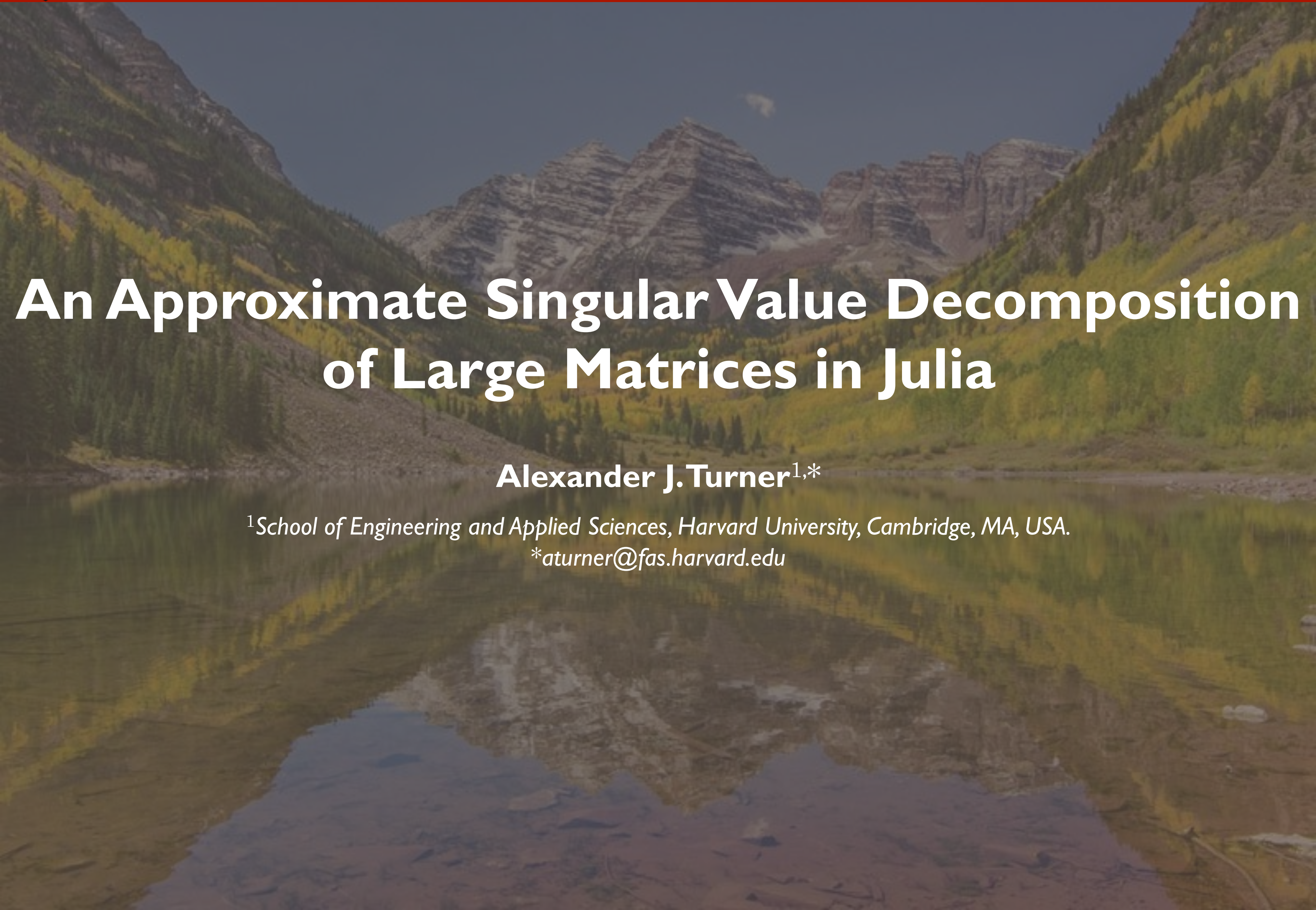


An Approximate Singular Value Decomposition of Large Matrices in Julia

Alexander J. Turner[1,*]

[1]*School of Engineering and Applied Sciences, Harvard University, Cambridge, MA, USA.*
[*]*aturner@fas.harvard.edu*

Motivation behind an Approximate SVD

- The singular value decomposition (SVD) is a widely used algorithm:
  - Data compression: allows for compact representation of matrices
  - Data assimilation: determine fastest growing perturbations
- Does not scale well: $\mathrm{O}(n^3)$ for a square matrix

