in that they work with just the left part of the matrix during each iteration, updating a single block of columns with the information from all previous orthogonal transformations. Left-looking algorithms, then, require only $\mathcal{O}(n^2)$ writes, whereas right-looking algorithms typically require $\mathcal{O}(n^3)$ writes.

A left-looking variant of `HQRRP`, which we shall call `HQRRP_left`, operates as follows. Just as with the original (right-looking) algorithm described in Section 6.2.1, we consider an input matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ with $m \geq n$ and choose a block size b with $1 \leq b \leq n$. We again assume that $n/b = p \in \mathbb{N}$ for notational simplicity. We then make initializations

$$\mathbf{R}^{(0)} = \mathbf{A}, \quad \mathbf{Q}^{(0)} = \mathbf{I}, \quad \mathbf{P}^{(0)} = \mathbf{I}.$$

We also perform the work in a loop with p iterations, just as before. The difference in the left-looking variant lies in the structure of $\mathbf{R}^{(i)}$. At step i, the matrix $\mathbf{R}^{(i-1)}$ is partitioned as

$$\mathbf{R}^{(i-1)} \rightarrow \begin{bmatrix} \mathbf{R}_1^{(i-1)} & \mathbf{R}_2^{(i-1)} & \mathbf{R}_3^{(i-1)} \end{bmatrix},$$

where $\mathbf{R}_1^{(i-1)} \in \mathbb{R}^{m \times ib}$ and $\mathbf{R}_2^{(i-1)} \in \mathbb{R}^{m \times b}$. Then, the following steps are performed:

1. Form the matrix $\mathbf{P}^{(i)}$ with the method described in 6.2.2.

2. Apply the permutation of $\mathbf{P}_{22}^{(i)}$ with

$$\mathbf{R}_P^{(i)} = \begin{bmatrix} \mathbf{R}_2^{(i-1)} & \mathbf{R}_3^{(i-1)} \end{bmatrix} \mathbf{P}_{22}^{(i)}.$$

3. Update $\mathbf{R}_P^{(i)}(:, 1:b)$ with the information of the previous $\mathbf{Q}^k$ transformations:

$$\mathbf{R}_{PQ}^{(i)} = (\mathbf{Q}^{(i-1)})^* \cdots (\mathbf{Q}^{(1)})^* (\mathbf{Q}^{(0)})^* \mathbf{R}_P^{(i)}.$$

4. Build $\mathbf{Q}^{(i)}$ by computing the unpivoted QR factorization of $\mathbf{R}_P^{(i)}(:, 1:b)$, obtaining

$$\mathbf{R}_{PQ}^{(i)} = \mathbf{Q}^{(i)} \mathbf{R}_2^{(i)}.$$

5. Set

$$\mathbf{R}^{(i)} = \begin{bmatrix} \mathbf{R}_1^{(i-1)} & \mathbf{R}_2^{(i)} & \mathbf{R}_P^{(i)}(:, b+1:n) \end{bmatrix}.$$

Observe that in this algorithm, at most $2b$ columns of the matrix $\mathbf{R}^{(i-1)}$ are altered at each step. In practice, each $\mathbf{R}^{(i-1)}$ is overwritten with $\mathbf{R}^{(i)}$, and so the algorithm requires $\mathcal{O}(mp)$ writes to the hard drive.

Remark 17. *`HQRRP_left` is not a completely left-looking algorithm, since the lower-right portion of* $\mathbf{R}^{(i)}$ *must be updated with the information from each* $\mathbf{Q}^{(k)}$ *at each step in order to build* $\mathbf{Y}^{(i)}$*. We simply do not write these results to the drive at each step. As a result, there is an asymptotic increase in the number of both reads and flops performed by `HQRRP_left` as compared to `HQRRP`. Performance testing indicates that the "left-looking" variant still performs better than the original for large matrix sizes due to the high cost of writing to the drive.*

6.3 Full rank-revealing orthogonal factorization

In this section, we develop an efficient implementation of a rank-revealing orthogonal decomposition for a matrix stored out-of-core. In Section 6.3.1, we review an efficient algorithm called `randUTV` for building such a decomposition when the matrices are stored in main memory. In Section 6.3.2, we discuss some modifications of `randUTV` which optimize its efficiency in the out-of-core setting.

6.3.1 Overview of `randUTV`

Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ with $m \leq n$. The algorithm `randUTV` [89] builds a rank-revealing UTV factorization of $\mathbf{A}$, that is, a decomposition

$$\begin{array}{ccccc} \mathbf{A} & = & \mathbf{U} & \mathbf{T} & \mathbf{V}^* \\ m \times n & & m \times m & m \times n & n \times n \end{array}$$

such that $\mathbf{U}$ and $\mathbf{V}$ are orthogonal and $\mathbf{T}$ is upper triangular. `randUTV` is blocked, so it proceeds by choosing a block size parameter b with $1 \leq b \leq n$ and performing its work inside a loop with $\lceil n/b \rceil$ iterations. For ease of notation, we again assume that $n/b = p \in \mathbb{N}$. At the beginning, we initialize,

$$\mathbf{T}^{(0)} = \mathbf{A}, \quad \mathbf{U}^{(0)} = \mathbf{I}, \quad \mathbf{V}^{(0)} = \mathbf{I}.$$

For the matrix $\mathbf{T}^{(i)}$, we will use the partitioning

$$\mathbf{T}^{(i)} \rightarrow \begin{bmatrix} \mathbf{T}_{11}^{(i)} & \mathbf{T}_{12}^{(i)} \\ \mathbf{T}_{21}^{(i)} & \mathbf{T}_{22}^{(i)} \end{bmatrix},$$

where $\mathbf{T}_{11}^{(i)}$ is $ib \times ib$. At iteration i, for $i = 1, \ldots, p$, we form matrices $\mathbf{T}^{(i)}, \mathbf{U}^{(i)}$ and $\mathbf{V}^{(i)}$ as follows:

1. Construct an orthogonal matrix $\hat{\mathbf{V}}_{22}^{(i)}$ such that its leading b columns span approximately the same subspace as the leading b right singular vectors of $\mathbf{T}_{22}^{(i-1)}$.

2. Compute the unpivoted QR factorization of $\mathbf{T}^{(i-1)}\hat{\mathbf{V}}_{22}^{(i)}(:, 1:b)$ to obtain

$$\mathbf{T}^{(i-1)}\hat{\mathbf{V}}_{22}^{(i)}(:, 1:b) = \hat{\mathbf{U}}_{22}^{(i)}\mathbf{S}.$$

3. Compute the SVD of $\left(\hat{\mathbf{U}}_{22}^{(i)}(:, 1:b)\right)^* \mathbf{T}^{(i-1)}\hat{\mathbf{V}}_{22}^{(i)}(:, 1:b)$, yielding

$$\left(\hat{\mathbf{U}}_{22}^{(i)}(:, 1:b)\right)^* \mathbf{T}^{(i-1)}\hat{\mathbf{V}}_{22}^{(i)}(:, 1:b) = \mathbf{U}_{\text{SVD}}\mathbf{D}\mathbf{V}_{\text{SVD}}^*.$$

4. Calculate $\mathbf{V}^{(i)}$ with

$$\mathbf{V}^{(i)} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \hat{\mathbf{V}}_{22}^{(i)} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{V}_{\text{SVD}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{bmatrix}$$

5. Calculate $\mathbf{U}^{(i)}$ with

$$\mathbf{U}^{(i)} = \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \hat{\mathbf{U}}_{22}^{(i)} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{U}_{\text{SVD}} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{bmatrix}$$

6. Calculate $\mathbf{T}^{(i)}$ with

$$\mathbf{T}^{(i)} = (\mathbf{U}^{(i)})^* \mathbf{T}^{(i-1)} \mathbf{V}^{(i)}.$$

Once all $\lceil p/b \rceil$ iterations have completed, we may compute the final factors with

$$\mathbf{T} = \mathbf{T}^{(p)},$$

$$\mathbf{U} = \mathbf{U}^{(0)} \mathbf{U}^{(1)} \cdots \mathbf{U}^{(p)},$$

$$\mathbf{V} = \mathbf{V}^{(0)} \mathbf{V}^{(1)} \cdots \mathbf{V}^{(p)}.$$

This leaves only the question of how the matrix $\hat{\mathbf{V}}_{22}^{(i)}$ is formed in step 1 above. The method, inspired from work in randomized linear algebra including [111, 69, 90], is the following:

1. Draw a random matrix $\mathbf{G}^{(i)} \in \mathbb{R}^{m-ib \times b}$, with i.i.d. entries drawn from the standard normal distribution.

