An Approximate Singular Value Decomposition of Large Matrices in Julia

Alexander J. Turner[1,*]

[1]*Harvard University, School of Engineering and Applied Sciences, Cambridge, MA, USA.*

In this project, I implement a parallel approximate singular value decomposition (SVD) in Julia. The approach taken here follows the algorithm described by Friedland et al., [1] and implement it using `AbstractMatrices` and `DArrays` in Julia to give the user additional flexibility. For additional speed, the algorithm makes direct calls to the `DGEMV` routine in the BLAS kernels. An error analysis using (1) random matrices drawn from a uniform distribution, (2) random matrices drawn from a normal distribution, and (3) a real image is performed to quantify the error induced by using the approximate algorithm. The error is found to be less than 6% and the algorithm exhibits good scaling for large matrices. This algorithm is ready to be used by other Julia users and can be found online at `https://github.com/alexjturner/SVDapprox`.

I. INTRODUCTION

Analyzing big data may require finding an approximate representation for the system due to computational limitations. A popular method for approximating large systems is by first factorizing the system with the Singular Value Decomposition (SVD). The SVD decomposes an $m \times n$ matrix $\mathbf{A}$ as,

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{1}$$

where $\mathbf{U}$ is an $m \times m$ matrix with orthonormal columns (referred to as the left-singular vectors), $\mathbf{\Sigma}$ is an $m \times n$ diagonal matrix with monotonically decreasing diagonal entries (referred to as the singular values), and $\mathbf{V}^T$ is an $n \times n$ matrix with orthonormal rows (referred to as the right-singular vectors).

This matrix factorization can be employed in matrix compression by approximating matrix $\mathbf{A}$ as,

$$\mathbf{A} \approx \mathbf{A}_k = \mathbf{U}_k\mathbf{\Sigma}_k\mathbf{V}_k^T \tag{2}$$

*Electronic address: aturner@fas.harvard.edu

where $\mathbf{A}_k$ is a k-rank approximation to $\mathbf{A}$, $\mathbf{U}_k$ is the first k columns of $\mathbf{U}$, $\mathbf{\Sigma}_k$ is the upper left $k \times k$ portion of $\mathbf{\Sigma}$, and $\mathbf{V}_k^T$ is the first k rows of $\mathbf{V}^T$.

Computing the SVD is typically done in a two step process by first reducing to a bidiagonal matrix, $\mathcal{O}(mn^2)$ operations, and then computing the SVD of the bidiagonal matrix through an iterative method, $\mathcal{O}(n)$ operations. However, a recent paper by Friedland et al. [1] presents an approach to compute the k-rank approximation to $\mathbf{A}$ in $\mathcal{O}(kmn)$ operations using an iterative Monte Carlo sampling approach. An important point for this algorithm is that each iteration is guaranteed to improve the approximation of $\mathbf{A}$. The resulting k-rank approximation from this algorithm will be referred to as $\mathbf{B}$,

$$\mathbf{B} \approx \mathbf{A}_k \tag{3}$$

II. SUMMARY OF THE ALGORITHM

The algorithm relies heavily on the Modified Gram-Schmidt Algorithm (MGSA) that I'll briefly describe before detailing the algorithm.

A. Modified Gram-Schmidt Algorithm

Following [1], the Gram-Schmidt process obtains a set of orthogonal and orthonormal vectors $\mathbf{w}_1, \ldots, \mathbf{w}_p$ and $\mathbf{x}_1, \ldots, \mathbf{x}_p$, respectively, from a set of $l \geq p$ vectors $\mathbf{z}_1, \ldots, \mathbf{z}_l$. A Gram-Schmidt process works by projecting the vector $\mathbf{z}$ onto the line spanned by the orthogonal vector $\mathbf{w}$. Fig. 1 shows the first two steps in this process.

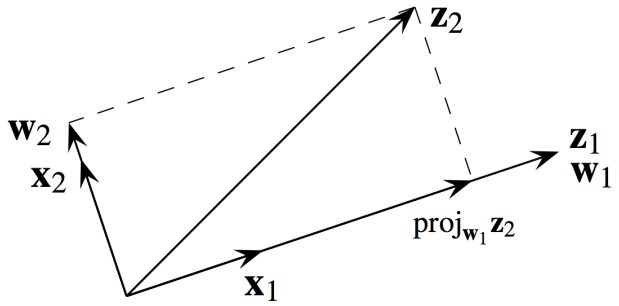

FIG. 1: The first two steps in the Gram-Schmidt process, adapted from [2].