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

- Approximate algorithm based on Friedland et al., (2009)

$$\mathbf{B} \approx \mathbf{A}_k = \mathbf{U}_k\boldsymbol{\Sigma}_k\mathbf{V}_k^T$$

- Two attractive features:
  - Better scaling: $\mathrm{O}(kn^2)$ for a square matrix
  - Small memory footprint

Description of the Algorithm

Psuedocode

```
1  C          = rand_cols(A,k)
2  X          = run_orth(C)
3  (Bx,By)    = compute_B(X,A,k)
4  N          = compute_norm(Bx,By,k)
5  while iter == true
6      C          = rand_cols(A,ℓ)
7      X          = run_orth(hcat(X,C))
8      G          = compute_G(X,A,k+ℓ)
9      (O,λ)      = eig_G(G,k,ℓ)
10     (U,S)      = svd_G(X,O,λ)
11     (Bx,By)    = compute_B(U,A,k)
12     N          = compute_norm(Bx,By,k)
13 end
```

Explanation

1 Randomly draw “*k*” columns from “A”
2 Obtain an orthonormal set from the columns
3 Construct the “B” matrix using the ⊥ set
4 Compute the norm of “B”
5 Begin iterating
6   Randomly draw “ℓ” more columns
7   Obtain a new ⊥ set
8   Construct the “G” matrix using the ⊥ set
9   Compute the eigenvectors/values of “G”
10   Compute the SVD of “G”
11   Compute the “B” using the SVD of “G”
12   Compute the norm of “B”
13 Iterate

- Iteratively sample **A** and obtain orthonormal sets
  - Uses QR factorization to obtain orthonormal set

Description of the Algorithm

Psuedocode

```
1  C          = rand_cols(A,k)
2  X          = run_orth(C)
3  (Bx,By)    = compute_B(X,A,k)
4  N          = compute_norm(Bx,By,k)
5  while iter == true
6      C          = rand_cols(A,ℓ)
7      X          = run_orth(hcat(X,C))
8      G          = compute_G(X,A,k+ℓ)
9      (O,λ)      = eig_G(G,k,ℓ)
10     (U,S)      = svd_G(X,O,λ)
11     (Bx,By)    = compute_B(U,A,k)
12     N          = compute_norm(Bx,By,k)
13 end
```

Explanation

1 Randomly draw “k” columns from “A”
2 Obtain an orthonormal set from the columns
3 Construct the “B” matrix using the ⊥ set
4 Compute the norm of “B”
5 Begin iterating
6   Randomly draw “ℓ” more columns
7   Obtain a new ⊥ set
8   Construct the “G” matrix using the ⊥ set
9   Compute the eigenvectors/values of “G”
10   Compute the SVD of “G”
11   Compute the “B” using the SVD of “G”
12   Compute the norm of “B”
13 Iterate

- Apply orthonormal sets to the **A** matrix
  - Main bottleneck: $O(kmn)$ complexity

Implementation and Validation of the Algorithm

- Takes either `AbstractMatrices` or `DArrays`
  - Algorithm proceeds differently depending on the array type

- Compactly store matrices as,

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

- k pairs of $[m \times 1]$, $[n \times 1]$ vectors instead of an $[m \times n]$ matrix
  - $k(m+n)$ elements instead of mn elements

- Never actually construct the full $\mathbf{B}$

Implementation and Validation of the Algorithm

- **Two validation cases:** "`rand(N`$_x$`,N`$_y$`)`" & "`randn(N`$_x$`,N`$_y$`)`"

- True error: $$\epsilon_T = \frac{||\mathbf{A} - \mathbf{A}_k||_F}{||\mathbf{A}||_F}$$