2. Compute the unpivoted QR factorization of $((\mathbf{T}_{22}^{(i-1)})^* \mathbf{T}_{22}^{(i-1)})^q (\mathbf{T}_{22}^{(i-1)})^* \mathbf{G}$, where q is some small nonnegative integer, typically less than three. The result is

$$((\mathbf{T}_{22}^{(i-1)})^* \mathbf{T}_{22}^{(i-1)})^q (\mathbf{T}_{22}^{(i-1)})^* \mathbf{G} = \hat{\mathbf{V}}_{22}^{(i)} \mathbf{R}_{\text{trash}}.$$

This simple algorithm has been demonstrated to consistently provide high quality subspace approximations to the space spanned by the leading b right singular vectors of $\mathbf{T}_{22}^{(i-1)}$. It is largely for this reason that `randUTV` reveals rank comparably to the SVD. For details on `randUTV`, see [89].

6.3.2 Executing `randUTV` out of core

For matrices so large they do not fit in RAM, `randUTV` requires significant management of I/O tasks. If the orthogonal matrices $\mathbf{U}$ and $\mathbf{V}$ are required, then these must be stored out of core as well. To implement

randUTV efficiently under these constraints, it is helpful to reorganize the algorithm as an *algorithm-by-blocks*. Like a blocked algorithm, an algorithm-by-blocks seeks to cast most of the flops in a factorization in terms of the level-3 BLAS. It does so, however, in a way that reduces data synchronization points relative to a blocked algorithm counterpart. This gives an algorithm-by-blocks greater flexibility when scheduling operations to make more efficient use of computing resources.

To redesign randUTV as an algorithm-by-blocks, we begin by considering an unblocked version of the algorithm, that is, randUTV with $b = 1$. We then raise the granularity of the data, considering the matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ into submatrices $\mathbf{A}_{ij}$ of size $m_b \times n_b$ with $1 < m_b, n_b < \min(m, n)$. Then, the unblocked algorithm is re-written with this new conception of $\mathbf{A}$. Any operation must interact with complete blocks $\mathbf{A}_{ij}$ of $\mathbf{A}$ at a time. Additionally, in order to decrease data synchronization points, just one block of any matrix should be altered during any one operation. The resulting algorithm is randUTV as an algorithm-by-blocks, referred to hereafter as randUTV_AB.

randUTV_AB largely performs the same arithmetic operations as randUTV, with the exception that every unpivoted QR factorization must be computed in multiple steps with a scheme based on updating an existing QR factorization (see [67, 105] for details). The key difference is that randUTV_AB has much more flexibility in the ordering of the execution of the operations. The implementation of randUTV_AB is largely accomplished through a runtime system which keeps track of data dependencies and schedules operations during the execution phase. For out of core computations, our experiments showed the best results when the hard drive had only one active read/write request at a time. Thus, the runtime system prioritized reading/writing one block of the stored matrix at a time and devoting processor power from every core to operating on one block at a time. By overlapping floating point operations with I/O, randUTV_AB can be executed with cost similar to randUTV with the matrix stored in RAM. See Chapter 4 for more detail on algorithms-by-blocks, its implementation, and randUTV as an algorithm-by-blocks.

6.4 Numerical results

In this section, we present the experiments demonstrating the scalability and computational costs of implementations of HQRRP and randUTV_AB for matrices stored out of core. In Section 6.4.1, we

compare several implementations of HQRRP with different strategies for handling the I/O. In Section 6.4.2, we examine computational cost of an implementation of randUTV.

6.4.1 Partial CPQR factorization with HQRRP

The machine used for these experiments had four DIMM memory chips with 16 GB of DDR4 memory each. The CPU was an Intel ®Core ™i7-6700K (4.00 GHz) with four cores. Experiments were run on two different hard drives. One was a Toshiba P300 HDD with 3 TB of memory and 7200 RPM; the other was a Samsung V-NAND SSD 950 Pro with 512 GB of memory. The code was compiled with gcc (version 5.4.0) and linked with the Intel ®MKL library (Version 2017.0.3).

The computational cost of three different implementations of HQRRP for matrices stored out of core were tested.

- **Algorithm 1 – In place:** This implementation did not carry out physical permutations on the columns of $\mathbf{A}$ but instead applied the permutation information during each I/O task. In other words, the routine returns $\mathbf{RP}^*$ rather than $\mathbf{R}$. As a result, no more data than necessary is transferred to and from the hard drive, but many reads occur from noncontiguous locations in the drive.

- **Algorithm 2 – Physical pivoting:** Permutations *are* physically performed during the computation, so that upon completion $\mathbf{R}$ is stored in the hard drive. This implementation reads and writes more data than the "in place" version, but the data transfer mostly occurs in contiguous portions of memory.

- **Algorithm 3 – Left-looking:** An implementation of a left-looking version of HQRRP, outlined in Section 6.2.3. This version requires only $\mathcal{O}(n^2)$ write operations, but the total number of operations (including reading, writing, and flops) is asymptotically higher than the "in place" and "physical pivoting" implementations.

Two key observations can be made of the results of testing depicted in Figure 6.1. First, the number of writes required by the implementation has a dramatic effect on its performance, especially when the

matrix is stored on a hard disk drive. Algorithms 2 and 3, while performing reasonably well on the SSD, did not even scale correctly on the HDD, or at the very least did not reach asymptotic behavior as early as the SSD experiments. Algorithm 3, which requires $\mathcal{O}(n^2)$ write operations rather than $\mathcal{O}(n^3)$, outperforms the right-looking alternatives on both hard drives for large matrices. Second, the best-performing algorithm takes roughly 3 times longer than the predicted performance for an in-core factorization of a matrix of the same size on the SSD. On the HDD, it takes about 5.4 times longer. This suggests that there is substantial room for further improvement in computational speeds.

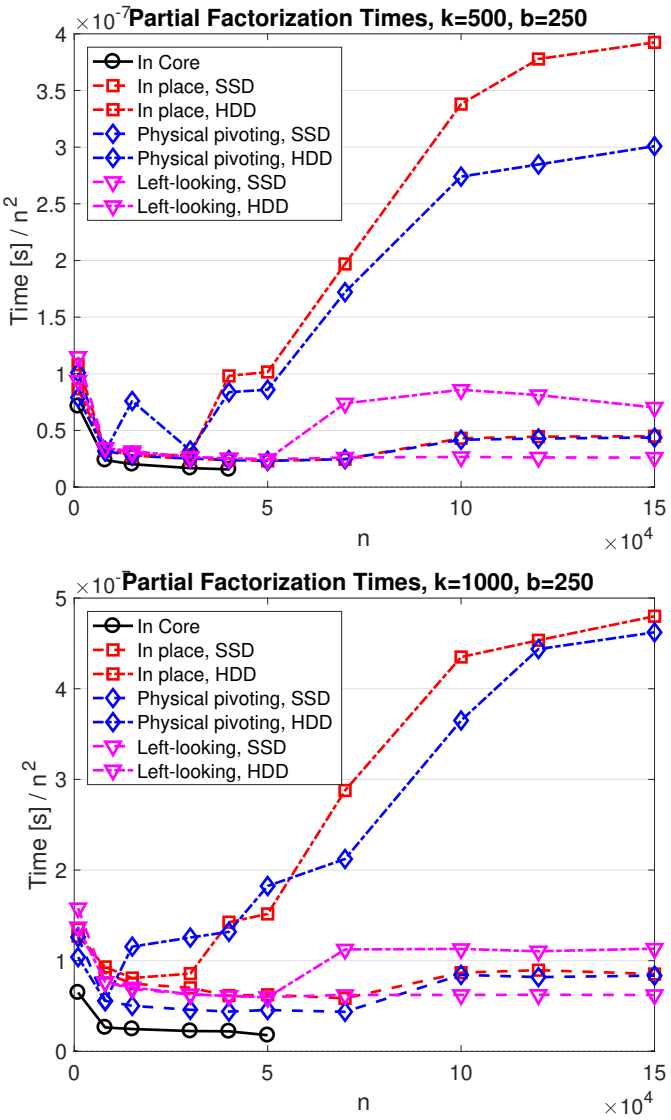


Figure 6.1: A comparison of the computational cost for three different algorithms for computing a partial CPQR when the matrix may be too large to fit in RAM. In the top figure, 1000 columns of the input matrix were processed. In the bottom figure, 500 columns were processed. The block size b in each case was 250.