In the Gram-Schmidt process the orthogonal vectors are computed as,

$$\mathbf{w}_k = \mathbf{z}_k - \text{proj}_{\mathbf{w}_1}(\mathbf{z}_k) - \text{proj}_{\mathbf{w}_2}(\mathbf{z}_k) - \cdots - \text{proj}_{\mathbf{w}_1}(\mathbf{z}_{k-1}) \tag{4}$$

however, the classical Gram-Schmidt process is numerically unstable and can be stabilized by a minor modification, yielding the Modified Gram-Schmidt Algorithm,

$$\begin{aligned}
\mathbf{w}_k^{(1)} &= \mathbf{z}_k - \text{proj}_{\mathbf{w}_1}(\mathbf{z}_k), \\
\mathbf{w}_k^{(2)} &= \mathbf{w}_k^{(1)} - \text{proj}_{\mathbf{w}_2}(\mathbf{z}_k^{(1)}), \\
&\vdots \\
\mathbf{w}_k^{(k-2)} &= \mathbf{w}_k^{(k-3)} - \text{proj}_{\mathbf{w}_{k-2}}(\mathbf{z}_k^{(k-3)}), \\
\mathbf{w}_k^{(k-1)} &= \mathbf{w}_k^{(k-2)} - \text{proj}_{\mathbf{w}_{k-1}}(\mathbf{z}_k^{(k-2)}).
\end{aligned} \tag{5}$$

The orthonormal vectors can then be computed as,

$$\mathbf{x}_k = \frac{\mathbf{w}_k}{||\mathbf{w}_k||} \tag{6}$$

The Modified Gram-Schmidt Algorithm requires $\mathcal{O}(nk^2)$ operations.

B. Algorithm

The rest of the algorithm proceeds as follows. The algorithm begins by randomly choosing k columns from $\mathbf{A}$, denoted by $\mathbf{c}_1, \ldots, \mathbf{c}_k$. Then an orthonormal set $\mathbf{x}_1, \ldots, \mathbf{x}_k$ is obtained from $\mathbf{c}_1, \ldots, \mathbf{c}_k$ using the MGSA. This orthonormal set can then be used to compute the first approximation of $\mathbf{B}$ as,

$$\mathbf{B}_0 = \sum_{i=1}^{k} \mathbf{x}_i \left(\mathbf{A}^T \mathbf{x}_i\right)^T \tag{7}$$

This computation requires $\mathcal{O}(kmn)$ operations. However, it should be reasonably straight forward to parallelize this computation through a variety of approaches. Additionally, $\mathbf{B}_0$ can be compactly stored by only storing k pairs of n and m length vectors as,

$$\mathbf{B}_0 = \mathbf{x}_1\mathbf{y}_1^T + \mathbf{x}_2\mathbf{y}_2^T + \ldots + \mathbf{x}_k\mathbf{y}_k^T \tag{8}$$

where $\mathbf{y}_i = \mathbf{A}^T\mathbf{x}_i$.

We can then iteratively improve our approximation of $\mathbf{B}$ by selecting another l columns from $\mathbf{A}$ (preferably that were not chosen before) and updating $\mathbf{B}$ by first defining the $(k+l) \times (k+l)$ matrix $\mathbf{G}$ as,

$$G_{i,j} = \left(\mathbf{A}^T\mathbf{x}_i\right)^T \left(\mathbf{A}^T\mathbf{x}_j\right) \tag{9}$$

a spectral decomposition of $\mathbf{G}$ yields, in $\mathcal{O}((k+l)^3)$ operations, the $\lambda_1 \geq \ldots \geq \lambda_{k+l}$ eigenvalues and $\mathbf{v}_1, \ldots, \mathbf{v}_{k+l}$ orthonormal eigenvectors. We define a $(k+l) \times k$ matrix $\mathbf{O} = [\mathbf{o}_1, \ldots, \mathbf{o}_k]$ as