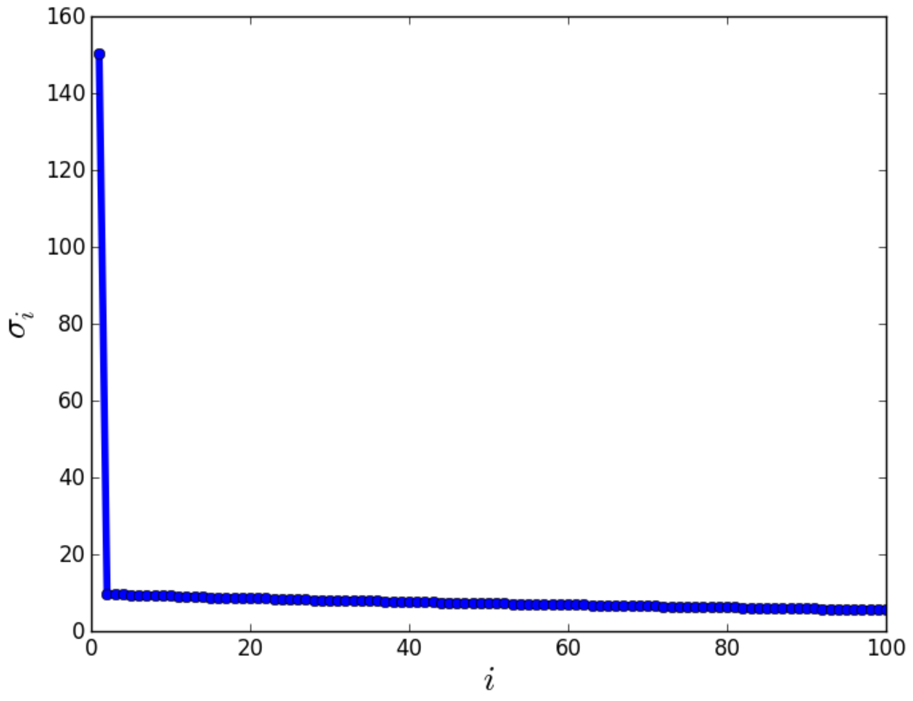
- Approximation error: $$\epsilon_A = \frac{||\mathbf{A} - \mathbf{B}||_F}{||\mathbf{A}||_F}$$

- Error added from approximation: $$\epsilon = \left| \frac{\epsilon_T - \epsilon_A}{\epsilon_T} \right| \times 100$$

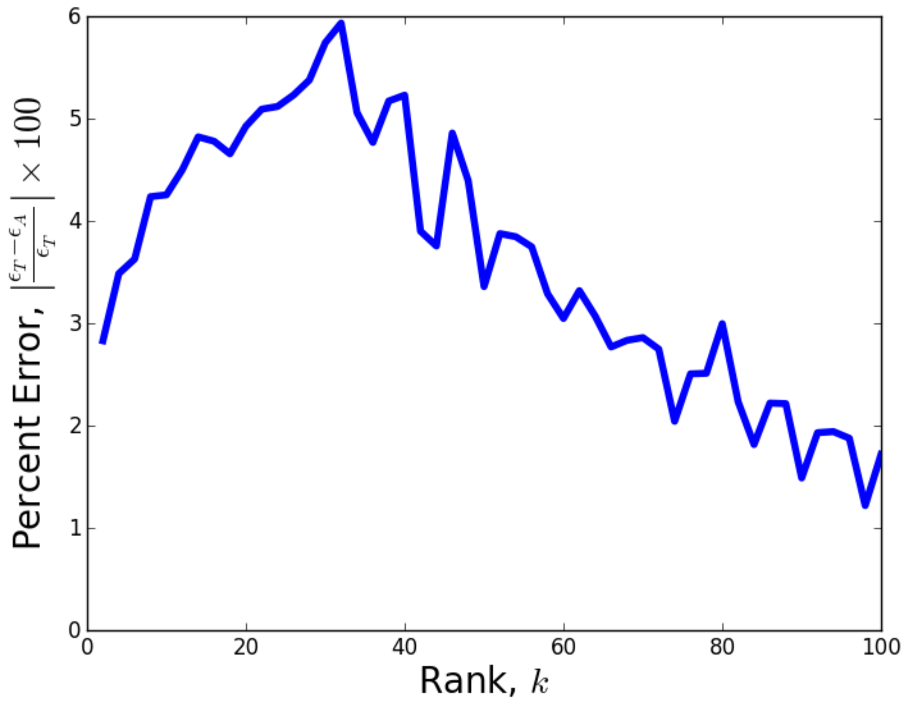
Implementation and Validation of the Algorithm

- "`rand(N`$_x$`,N`$_y$`)`" case
  - Sharp dropoff in the eigenvalue spectrum
  - $N_x, N_y = 300$