6.4.2 Full factorization with `randUTV`

The experiments reported in this subsection were performed on an HP computer. The computer contained two Intel Xeon® CPU X5560 processors at 2.8 GHz, with 12 cores and 48 GiB of RAM in total.

Its OS was GNU/Linux (Version 3.10.0-514.21.1.el7.x86_64). Intel?s icc compiler (Version 12.0.0 20101006) was employed. LAPACK routines were taken from the Intel(R) Math Kernel Library (MKL) Version 10.3.0 Product Build 20100927 for Intel(R) 64 architecture, since this library usually delivers much higher performances than LAPACK and ScaLAPACK codes from the Netlib repository. Our implementations were coded with libflame (Release 11104).

We compare computational costs of implementations for several rank-revealing factorizations:

- **In-core SVD, CPQR, QR:** The highly optimized implementations from the Intel MKL of commonly used matrix factorizations. Note that QR is not a rank-revealing factorization, and therefore does not provide as much information as any of the others.
- **In-core `randUTV`:** The implementation of `randUTV_AB` for matrices stored in-core which was discussed in Chapter 4 of this thesis.
- **Out-of-core QR:** The implementation in [105] of unpivoted QR factorization for a matrix stored out-of-core.
- **Out-of-core `randUTV`:** The implementation of the algorithm summarized in Section 6.3.

We note that the OOC `randUTV` is nearly as fast as the predicted performance for the in-core implementation on a matrix of the same size (considering "predicted performance" to be a reasonable extrapolation of the in-core `randUTV` curve). This means that the I/O has been largely overlapped with the required floating point operations, and therefore the cost associated with storing the matrix in main memory is small. Not only is `randUTV_AB` closer to unpivoted QR than CPQR or SVD in terms of speed, but the out-of-core implementation is close to optimal relative to its in-core counterpart.

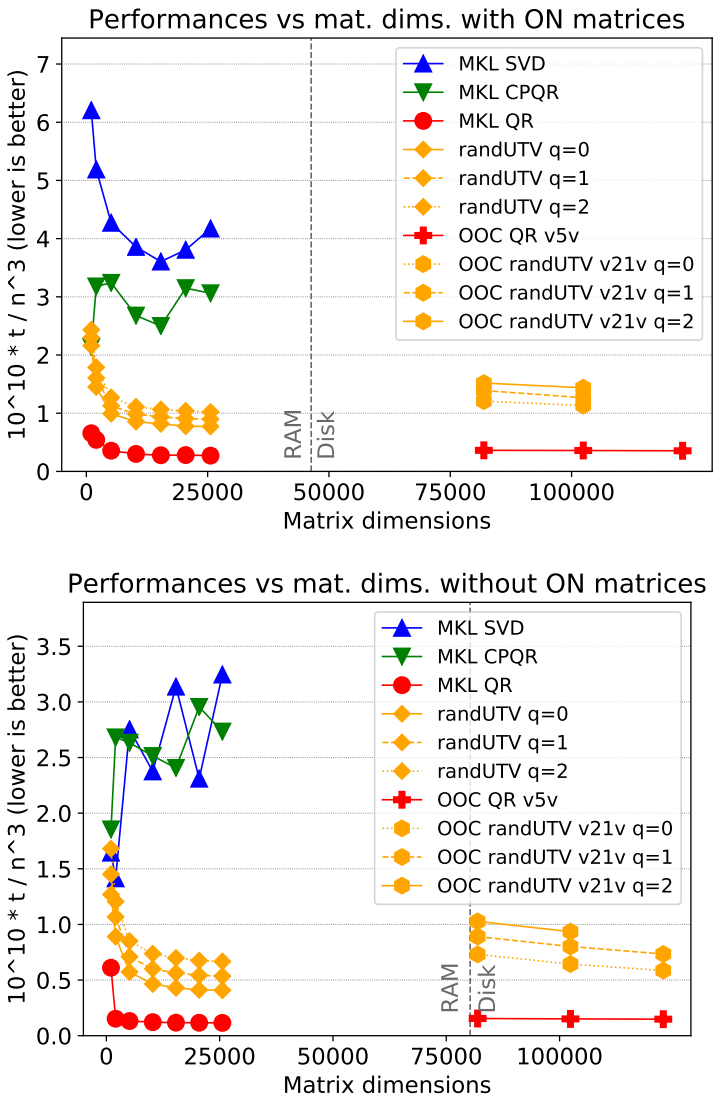


Figure 6.2: Computational costs for the out-of-core implementation of `randUTV_AB`. The block size for I/O operations was 10240, and the block size for carrying out floating point operations was 128. For the in-core implementation of `randUTV_AB`, the block size was 128.

Chapter 7

Efficient nuclear norm approximation via the randomized UTV algorithm

The recently introduced algorithm `randUTV` provides a highly efficient technique for computing accurate approximations to all the singular values of a given matrix $\mathbf{A}$. The original version of `randUTV` was designed to compute a full factorization of the matrix in the form $\mathbf{A} = \mathbf{U}\mathbf{T}\mathbf{V}^*$ where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices, and $\mathbf{T}$ is upper triangular. The estimates to the singular values of $\mathbf{A}$ appear along the diagonal of $\mathbf{T}$. This manuscript describes how the `randUTV` algorithm can be modified when the only quantity of interest being sought is the vector of approximate singular values. The resulting method is particularly effective for computing the nuclear norm of $\mathbf{A}$, or more generally, other Schatten-p norms. The report also describes how to compute an estimate of the errors incurred, at essentially negligible cost.

7.1 Overview

This note describes an efficient algorithm for computing an accurate estimate for the nuclear norm $\|\cdot\|_*$ of a given matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$. The nuclear norm has recently found uses in numerical optimization, beginning with the introduction by Fazel et al. in [47, 48] of its use as an effective heuristic for solving the rank minimization problem

$$\begin{aligned} &\text{minimize} && \operatorname{rank} \mathbf{X} \\ &\text{subject to} && \mathbf{X} \in \mathcal{C} \end{aligned}$$

where $\mathbf{X} \in \mathbb{R}^{m\times n}$ is the decision variable and $\mathcal{C}$ is some given convex constraint set. Recht et al. later proved in [108] that in certain cases, minimizing the nuclear norm also yields the theoretical solution to the

corresponding rank minimization problem, solidifying the validity of the heuristic. The algorithm discussed in this note, `randNN`, may be used in a line search to choose the step size for nuclear norm minimization algorithms such as projected subgradient methods [108] or mirror descent [99, 10].

The recently proposed algorithm `randUTV` [89] is designed to compute, given a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$, a factorization of the form

$$\begin{array}{ccccc} \mathbf{A} & = & \mathbf{U} & \mathbf{T} & \mathbf{V}^*. \\ m\times n & & m\times m & m\times n & n\times n \end{array} \tag{7.1}$$

In (7.1), the matrices $\mathbf{U}$ and $\mathbf{V}$ are orthogonal, $\mathbf{T}$ is upper triangular, and the diagonal entries of $\mathbf{T}$ are good approximations to the singular values of $\mathbf{A}$. This note describes a modified version of `randUTV` which is of particular use in the sub-problem of conducting a line search during nuclear norm minimization. Specifically, we re-derive the `randUTV` algorithm from the perspective of nuclear norm estimation. In this case, since only the singular value estimates for $\mathbf{A}$, rather than the entire matrix factorization, are desired, several steps of `randUTV` may be omitted or modified to yield `randNN`, which sees modest acceleration over `randUTV` and major acceleration over a full (SVD) computation of the singular values. We also mention a rough upper bound on the accuracy of the computed singular values that may computed as a part of `randNN` at very little extra cost from the middle matrix $\mathbf{T}$.

The structure of this note is as follows. In Section 7.2, we review the notation used throughout the remainder. We derive the algorithm `randNN` in 7.3, and in 7.4 we discuss an error bound for the resulting approximations to the singular values. Finally, Section 7.5 contains numerical experiments exploring the accuracy of the singular values estimated by `randNN` and the performance of the algorithm compared to the industry standard.

7.2 Preliminaries

In this note, we use the notation $\mathbf{A} \in \mathbb{R}^{m\times n}$ to denote a matrix $\mathbf{A}$ of dimension $m \times n$ with real entries. $A_{i,j}$ denotes the i,jth entry of $\mathbf{A}$. We use the notation of Golub and Van Loan [57] to specify submatrices: If $\mathbf{A}$ is an $m \times n$ matrix, and $I = [i_1, i_2, \ldots, i_k]$ and $J = [j_1, j_2, \ldots, j_\ell]$ are index vectors, then $\mathbf{A}(I, J)$ denotes the corresponding $k \times \ell$ submatrix. We let $\mathbf{A}(I, :)$ denote the matrix $\mathbf{A}(I, [1, 2, \ldots, n])$, and define

$\mathbf{A}(:, J)$ analogously. The transpose of a matrix $\mathbf{A}$ is denoted $\mathbf{A}^*$, and we use $\sigma_i(\mathbf{A})$ to reference the ith leading singular value of $\mathbf{A}$.