the eigenvectors corresponding to the k largest eigenvectors of $\mathbf{G}$ and define $\mathbf{U} = [\mathbf{u}_1, \ldots, \mathbf{u}_k] = [\mathbf{x}_1, \ldots, \mathbf{x}_{k+l}]\mathbf{O}$. Columns of $\mathbf{U}$ are then the left-singular vectors for the current approximation of $\mathbf{A}$ and $\sqrt{\lambda_1} \geq \ldots \geq \sqrt{\lambda_k}$ are the corresponding singular values. The right-singular vectors, $\mathbf{V} = [\mathbf{v}_1, \ldots, \mathbf{v}_k]$, can be computed as,

$$\mathbf{v}_i = \frac{\mathbf{A}^T \mathbf{u}_i}{\sqrt{\lambda_i}} \tag{10}$$

Finally, the new $\mathbf{B}$ can be computed as,

$$\mathbf{B}_1 = \sum_{i=1}^{k} \mathbf{u}_i \left(\mathbf{A}^T \mathbf{u}_i\right)^T \tag{11}$$

This process can then be repeated to compute $\mathbf{B}_t$ and $\mathbf{B}_{t-1}$ until,

$$\frac{||\mathbf{B}_{t-1}||}{||\mathbf{B}_t||} > 1 - \epsilon \tag{12}$$

or the algorithm reaches a predetermined number of iterations.

Each update step relies on the computation of the left-singular vectors to compute $\mathbf{B}$, so it implicitly yields the singular vectors and singular values.

C. A Note on Norms

We'd like to keep from explicitly constructing $\mathbf{B}$ and use the compact representation instead. We note that this compact representation is the summation of rank-one outer products $\mathbf{B} = \sum_{i=1}^{k} \mathbf{B}^{(i)} = \sum_{i=1}^{k} \mathbf{x}_i \otimes \mathbf{y}_i$ where $\mathbf{x}_i$ is an m-vector and $\mathbf{y}_i$ is an n-vector. For any n-vector $\mathbf{d}$, we can bound $\left|\left|\mathbf{B}^{(i)}\mathbf{d}\right|\right|$ as,

$$\left|\left|\mathbf{B}^{(i)}\mathbf{d}\right|\right| = \left|\left|\mathbf{x}_i \mathbf{y}_i^T \mathbf{d}\right|\right| = ||\mathbf{x}_i||\ \left|\mathbf{y}_i^T \mathbf{d}\right| \leq ||\mathbf{x}_i||\ ||\mathbf{y}_i||\ ||\mathbf{d}|| \tag{13}$$

Therefore $\left|\left|\mathbf{B}^{(i)}\right|\right| \leq ||\mathbf{x}_i||\ ||\mathbf{y}_i||$. Additionally, we can use the triangle inequality that says,

$$\left|\left|\sum \mathbf{B}^{(i)}\right|\right| \leq \sum \left|\left|\mathbf{B}^{(i)}\right|\right| \tag{14}$$

Putting this all together we have,

$$\sum_{i=1}^{k} ||\mathbf{x}_i||\ ||\mathbf{y}_i|| \leq ||\mathbf{B}|| \tag{15}$$

and we can now compute the exit criteria for iteration as,

$$\frac{\sum_{i=1}^{k} \left|\left|\mathbf{x}_i^{(t-1)}\right|\right|\ \left|\left|\mathbf{y}_i^{(t-1)}\right|\right|}{\sum_{i=1}^{k} \left|\left|\mathbf{x}_i^{(t)}\right|\right|\ \left|\left|\mathbf{y}_i^{(t)}\right|\right|} > 1 - \epsilon \tag{16}$$

without computing the expensive $\mathcal{O}(kmn)$ matrix-vector products.