Singular Values



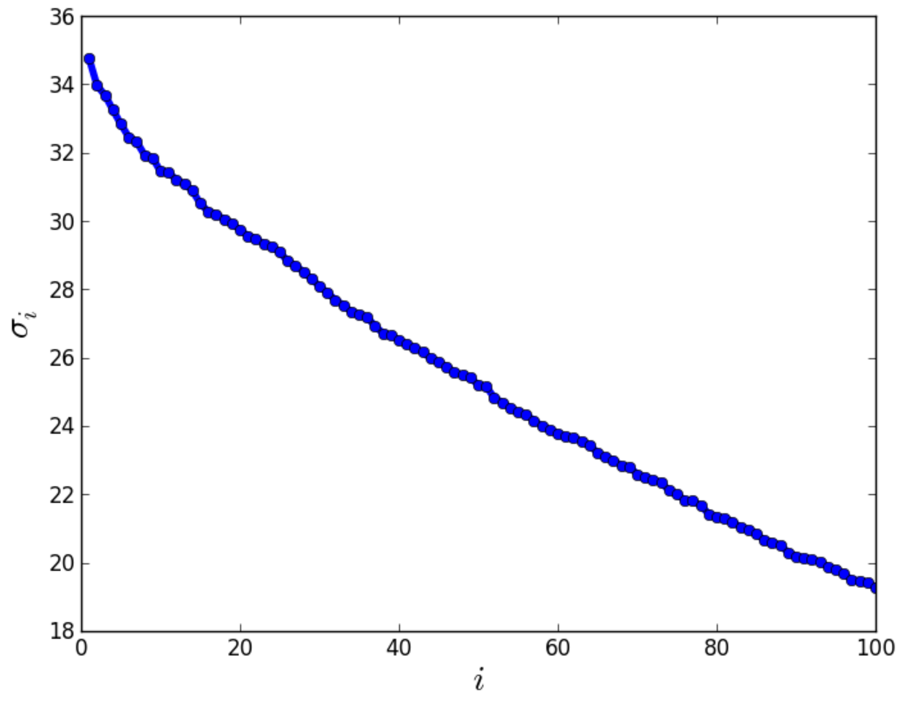
Error



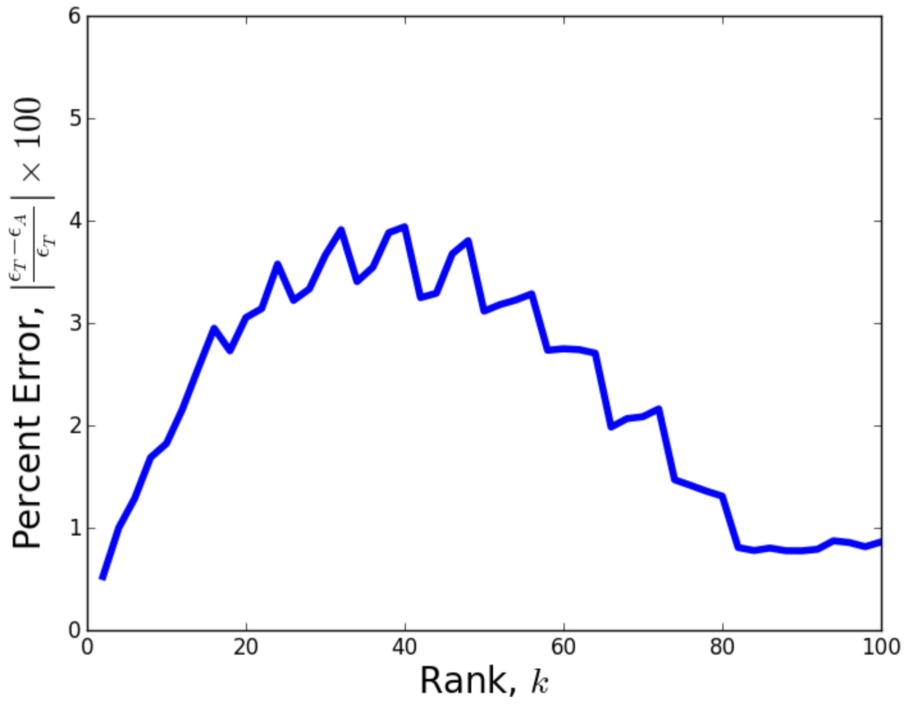
Implementation and Validation of the Algorithm

- "`randn(N`$_x$`,N`$_y$`)`" case
  - Smooth dropoff in the eigenvalue spectrum
  - $N_x, N_y = 300$