We measure matrices with either the spectral norm $\|\cdot\|$, nuclear norm $\|\cdot\|_*$, or Frobenius norm $\|\cdot\|_F$, with definitions given by

$$\|\mathbf{A}\| = \sup_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\|_2 = \sigma_1(\mathbf{A}), \quad \|\mathbf{A}\|_* = \sum_{i=1}^{\min(m,n)} \sigma_i(\mathbf{A}), \quad \|\mathbf{A}\|_F = \left(\sum_{i=1}^{m} \sum_{j=1}^{n} A_{i,j}^2 \right)^{1/2}$$

(where $\|\cdot\|_2$ is the Euclidean 2-norm for vectors in $\mathbb{R}^n$).

7.3 Description of the algorithm

7.3.1 Approach

Consider a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$. Now partition $\mathbf{A}$ as

$$\mathbf{A} = \begin{array}{c} \\ b \\ m-b \end{array} \begin{array}{c} \begin{array}{cc} b & n-b \end{array} \\ \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} \end{array}.$$

The construction of the algorithm will be guided by several fundamental observations:

- If $\mathbf{A}_{12} = \mathbf{0}$ and $\mathbf{A}_{21} = \mathbf{0}$, then $\|\mathbf{A}\|_* = \|\mathbf{A}_{11}\|_* + \|\mathbf{A}_{22}\|_*$.
- Multiplying $\mathbf{A}$ by an orthogonal matrix does not change the nuclear norm.
- Consider the matrix $\tilde{\mathbf{A}} = \mathbf{U}^*\mathbf{A}\mathbf{V}$, where $\mathbf{U}$ and $\mathbf{V}$ are orthogonal matrices whose first b columns span the same space as the leading b left and right singular vectors of $\mathbf{A}$, respectively. Then giving $\tilde{\mathbf{A}}$ the same partition as $\mathbf{A}$, we have $\tilde{\mathbf{A}}_{12} = \mathbf{0}$ and $\tilde{\mathbf{A}}_{21} = \mathbf{0}$. For this reason, we will call such a $\mathbf{U}$ and $\mathbf{V}$ the "optimal" rotation matrices for the problem.

These remarks lay out a path for estimating $\|\mathbf{A}\|_*$. We will choose b to be a relatively small block size, say 50 or 100, and will compute an orthonormal matrices $\mathbf{U}$ and $\mathbf{V}$ whose first b columns *approximately* span the same space as the respective leading b left and right singular vectors of $\mathbf{A}$. We will then form

$\mathbf{A}^{(1)} = \mathbf{U}^*\mathbf{A}\mathbf{V}$, after which we will have $\mathbf{A}_{21}^{(1)} = \mathbf{0}$ and $\|\mathbf{A}_{12}^{(1)}\|$ is small. At this point, we may estimate the norm by finding the norms of the small $\mathbf{A}_{11}^{(1)}$ block and the large $\mathbf{A}_{22}^{(1)}$ block separately. $\|\mathbf{A}_{11}^{(1)}\|_*$ may be computed efficiently with an SVD computation, and we may recursively apply our strategy for $\mathbf{A}$ to $\mathbf{A}_{22}^{(1)}$ to estimate its norm.

7.3.2 Computing the right orthogonal transformation

At this point in the derivation, the most pressing question is how to find $\mathbf{U}$ and $\mathbf{V}$ that satisfy the aforementioned conditions. We first consider the construction of $\mathbf{V}$, at which point $\mathbf{U}$ will easily follow. Recent work in randomized subspace iteration, discussed in papers including [69, 111, 131, 68], enables the efficient computation of the desired $\mathbf{V}$.

In particular, consider a sampling matrix $\mathbf{Y} = \mathbf{A}^*\mathbf{G}$, where $\mathbf{G}$ is an $m \times b$ Gaussian random matrix. Then with high probability, the column space of $\mathbf{Y}$ often aligns closely to the subspace spanned by the leading b right singular vectors of $\mathbf{A}$. As discussed in [111, 69], in certain situations when the decay of singular values of $\mathbf{A}$ makes the alignment suboptimal, then using $\mathbf{Y} = (\mathbf{A}^*\mathbf{A})^q\mathbf{A}^*\mathbf{G}$ instead provides considerable correction, where q is some small nonnegative integer. Typically, $q = 0, 1, 2$ suffices to obtain a close-to-optimal approximation to the desired subspace.

Thus, the following steps compute a matrix $\mathbf{V}$ whose columns span approximately the same subspace as the leading b right singular vectors of $\mathbf{A}$:

- Draw an $n \times b$ random Gaussian matrix $\mathbf{G}$.
- Compute $\mathbf{Y} = (\mathbf{A}^*\mathbf{A})^q\mathbf{A}^*\mathbf{G}$ for some small nonnegative integer q.
- Compute an orthonormal basis for the column space of $\mathbf{Y}$ with a QR factorization using Householder reflectors to obtain $\mathbf{Y} = \mathbf{V}\mathbf{R}$.

7.3.3 Computing the left orthogonal transformation

To compute $\mathbf{U}$, we first note that if our computed $\mathbf{V}$ were optimal, then the first b columns of $\mathbf{A}\mathbf{V}$ would be composed exclusively of linear combinations of the leading b left singular vectors of $\mathbf{A}$. Thus,

to form $\mathbf{U}$, we may simply compute an orthonormal basis for the first b columns of $\mathbf{AV}$ via Householder reflectors to obtain $\mathbf{AV}(:, 1:b) = \mathbf{UR}$.

Remark 18. *In applications, the input matrix $\mathbf{A}$ is often rank-deficient, with numerical rank k determined by the k for which $\sigma_i(\mathbf{A})$ is less than some specified tolerance for $k < i \leq n$. In the case where $k \ll n$, `randNN` may be sped up substantially by terminating the algorithm once the estimated singular values drop below some user-defined threshold.*

```
function ss = randNN(A,b,q)
  m = size(A,1); n = size(A,2);
  ss = zeros(n,1);
  T = A;
  for i=1:ceil(n/b)
    I1 = (b*(i-1)+1):min((b*i),n);
    I2 = (b*i+1):m;
    J1 = (b*(i-1)+1):min((b*i),n);
    J2 = (b*i+1):n;
    if isempty(J2) == 0
      T_work = T([I1 I2],[J1 J2]);
      [TT,ss_part] = step_nn(T_work,b,q);
      ss(I1) = ss_part;
      T([I1 I2],[J1 J2]) = TT;
    else
      ss_part = svd(T(I1,J1));
      ss(I1) = ss_part;
    end
  end
return
```

```
function [T,ss] = step_nn(A,b,q)
  G = randn(size(A,1),b);
  Y = A'*G;
  for i=1:q
    Y = A'*(A*Y);
  end
  [V,~] = qr(Y);
  T = A*V;
  [U,R] = qr(T(:,1:b));
  T(:,1:b) = R;
  T(:,(b+1):end) = U'* ...
      T(:,(b+1):end);
  ss = svd(R);
return
```

Figure 7.1: Matlab code for the algorithm `randNN` that, given an $m \times n$ matrix $\mathbf{A}$, computes estimates for each of its singular values. The input parameters b and q reflect the block size and the number of steps of power iteration, respectively. This code is simplistic in that it does not store or apply the transformation matrices $\mathbf{U}$ and $\mathbf{V}$ efficiently as products of Householder vectors. It also does not apply a stopping criterion to terminate the algorithm if the estimated singular values become small, nor does it compute the error bound discussed in Section 7.4.

7.4 Error bounds

In this section, we make note of an error bound for the accuracy of the estimated singular values resulting from the `randNN` algorithm.