III. IMPLEMENTATION

The main bottleneck of the code should be computing $\mathcal{O}(kmn)$ matrix-vector products. I implemented my own code profiler to verify that this was indeed the limiting step in the algorithm and maximize my time spent optimizing the code. The code profiler produces output like that shown below:

```
julia> svd_approx(rand(10000,10000))
ITER  0: rand_cols    - 43.20%
ITER  0: run_orth     -  2.44%
ITER  0: compute_B    - 48.75%
ITER  0: compute_norm -  5.61%
ITER  0:   1.65529299 seconds

ITER  1: rand_cols    -  3.82%
ITER  1: run_orth     -  2.32%
ITER  1: compute_G    - 44.73%
ITER  1: eig_G        -  2.10%
ITER  1: svd_G        -  0.33%
ITER  1: compute_B    - 43.40%
ITER  1: compute_norm -  3.29%
ITER  1:   3.21842003 seconds
.
.
.
Exited at iter  8 in 64.67 seconds
```

From this I was able to diagnose which routines were the bottlenecks in the code. Initially, the `compute_norm` routine was taking most of the computational time because I was constructing $\mathbf{B}$ to take the norm. After profiling the code I realized that I needed to find a different way to compute the exit criteria. This prompted the analysis in Section II C.

A. Computing the Matrix-Vector Product

The most expensive step in the algorithm is the k matrix-vector products. I parallelized this section of the code using two different approaches. The first approach is used if the user inputs the matrix $\mathbf{A}$ as an `AbstractMatrix`. This approach assumes that $\mathbf{A}$ is small enough to fit in the memory of a single processor. In this case, we simply share $\mathbf{A}$ across all processors and distribute the $\mathbf{x}_i$'s across the processors and compute p local matrix-vector product using `DGEMV`s. This is

implemented as a map reduce with `DGEMV` as the mapped operator and a horizontal concatenation reduction operator. The schematic for this approach is shown in Fig. 3.

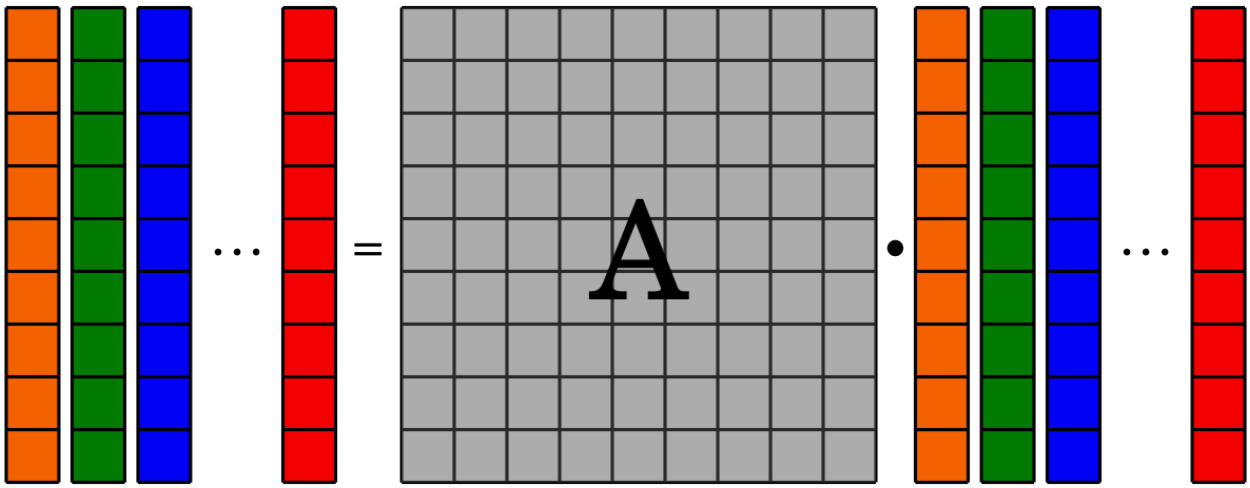

FIG. 2: Schematic of the memory layout for the case using abstract matrices. Gray blocks must be known across all processors and the colored blocks are local to one processor.

The second approach is used if the user inputs the matrix $\mathbf{A}$ as an `DArray`. This approach assumes that $\mathbf{A}$ is too large to store in memory on a single processor and must be distributed across multiple machines. We can no longer just compute a local matrix-vector product because it would require passing large amounts of data between the processors because each processor would need to know all elements of $\mathbf{A}$. Instead we leave the original distribution of $\mathbf{A}$ and simply distribute the $\mathbf{x}_i$ across all processors. We then compute pieces of the matrix-vector product and perform a reduction across all processors. For this approach we compute k matrix-vector products in serial but break each matrix-vector product up into smaller components. It is implemented as a map reduce with `DGEMV` as the mapped operator and an addition operator for the reduction. The schematic for this approach is shown in Fig. 3.