Singular Values

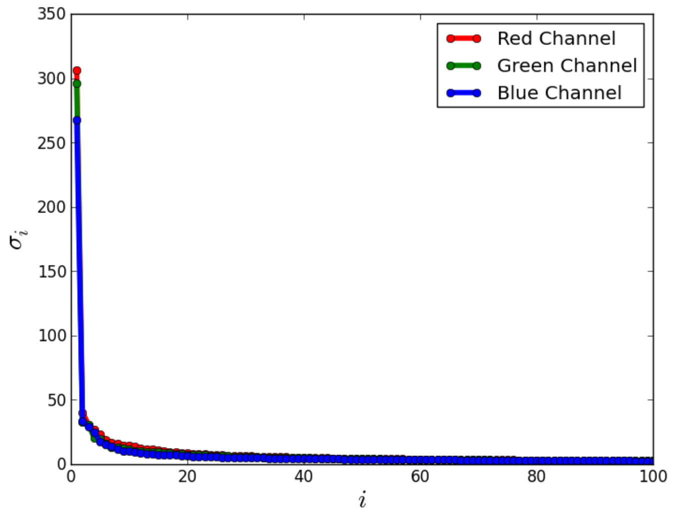


Error

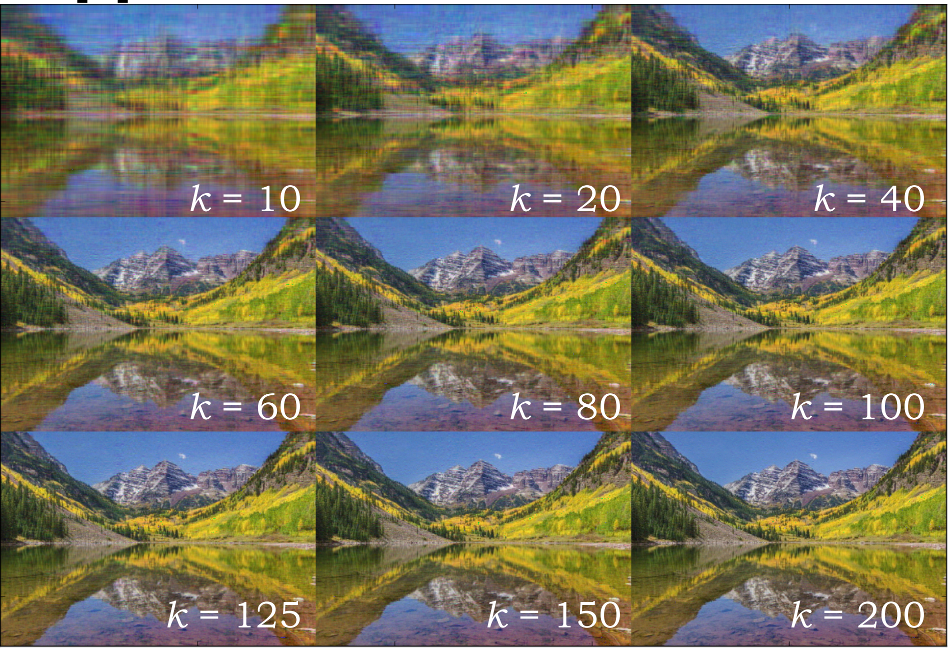


Implementation and Validation of the Algorithm

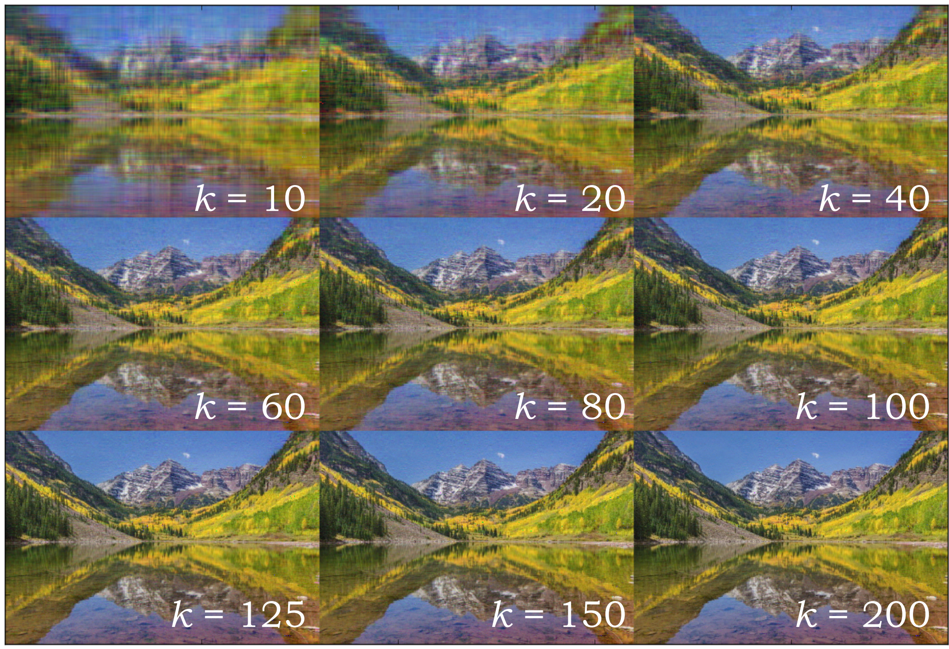
- Testing with a real image
- Fairly sharp eigenvalue dropoff
  - Original image is [800×542]