First, note that after applying $\lceil n/b \rceil$ left and right rotation matrices to $\mathbf{A}$ as described in 7.3, we are left with an upper triangular matrix which we shall call $\mathbf{T}$ whose mass is concentrated in blocks of size $b \times b$ along its diagonal. Therefore, let $\mathbf{T}_d$ be the block diagonal matrix consisting of the $b \times b$ main diagonal blocks of $\mathbf{T}$, and let $\mathbf{T}_u$ be the upper triangular matrix defined by the relation

$$\mathbf{T} = \mathbf{T}_d + \mathbf{T}_u.$$

Observe that the estimated singular values of $\mathbf{A}$ in our algorithm consist precisely of the computed singular values of $\mathbf{T}_d$. Next, since $\mathbf{A}$ and $\mathbf{T}$ only differ by orthogonal transforms, we have that $\|\mathbf{A}\|_* = \|\mathbf{T}\|_*$, and more specifically, $\sigma_i(\mathbf{A}) = \sigma_i(\mathbf{T})$ for $i = 1, 2, \ldots, \min(m, n)$. Thus an error bound given in [117] and [98] states that

$$\sqrt{\sum_{i=1}^{\min(m,n)} \big(\sigma_i(\mathbf{A}) - \sigma_i(\mathbf{T}_d)\big)^2} \leq \|\mathbf{T}_u\|_F. \tag{7.2}$$

Since the calculation of $\|\mathbf{T}_u\|_F$ is $\mathcal{O}(n^2)$, this bound may be calculated at negligible relative cost as part of `randNN`, serving as an assurance of the estimation's validity.

7.5 Numerical experiments

7.5.1 Computational speed

In this section, we investigate the speed of the proposed algorithm `randNN` and compare it to the speed of an exact computation of the singular values using LAPACK's highly optimized `dgesvd`.

All experiments reported in this note were performed on an Intel Core i7-6700K processor (4.0 GHz) with 4 cores. In order to be able to show scalability results, the clock speed was throttled at 4.0 GHz, turning off so-called turbo boost. Other details of interest include that the OS used was Ubuntu (Version 16.04.2), and the code was compiled with Intel's `icc` (Version 17.0.4.196). Main routines for computing the singular values (`dgesvd`) were taken from Intel's MKL library (Version 2017.0.3). Standard BLAS routines used in our implementation of `randNN` were also taken from the Intel MKL library.

Each of the three algorithms we tested (`randNN`,`randUTV`,`SVD`) was applied to double-precision real matrices of size $n \times n$. We report the following times:

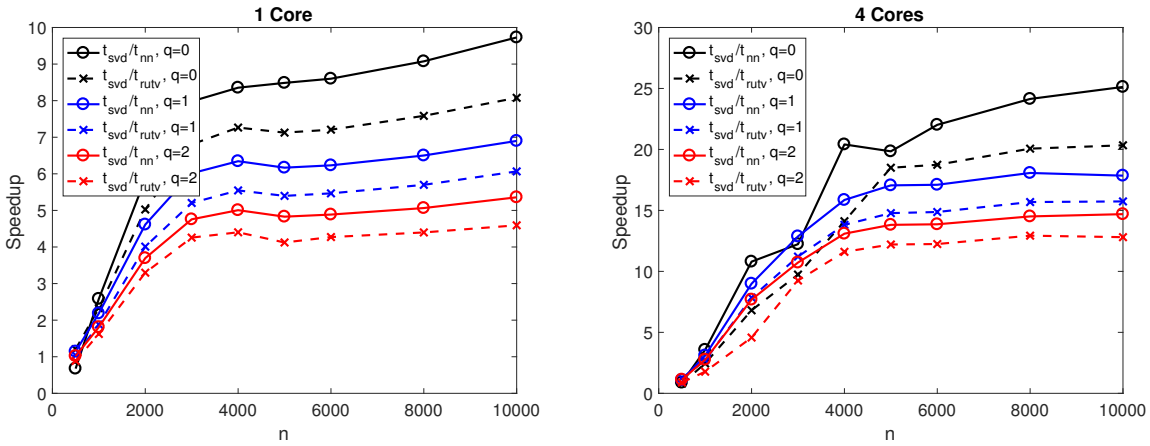


Figure 7.2: Computational speed of `randNN` compared to the speeds of the LAPACK routine `dgesvd` and the recently introduced `randUTV` [89]. The algorithms were applied to double-precision real matrices of size $n \times n$.

- T_{svd}. The time in seconds for the LAPACK function `dgesvd` from Intel's MKL, where the orthogonal matrices **U** and **V** in the SVD were not built.

- T_{randUTV}. The time in seconds for the function `randUTV` described in [89]. The authors' original implementation was used, and the orthogonal matrices **U** and **V** in the UTV decomposition were not built.

- T_{randNN}. The time in seconds for our implementation of `randNN`.

In all cases, we used a block size of $b = 64$. While likely not optimal for all problem sizes, this block size yields near best performance and allows us to easily compare and contrast the performance of the different implementations.

As expected, Figure 7.2 shows that both `randUTV` and `randNN` are markedly faster than LAPACK's `dgesvd`, even in the single case. When more cores are added to the computation, the gain in speed is even more dramatic, hi-lighting the fact that the SVD algorithm is not designed to make efficient use of parallel computing architectures. Finally the increase in speed from `randUTV` to `randNN` is modest, but in a line search setting where the nuclear norm must be computed many times, this saving can add up over the course of the solution of the rank minimization problem.

7.5.2 Errors

In this section, we report the results of numerical experiments that were conducted to test the accuracy of the approximation to the singular values provided by `randNN`. Specifically, we compare the estimated singular values to the true singular values of two different test matrices:

- *Matrix 1 (S shaped decay):* This is an $n \times n$ matrix of the form $\mathbf{A} = \mathbf{UDV}^*$ where $\mathbf{U}$ and $\mathbf{V}$ are randomly drawn matrices with orthonormal columns (obtained by performing QR on a random Gaussian matrix), and where the diagonal entries of $\mathbf{D}$ are chosen to first hover around 1, then decay rapidly, and then level out at 10^{-6}, as shown in Figure 7.3 (black line) on the left.

- *Matrix 2 (Single Layer BIE):* This matrix is the result of discretizing a Boundary Integral Equation (BIE) defined on a smooth closed curve in the plane. To be precise, we discretized the so called "single layer" operator associated with the Laplace equation using a 6^{th} order quadrature rule designed by Alpert [2]. This is a well-known ill-conditioned problem for which column pivoting is essential in order to stably solve the corresponding linear system.

In each case, a matrix size of $m = n = 5000$ was used with a "power iteration parameter" (see Section 7.3.2) of $q = 2$ and a block size of $b = 64$.

For each test matrix, we plot both the estimated and true singular values themselves and the relative error of the estimates

$$\frac{|\sigma_i(\mathbf{A}) - \sigma_i(\mathbf{T}_d)|}{|\sigma_i(\mathbf{A})|}, \quad i = 1, 2, \ldots, \min(m, n).$$

7.6 Availability of code

An implementation of the discussed algorithm is available under 3-clause (modified) BSD license from:

https://github.com/nheavner/nn_code

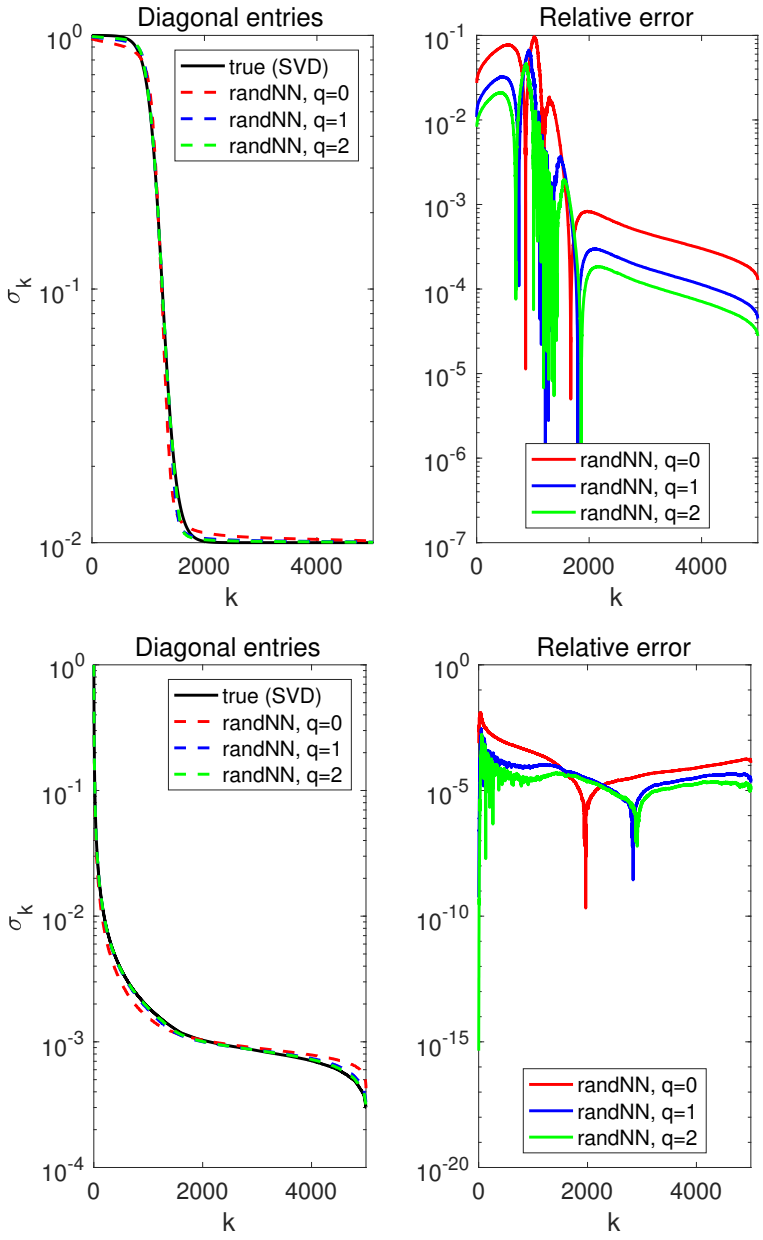


Figure 7.3: Errors for the test matrices described in Section 7.5.2. Left: "Matrix 1," the matrix artificially to have the singular value decay pattern shown here. Right: "Matrix 2," a matrix resulting from the discretization of of Boundary Integral Equation.

Bibliography

[1] Nir Ailon and Bernard Chazelle. Approximate nearest neighbors and the fast Johnson-Lindenstrauss transform. In Proceedings of the thirty-eighth annual ACM symposium on Theory of computing, pages 557–563. ACM, 2006.

[2] Bradley K Alpert. Hybrid Gauss-trapezoidal quadrature rules. SIAM Journal on Scientific Computing, 20(5):1551–1584, 1999.

[3] AMD. AMD Core Math Library. `http://developer.amd.com/tools-and-sdks/cpu-development/amd-core-math-library-acml/`.

[4] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. LAPACK Users' guide (third ed.). SIAM, Philadelphia, PA, USA, 1999.

[5] Ed Anderson, A Benzoni, J Dongarra, S Moulton, S Ostrouchov, Bernard Tourancheau, and Robert van de Geijn. Basic linear algebra communication subprograms. In The Sixth Distributed Memory Computing Conference, 1991. Proceedings, pages 287–290. IEEE, 1991.

[6] Jesse L Barlow. Modification and maintenance of ULV decompositions. In Applied Mathematics and Scientific Computing, pages 31–62. Springer, 2002.

[7] Jesse L Barlow, Peter A Yoon, and Hongyuan Zha. An algorithm and a stability theory for downdating the ULV decomposition. BIT Numerical Mathematics, 36(1):14–40, 1996.

[8] Klaus-Jürgen Bathe. Solution methods for large generalized eigenvalue problems in structural engineering. National Technical Information Service, US Department of Commerce, 1971.

[9] Mario Bebendorf and Sergej Rjasanow. Adaptive low-rank approximation of collocation matrices. Computing, 70(1):1–24, 2003.

[10] Amir Beck and Marc Teboulle. Mirror descent and nonlinear projected subgradient methods for convex optimization. Operations Research Letters, 31(3):167–175, 2003.

[11] Peter Benner, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. Solving linear-quadratic optimal control problems on parallel computers. Optimization Methods and Software, 23(6):879–909, 2008.

[12] Paolo Bientinesi, Enrique S Quintana-Ortí, and Robert A Geijn. Representing linear algebra algorithms in code: the FLAME application program interfaces. ACM Transactions on Mathematical Software (TOMS), 31(1):27–59, 2005.

[13] C. H. Bischof and Gregorio Quintana-Ortí. Algorithm 782: Codes for rank-revealing QR factorizations of dense matrices. ACM Trans. Math. Soft., 24(2):254–257, 1998.

[14] C. H. Bischof and Gregorio Quintana-Ortí. Computing rank-revealing QR factorizations of dense matrices. ACM Trans. Math. Soft., 24(2):226–253, 1998.

[15] Christian Bischof and Charles Van Loan. The WY representation for products of Householder matrices. SIAM J. Sci. Stat. Comput., 8(1):s2–s13, Jan. 1987.

[16] Christian H Bischof and Gautam M Shroff. On updating signal subspaces. IEEE Transactions on Signal Processing, 40(1):96–105, 1992.

[17] Ake Bjorck. Numerical methods for least squares problems, volume 51. Siam, 1996.

[18] L Susan Blackford, Jaeyoung Choi, Andy Cleary, Eduardo D'Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, et al. ScaLAPACK users' guide. SIAM, 1997.

[19] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. An updated set of basic linear algebra subprograms (BLAS). ACM Transactions on Mathematical Software, 28(2):135–151, 2002.

[20] Andre R Brodtkorb, Christopher Dyken, Trond R Hagen, Jon M Hjelmervik, and Olaf O Storaasli. State-of-the-art in heterogeneous computing. Scientific Programming, 18(1):1–33, 2010.

[21] Peter Businger and Gene H Golub. Linear least squares solutions by Householder transformations. Numerische Mathematik, 7(3):269–276, 1965.

[22] Ernie Chan, Enrique S Quintana-Ortí, Gregorio Quintana-Ortí, and Robert Van De Geijn. Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures, pages 116–125. ACM, 2007.

[23] Ernie Chan, Field G Van Zee, Paolo Bientinesi, Enrique S Quintana-Orti, Gregorio Quintana-Orti, and Robert Van de Geijn. Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pages 123–132. ACM, 2008.

[24] Tony F Chan. Rank revealing QR factorizations. Linear algebra and its applications, 88:67–82, 1987.

[25] Tony F Chan and Per Christian Hansen. Computing truncated singular value decomposition least squares solutions by rank revealing QR-factorizations. SIAM Journal on Scientific and Statistical Computing, 11(3):519–530, 1990.

[26] Tony F Chan and Per Christian Hansen. Some applications of the rank revealing QR factorization. SIAM Journal on Scientific and Statistical Computing, 13(3):727–741, 1992.

[27] Shivkumar Chandrasekaran and Ilse CF Ipsen. On rank-revealing factorisations. SIAM Journal on Matrix Analysis and Applications, 15(2):592–622, 1994.

[28] Jaeyoung Choi, James Demmel, Inderjiit Dhillon, Jack Dongarra, Susan Ostrouchov, Antoine Petitet, Ken Stanley, David Walker, and R Clinton Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers – design issues and performance. Computer Physics Communications, 97(1-2):1–15, 1996.

[29] Jaeyoung Choi, Jack J Dongarra, L Susan Ostrouchov, Antoine P Petitet, David W Walker, and R Clint Whaley. Design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. Scientific Programming, 5(3):173–184, 1996.

[30] Jaeyoung Choi, Jack J Dongarra, Roldan Pozo, and David W Walker. Scalapack: A scalable linear algebra library for distributed memory concurrent computers. In Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the, pages 120–127. IEEE, 1992.

[31] Kenneth L Clarkson and David P Woodruff. Low-rank approximation and regression in input sparsity time. Journal of the ACM (JACM), 63(6):54, 2017.

[32] National Research Council et al. Getting up to speed: The future of supercomputing. National Academies Press, 2005.

[33] Jan JM Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. Numerische Mathematik, 36(2):177–195, 1980.

[34] Eduardo D'Azevedo and Jack Dongarra. The design and implementation of the parallel out-of-core ScaLAPACK LU, QR, and Cholesky factorization routines. Concurrency: Practice and Experience, 12(15):1481–1493, 2000.

[35] James Demmel, Ioana Dumitriu, and Olga Holtz. Fast linear algebra is stable. Numerische Mathematik, 108(1):59–91, 2007.

[36] James W Demmel. Applied numerical linear algebra. Siam, 1997.

[37] Jack J Dongarra, Jermey Du Cruz, Sven Hammarling, and Iain S Duff. Algorithm 679: A set of level 3 basic linear algebra subprograms: model implementation and test programs. ACM Transactions on Mathematical Software (TOMS), 16(1):18–28, 1990.

[38] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. A set of level 3 basic linear algebra subprograms. ACM Transactions on Mathematical Software (TOMS), 16(1):1–17, 1990.