The first approach is simpler, should allow for more parallelization, and should be faster for small matrices, because of this we use it as the default unless the user supplies a `DArray`.

IV. ERROR ANALYSIS

We can compute the error induced by using this approximate algorithm as,

$$\epsilon_{\text{approx}} = \frac{||\mathbf{A} - \mathbf{B}||}{||\mathbf{A}||} \tag{17}$$

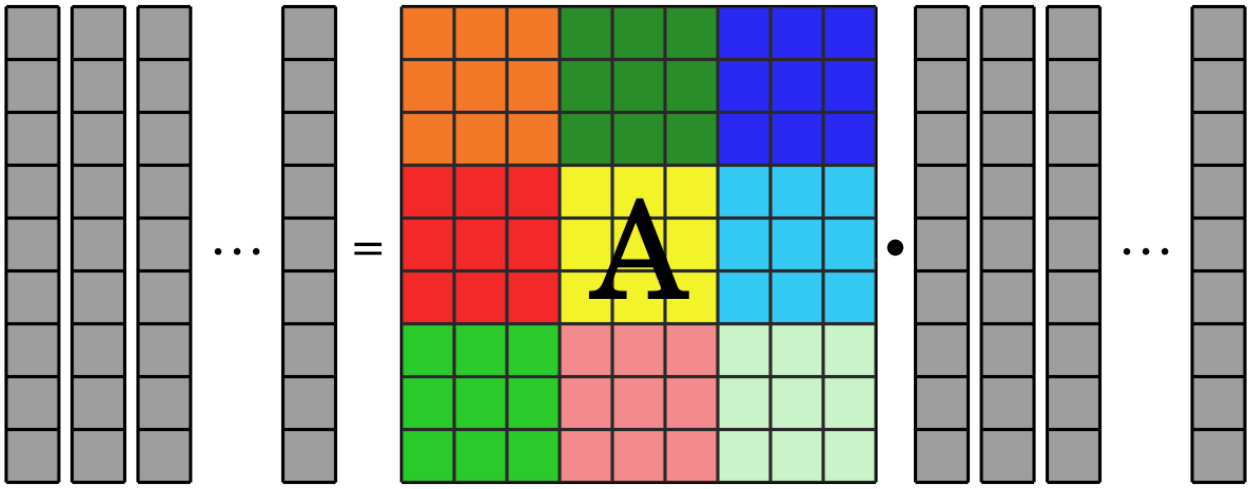

FIG. 3: Schematic of the memory layout for the case using distributed arrays. Gray blocks must be known across all processors and the colored blocks are local to one processor.

however there is an error induced by doing a k-rank approximation of $\mathbf{A}$ even with the true SVD,

$$\epsilon_{\text{true}} = \frac{||\mathbf{A} - \mathbf{A}_k||}{||\mathbf{A}||} \tag{18}$$

So we instead look at the additional error induced by doing this approximation,

$$\epsilon = \left| \frac{\epsilon_{\text{true}} - \epsilon_{\text{approx}}}{\epsilon_{\text{true}}} \right| \tag{19}$$

This was tested for two different types of matrices: (1) random matrices drawn from a uniform distribution using `rand` and (2) random matrices drawn from a normal distribution using `randn`. These two distributions were chosen for their different eigenvalue spectrum. The former will exhibit a sharp drop off in eigenvalues while the latter will exhibit a much smoother decay in eigenvalues.

Fig. 4 shows the error as a function of rank for the first case and Fig. 5 shows the error for the second case. We can also see the different eigenvalue spectrums for these two cases. The algorithm does slightly better for the second case by this metric. However, both matrices compare quite well with less than 6% error induced.

I performed an additional test using a real image (see Fig. 6). The eigenvalue spectrum for the real image is similar to that of the random matrix drawn from a uniform distribution but has a slightly smoother transition. We find that the approximate SVD captures most of the features shown in the true SVD.