Approximated



True



Profiling the Code

- Developed a code profiler
- Allowed me to quickly determine bottlenecks
  - Helped me choose the norm
- Helped me choose default parameters

```
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
```

Optimizations

- Direct calls to the BLAS/LAPACK
  - QR factorization (**`DGEQRF + DORGQR`**)
  - Matrix-Vector multiplication (**`DGEMV`**)

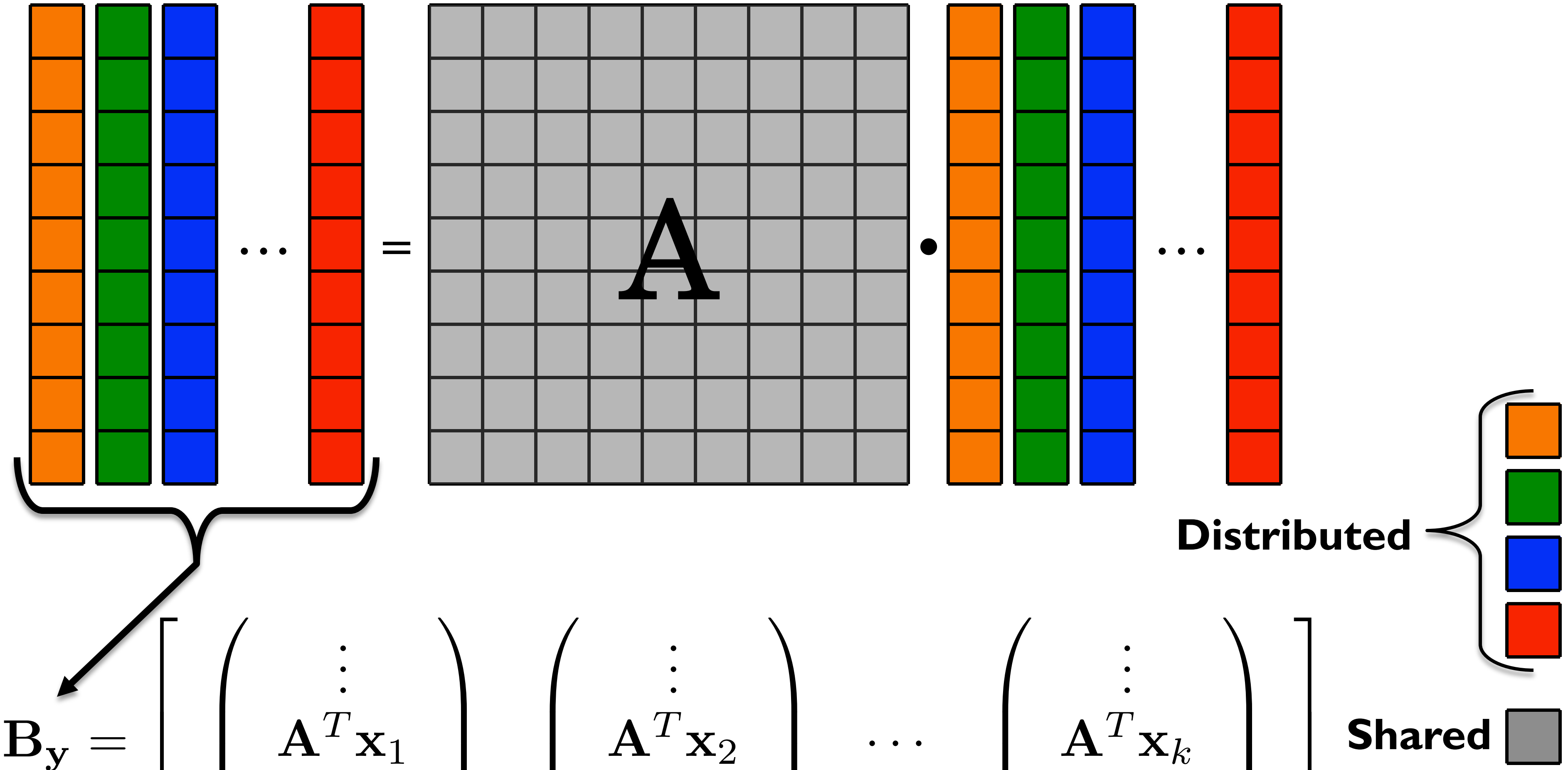
- Parallel matrix multiplication
  - Need to perform k matrix-vector multiplications at every iteration:

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

  - Can do them in simultaneously in parallel or can break them into smaller matrix-vector multiplications

Optimizations: A Tale of Two DGEMVs

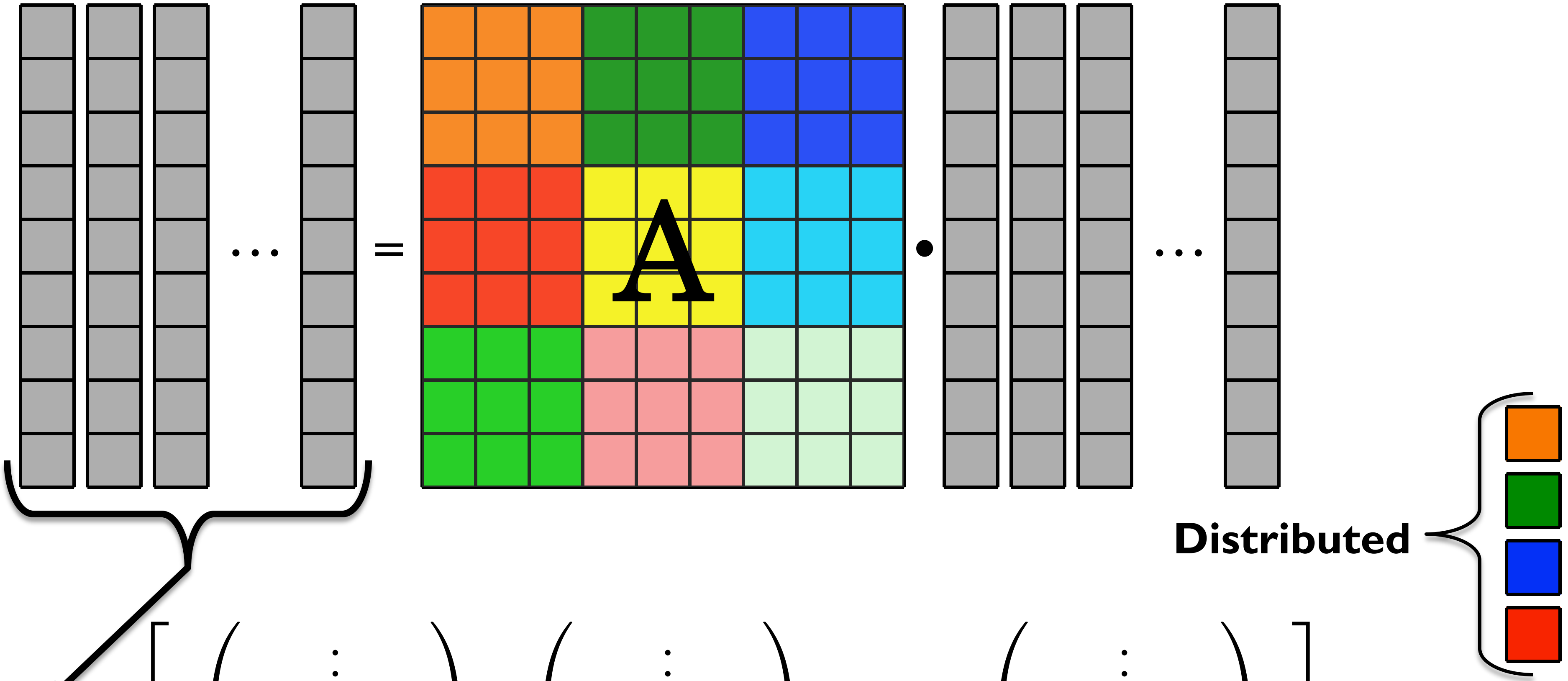
Shared Memory: k large DGEMVs in parallel on p processors



$$\mathbf{B_y} = \left[\begin{pmatrix} \vdots \\ \mathbf{A}^T\mathbf{x}_1 \\ \vdots \end{pmatrix} \begin{pmatrix} \vdots \\ \mathbf{A}^T\mathbf{x}_2 \\ \vdots \end{pmatrix} \cdots \begin{pmatrix} \vdots \\ \mathbf{A}^T\mathbf{x}_k \\ \vdots \end{pmatrix} \right]$$