[39] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J Hanson. Algorithm 656: an extended set of basic linear algebra subprograms: model implementation and test programs. ACM Transactions on Mathematical Software (TOMS), 14(1):18–32, 1988.

[40] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. Solving Linear Systems on Vector and Shared Memory Computers. SIAM, Philadelphia, PA, 1991.

[41] P. Drineas and M.W. Mahoney. Lectures on randomized linear algebra. In M.W. Mahoney, J.C. Duchi, and A.C. Gilbert, editors, The Mathematics of Data, volume 25, chapter 4, pages 1 – 48. American Mathematical Society, 2018. IAS/Park City Mathematics Series.

[42] Petros Drineas, Ravi Kannan, and Michael W. Mahoney. Fast Monte Carlo algorithms for matrices. II. Computing a low-rank approximation to a matrix. SIAM J. Comput., 36(1):158–183 (electronic), 2006.

[43] J. Duersch and M. Gu. True BLAS-3 performance QRCP using random sampling, 2015. arXiv preprint #1509.06820.

[44] Jed A. Duersch and Ming Gu. Randomized QR with column pivoting. SIAM Journal on Scientific Computing, 39(4):C263–C291, 2017.

[45] Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. Psychometrika, 1(3):211–218, 1936.

[46] Hasan Erbay, Jesse L Barlow, and Zhenyue Zhang. A modified Gram–Schmidt-based downdating technique for ULV decompositions with applications to recursive TLS problems. Computational statistics & data analysis, 41(1):195–209, 2002.

[47] Maryam Fazel. Matrix rank minimization with applications. PhD thesis, PhD thesis, Stanford University, 2002.

[48] Maryam Fazel, Haitham Hindi, and Stephen P Boyd. A rank minimization heuristic with application to minimum order system approximation. In American Control Conference, 2001. Proceedings of the 2001, volume 6, pages 4734–4739. IEEE, 2001.

[49] Ricardo D. Fierro and Per Christian Hansen. Low-rank revealing UTV decompositions. Numerical Algorithms, 15(1):37–55, 1997.

[50] Ricardo D Fierro, Per Christian Hansen, and Peter Søren Kirk Hansen. UTV tools: Matlab templates for rank-revealing UTV decompositions. Numerical Algorithms, 20(2-3):165–194, 1999.

[51] Leslie V Foster. Rank and null space calculations using matrix decomposition without column interchanges. Linear Algebra and its Applications, 74:47–71, 1986.

[52] Alan Frieze, Ravi Kannan, and Santosh Vempala. Fast Monte-Carlo algorithms for finding low-rank approximations. J. ACM, 51(6):1025–1041 (electronic), 2004.

[53] David Geer. Chip makers turn to multicore processors. Computer, 38(5):11–13, 2005.

[54] Gene Golub. Numerical methods for solving linear least squares problems. Numerische Mathematik, 7(3):206–216, 1965.

[55] Gene Golub, Virginia Klema, and Gilbert W Stewart. Rank degeneracy and least squares problems. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1976.

[56] Gene H Golub and Charles F Van Loan. An analysis of the total least squares problem. SIAM journal on numerical analysis, 17(6):883–893, 1980.

[57] Gene H. Golub and Charles F. Van Loan. Matrix computations. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD, third edition, 1996.

[58] Abinand Gopal and Per-Gunnar Martinsson. The powerURV algorithm for computing rank-revealing full factorizations. arXiv preprint arXiv:1812.06007, 2018.

[59] Kazushige Goto and Robert van de Geijn. High-performance implementation of the level-3 BLAS. ACM Trans. Math. Softw., 35(1):4:1–4:14, July 2008.

[60] Kazushige Goto and Robert A. van de Geijn. On reducing TLB misses in matrix multiplication. Technical Report CS-TR-02-55, Department of Computer Sciences, The University of Texas at Austin, 2002.

[61] William B Gragg and Gilbert W Stewart. A stable variant of the secant method for solving nonlinear equations. SIAM Journal on Numerical Analysis, 13(6):889–903, 1976.

[62] M. Gu. Subspace iteration randomization and singular value problems. SIAM Journal on Scientific Computing, 37(3):A1139–A1173, 2015.

[63] Ming Gu and Stanley C Eisenstat. A divide-and-conquer algorithm for the bidiagonal SVD. SIAM Journal on Matrix Analysis and Applications, 16(1):79–92, 1995.

[64] Ming Gu and Stanley C. Eisenstat. Efficient algorithms for computing a strong rank-revealing QR factorization. SIAM J. Sci. Comput., 17(4):848–869, 1996.

[65] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. ACM Trans. Math. Soft., 27(4):422–455, December 2001. Download from `http://www.cs.utexas.edu/users/flame/web/FLAMEPublications.html`http://www.cs.utexas.edu/users/flame/web/FLAMEPublications.html.

[66] Brian C Gunter, Wesley C Reiley, and Robert A van de Geijn. Parallel out-of-core Cholesky and QR factorization with POOCLAPACK. In IPDPS, page 179, 2001.

[67] Brian C Gunter and Robert A Van De Geijn. Parallel out-of-core computation and updating of the QR factorization. ACM Transactions on Mathematical Software (TOMS), 31(1):60–78, 2005.

[68] Nathan Halko, Per-Gunnar Martinsson, Yoel Shkolnisky, and Mark Tygert. An algorithm for the principal component analysis of large data sets. SIAM Journal on Scientific computing, 33(5):2580–2594, 2011.

[69] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. SIAM review, 53(2):217–288, 2011.

[70] Per Christian Hansen and Plamen Y Yalamov. Computing symmetric rank-revealing decompositions via triangular factorization. SIAM Journal on Matrix Analysis and Applications, 23(2):443–458, 2001.

[71] IBM. Engineering and Scientific Subroutine Library. `http://www-03.ibm.com/systems/power/software/essl/`.

[72] Francisco D Igual, Ernie Chan, Enrique S Quintana-Ortí, Gregorio Quintana-Ortí, Robert A Van De Geijn, and Field G Van Zee. The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations. Journal of Parallel and Distributed Computing, 72(9):1134–1143, 2012.

[73] Intel. Math Kernel Library. `http://developer.intel.com/software/products/mkl/`.

[74] Thierry Joffrain, Tze Meng Low, Enrique S. Quintana-Ortí, Robert van de Geijn, and Field G. Van Zee. Accumulating Householder transformations, revisited. ACM Trans. Math. Softw., 32(2):169–179, June 2006.

[75] William B Johnson and Joram Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. Contemporary mathematics, 26(189-206):1, 1984.

[76] William Kahan. Numerical linear algebra. Canadian Mathematical Bulletin, 9(5):757–801, 1966.

[77] Ivars P Kirsteins and Donald W Tufts. Adaptive detection using low rank approximation to a data matrix. IEEE Transactions on Aerospace and Electronic Systems, 30(1):55–67, 1994.

[78] Virginia Klema and Alan Laub. The singular value decomposition: Its computation and some applications. IEEE Transactions on automatic control, 25(2):164–176, 1980.

[79] Jakub Kurzak, Hatem Ltaief, Jack Dongarra, and Rosa M Badia. Scheduling dense linear algebra operations on multicore processors. Concurrency and Computation: Practice and Experience, 22(1):15–44, 2010.

[80] Charles L Lawson and Richard J Hanson. Solving least squares problems, volume 15. Siam, 1995.

[81] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. ACM Transactions on Mathematical Software (TOMS), 5(3):308–323, 1979.

[82] Edo Liberty. Accelerated dense random projections. PhD thesis, Citeseer, 2009.

[83] Edo Liberty, Franco Woolfe, Per-Gunnar Martinsson, Vladimir Rokhlin, and Mark Tygert. Randomized algorithms for the low-rank approximation of matrices. Proceedings of the National Academy of Sciences, 104(51):20167–20172, 2007.

[84] David Luebke. CUDA: Scalable parallel programming for high-performance scientific computing. In 2008 5th IEEE international symposium on biomedical imaging: from nano to macro, pages 836–838. IEEE, 2008.

[85] Michael W Mahoney et al. Randomized algorithms for matrices and data. Foundations and Trends® in Machine Learning, 3(2):123–224, 2011.

[86] P.-G. Martinsson and S. Voronin. A randomized blocked algorithm for efficiently computing rank-revealing factorizations of matrices, mar 2015. ArXiv.org e-print #1503.07157. To appear in the SIAM Journal on Scientific Computing.