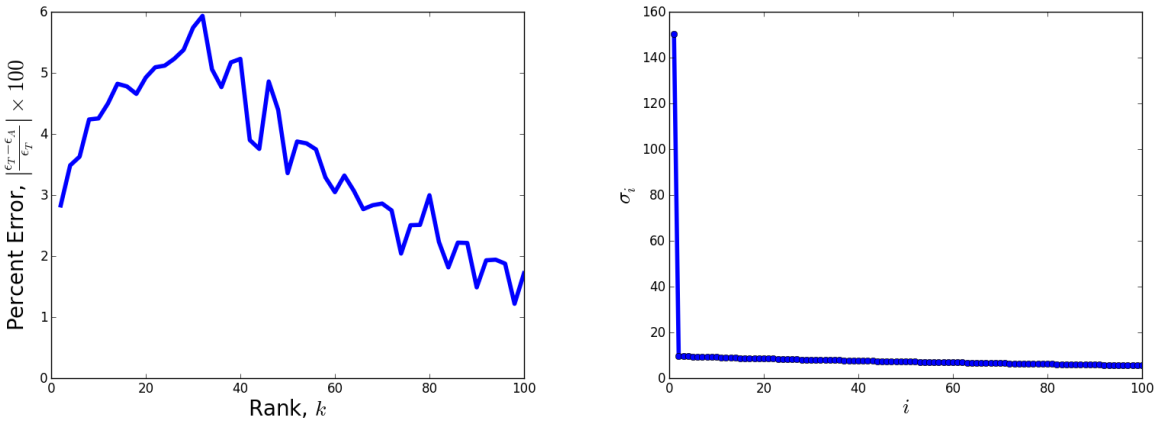

FIG. 4: (Left) The percent error induced by using the approximate SVD and (Right) the eigenvalue spectrum for the random matrix drawn from a uniform distribution.

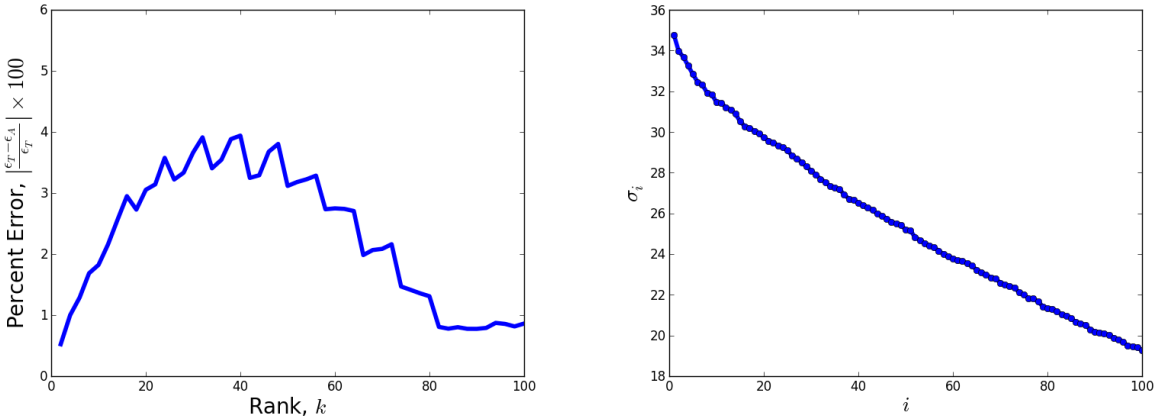

FIG. 5: (Left) The percent error induced by using the approximate SVD and (Right) the eigenvalue spectrum for the random matrix drawn from a normal distribution.

V. SPEEDUP

We test the two implementations of the algorithm with random matrices drawn from a uniform distribution. The tests use the default parameters set for the algorithm. Fig. 8 shows the wall time as a function of matrix size for the two implementations of the approximate algorithm as well as the true SVD. We see that both of the approximate implementations are slower for small ($< 10^7$ elements) matrices but exhibit much better scaling, indicated by both implementations beating the true SVD for matrices larger than 10^8 elements $[10,000 \times 10,000]$. The implementation using `AbstractMatrices` was faster than the implementation using `DArrays`, however only the `DArray` implementation can accommodate matrices too large to store on a single machine.