Optimizations: A Tale of Two DGEMVs

Distributed Memory: A broken into p parts, k serial DGEMVs

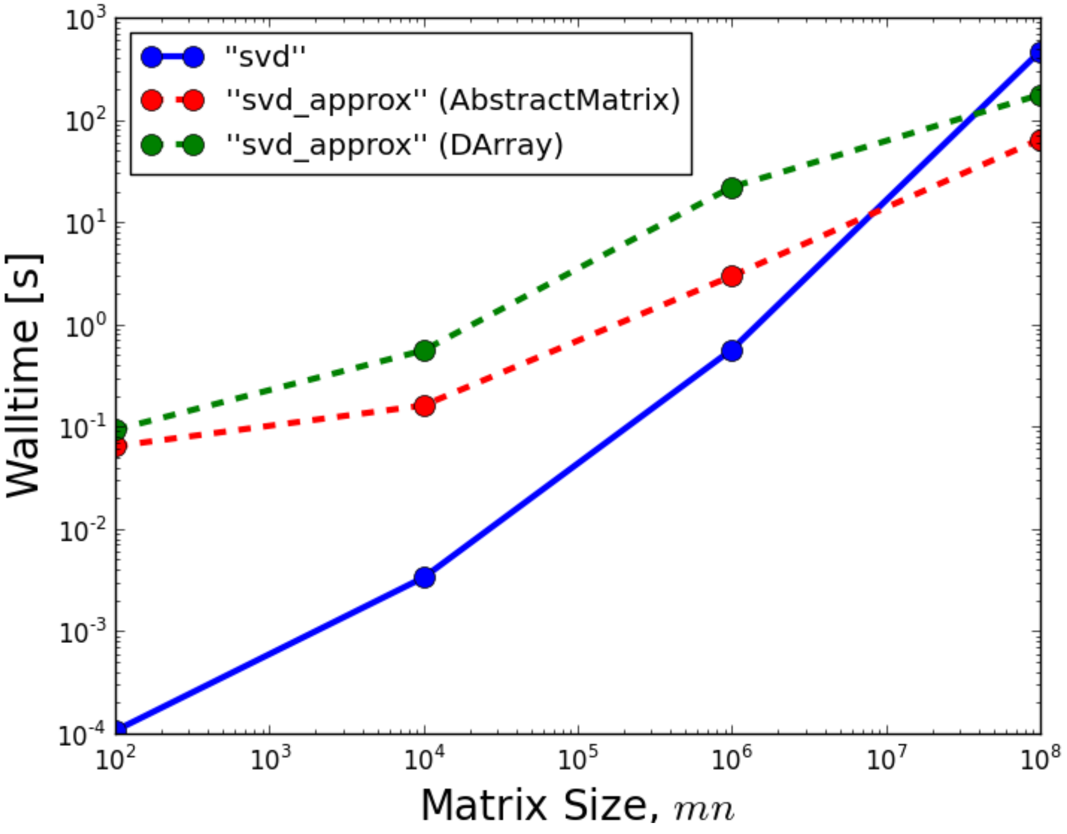


$$\mathbf{B_y} = \left[\begin{pmatrix} \vdots \\ \mathbf{A}^T\mathbf{x}_1 \\ \vdots \end{pmatrix} \begin{pmatrix} \vdots \\ \mathbf{A}^T\mathbf{x}_2 \\ \vdots \end{pmatrix} \cdots \begin{pmatrix} \vdots \\ \mathbf{A}^T\mathbf{x}_k \\ \vdots \end{pmatrix} \right]$$

Shared ⬜

Speedup: Matrix Size

- Tested the approximate SVD against the built in serial SVD (“`(U,S,V) = svd(A)`”)
  - “`svd(A)`” should scale as $O(mn^2)$
  - “`svd_approx(A)`” should scale as $O(kmn)$
- Approximate SVD does exhibit better scaling
  - `svd_approx` @ 10^8: 64s
  - `svd` @ 10^8: 468s
- Only use approximate SVD for large matrices



Summary

- Implemented an approximate SVD in Julia
  - Code is currently available on github at: “`https://github.com/alexjturner/SVDapprox`”
  - Scales as O(kmn) instead of O(mn^2) for standard SVD

- Currently works with both AbstractMatrices and DArrays
  - Different algorithm based on the user input

- Excellent speedup for large matrices ($>10^8$ elements)