[87] Per-Gunnar Martinsson. Randomized methods for matrix computations. In M.W. Mahoney, J.C. Duchi, and A.C. Gilbert, editors, The Mathematics of Data, volume 25, chapter 4, pages 187 – 231. American Mathematical Society, 2018. IAS/Park City Mathematics Series.

[88] Per-Gunnar Martinsson, Gregorio Quintana Ortí, and Nathan Heavner. Householder QR factorization with randomization for column pivoting (HQRRP). SIAM Journal on Scientific Computing, 39(2):C96–C115, 2017.

[89] Per-Gunnar Martinsson, Gregorio Quintana Ortí, Nathan Heavner, and Robert van de Geijn. randUTV: A blocked randomized algorithm for computing a rank-revealing UTV factorization. ACM Transactions on Mathematical Software, 2019.

[90] Per-Gunnar Martinsson, Vladimir Rokhlin, and Mark Tygert. A randomized algorithm for the approximation of matrices. Technical Report Yale CS research report YALEU/DCS/RR-1361, Yale University, Computer Science Department, 2006.

[91] Per-Gunnar Martinsson, Vladimir Rokhlin, and Mark Tygert. A randomized algorithm for the decomposition of matrices. Appl. Comput. Harmon. Anal., 30(1):47–68, 2011.

[92] Per-Gunnar Martinsson, Arthur Szlam, Mark Tygert, et al. Normalized power iterations for the computation of svd. Manuscript., Nov, 2010.

[93] P.G. Martinsson. Blocked rank-revealing qr factorizations: How randomized sampling can be used to avoid single-vector pivoting, 2015. arXiv preprint #1505.08115.

[94] P.G. Martinsson, V. Rokhlin, and M. Tygert. On interpolation and integration in finite-dimensional spaces of bounded functions. Comm. Appl. Math. Comput. Sci, pages 133–142, 2006.

[95] P.G. Martinsson and S. Voronin. A randomized blocked algorithm for efficiently computing rank-revealing factorizations of matrices. SIAM Journal on Scientific Computing, 38(5):S485–S507, 2016.

[96] R. Mathias and G.W. Stewart. A block QR algorithm and the singular value decomposition. Linear Algebra and its Applications, 182:91 – 100, 1993.

[97] Alan Miller. Subset selection in regression. Chapman and Hall/CRC, 2002.

[98] Leon Mirsky. Symmetric gauge functions and unitarily invariant norms. The quarterly journal of mathematics, 11(1):50–59, 1960.

[99] Arkadii Nemirovskii, David Borisovich Yudin, and Edgar Ronald Dawson. Problem complexity and method efficiency in optimization. 1983.

[100] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. 2008.

[101] Christos H Papadimitriou, Hisao Tamaki, Prabhakar Raghavan, and Santosh Vempala. Latent semantic indexing: A probabilistic analysis. In Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, pages 159–168. ACM, 1998.

[102] Haesun Park and Lars Eldén. Downdating the rank-revealing urv decomposition. SIAM Journal on Matrix Analysis and Applications, 16(1):138–155, 1995.

[103] Jack Poulson, Bryan Marker, Robert A Van de Geijn, Jeff R Hammond, and Nichols A Romero. Elemental: A new framework for distributed memory dense matrix computations. ACM Transactions on Mathematical Software (TOMS), 39(2):13, 2013.

[104] Enrique S. Quintana, Gregorio Quintana, Xiaobai Sun, and Robert van de Geijn. A note on parallel matrix inversion. SIAM J. Sci. Comput., 22(5):1762–1771, 2001.

[105] Gregorio Quintana-Ortí, Francisco D Igual, Mercedes Marqués, Enrique S Quintana-Ortí, and Robert A Van de Geijn. A runtime system for programming out-of-core matrix algorithms-by-tiles on multithreaded architectures. ACM Transactions on Mathematical Software (TOMS), 38(4):25, 2012.

[106] Gregorio Quintana-Ortí, Enrique S Quintana-Ortí, Robert A Geijn, Field G Van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. ACM Transactions on Mathematical Software (TOMS), 36(3):14, 2009.

[107] Gregorio Quintana-Ortí, Xiaobai Sun, and Christian H. Bischof. A BLAS-3 version of the QR factorization with column pivoting. SIAM Journal on Scientific Computing, 19(5):1486–1494, 1998.

[108] Benjamin Recht, Maryam Fazel, and Pablo A Parrilo. Guaranteed minimum-rank solutions of linear matrix equations via nuclear norm minimization. SIAM review, 52(3):471–501, 2010.

[109] Wesley C Reiley. Efficient parallel out-of-core implementation of the Cholesky factorization. University of Texas at Austin, Austin, TX, 1999.

[110] Wesley C Reiley and Robert A Van De Geijn. POOCLAPACK: Parallel out-of-core linear algebra package. University of Texas at Austin, Austin, TX, 1999.

[111] Vladimir Rokhlin, Arthur Szlam, and Mark Tygert. A randomized algorithm for principal component analysis. SIAM Journal on Matrix Analysis and Applications, 31(3):1100–1124, 2009.

[112] Támas Sarlós. Improved approximation algorithms for large matrices via random projections. In Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on, pages 143–152. IEEE, 2006.

[113] Robert Schreiber and Charles Van Loan. A storage-efficient WY representation for products of Householder transformations. SIAM J. Sci. Stat. Comput., 10(1):53–57, Jan. 1989.

[114] Gilbert W Stewart. An updating algorithm for subspace tracking. IEEE Transactions on Signal Processing, 40(6):1535–1541, 1992.

[115] Gilbert W Stewart. Updating a rank-revealing ULV decomposition. SIAM Journal on Matrix Analysis and Applications, 14(2):494–499, 1993.

[116] Gilbert W. Stewart. Matrix Algorithms: Basic Decompositions. SIAM, Society for industrial and applied mathematics, 1998.

[117] Gilbert W Stewart. Perturbation theory for the singular value decomposition. Technical report, 1998.

[118] GW Stewart. Rank degeneracy. SIAM Journal on Scientific and Statistical Computing, 5(2):403–413, 1984.

[119] GW Stewart. UTV decompositions. PITMAN RESEARCH NOTES IN MATHEMATICS SERIES, pages 225–225, 1994.

[120] GW Stewart. The QLP approximation to the singular value decomposition. SIAM Journal on Scientific Computing, 20(4):1336–1348, 1999.

[121] Sivan Toledo and Fred G Gustavson. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computations. In IOPADS, volume 96, pages 28–40. Citeseer, 1996.

[122] Lloyd N Trefethen and David Bau III. Numerical linear algebra, volume 50. Siam, 1997.

[123] Sabine Van Huffel and Joos Vandewalle. The total least squares problem: computational aspects and analysis, volume 9. Siam, 1991.

[124] Field G. Van Zee. `libflame`: The Complete Reference. `www.lulu.com`, 2012. Download from `http://www.cs.utexas.edu/users/flame/web/FLAMEPublications.html`http://www.cs.utexas.edu/users/flame/web/FLAMEPublications.html.

[125] Field G. Van Zee, Ernie Chan, Robert van de Geijn, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. The libflame library for dense matrix computations. IEEE Computation in Science & Engineering, 11(6):56–62, 2009.

[126] Field G. Van Zee and Robert A. van de Geijn. BLIS: A framework for rapidly instantiating BLAS functionality. ACM Transactions on Mathematical Software, 41(3), 2015.

[127] Field G Van Zee, Robert A Van de Geijn, and Gregorio Quintana-Ortí. Restructuring the tridiagonal and bidiagonal qr algorithms for performance. ACM Transactions on Mathematical Software (TOMS), 40(3):18, 2014.

[128] S. Voronin and P.G. Martinsson. A CUR factorization algorithm based on the interpolative decomposition. arXiv.org, 1412.8447, 2014.

[129] Rafi Witten and Emmanuel Candes. Randomized algorithms for low-rank matrix factorizations: sharp performance bounds. Algorithmica, 72(1):264–281, 2015.

[130] David P Woodruff et al. Sketching as a tool for numerical linear algebra. Foundations and Trends® in Theoretical Computer Science, 10(1–2):1–157, 2014.

[131] Franco Woolfe, Edo Liberty, Vladimir Rokhlin, and Mark Tygert. A fast randomized algorithm for the approximation of matrices. Applied and Computational Harmonic Analysis, 25(3):335–366, 2008.

[132] Xianyi Zhang, Qian Wang, and Yunquan Zhang. Model-driven level 3 BLAS performance optimization on loongson 3a processor. In Proceedings of the 2012 IEEE 18th International Conference on Parallel and Distributed Systems, pages 684–691. IEEE Computer Society, 2012.