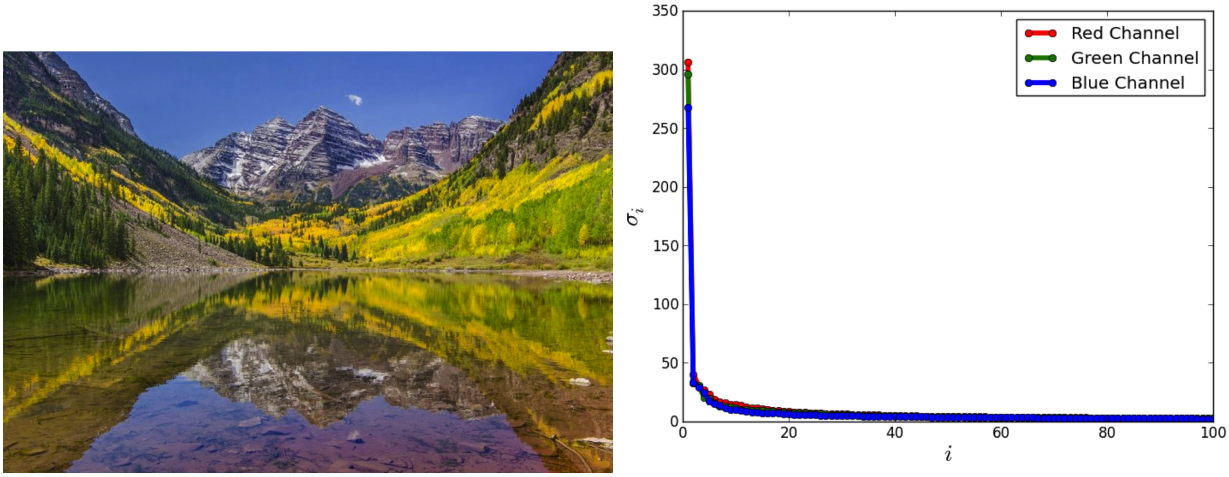

FIG. 6: (Left) The image used for the SVD test [800 × 542]. (Right) The RGB eigenvalue spectrum for the image.

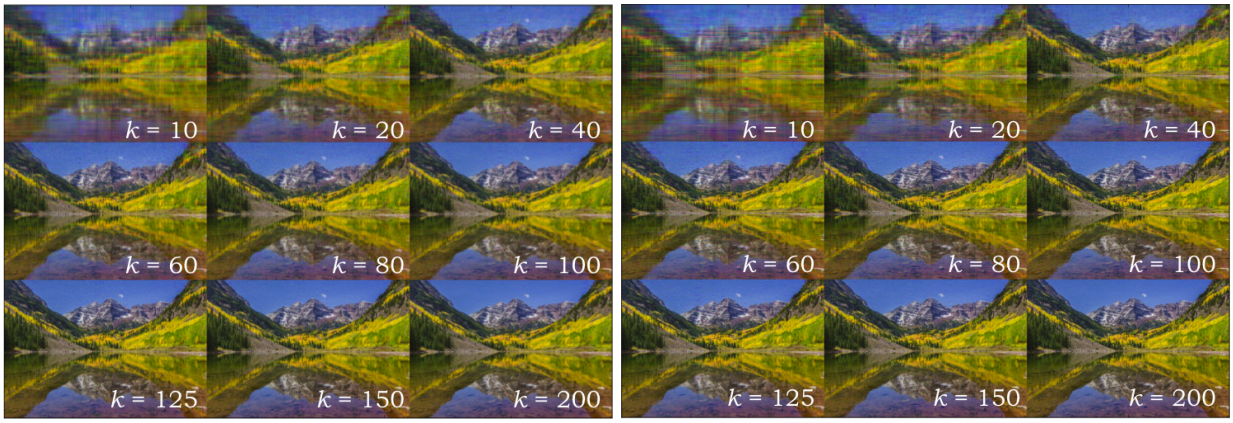

FIG. 7: (Left) The approximated image using the true SVD, $\mathbf{A}_k$. (Right) The approximated image using the approximate SVD, $\mathbf{B}$.

VI. CONCLUSIONS

I implemented an approximate singular value decomposition in Julia following the algorithm described by Friedland et al., [1]. The algorithm only induced small ($< 6\%$) errors and exhibits good scaling for large ($< 10^7$ elements) matrices. Additionally, the algorithm can be performed even if the matrix is too large to exist on a single machine. The code is available online through

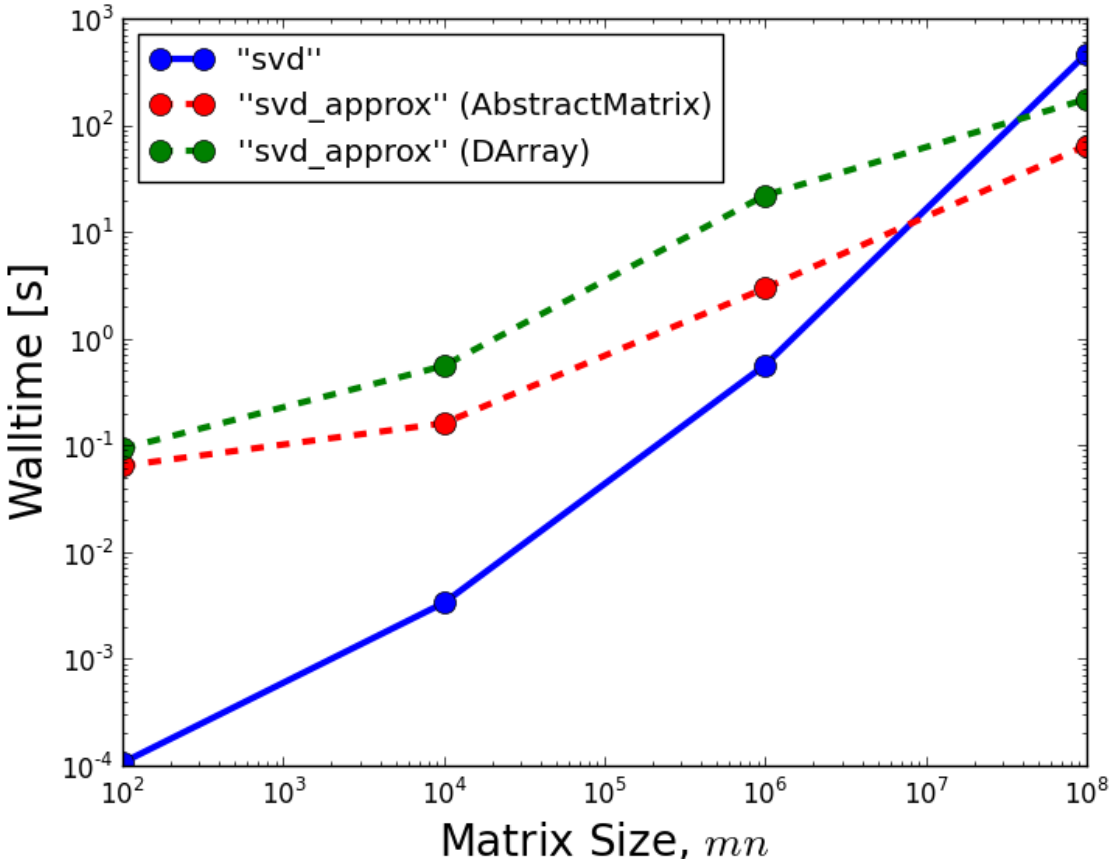

FIG. 8: Wall time for the true SVD (blue), approximate SVD with shared matrices (red), and the approximate SVD with distributed arrays (green). All tests were performed using 3.4GHz quad-core i5 processors (4 physical cores and 4 hyper threaded cores) and 32Gb of RAM.

github here: `https://github.com/alexjturner/SVDapprox`.

---

[1] S. Friedland, A. Niknejad, M. Kaveh, and H. Zare, "Fast monte-carlo low rank approximations for matrices," *Proc. IEEE SoSE*, pp. 218–223, 2006.

[2] Wikipedia, "Gram-schmidt process," *http://en.wikipedia.org/wiki/Gram-Schmidt_process*, 2013.