An Approximate Singular Value Decomposition of Large Matrices in Julia

Alexander J. Turner\textsuperscript{1,*}

\textsuperscript{1}School of Engineering and Applied Sciences, Harvard University, Cambridge, MA, USA.

*aturner@fas.harvard.edu
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: $O(n^3)$ for a square matrix

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

$$A = U \Sigma V^T$$

Two attractive features:
- Better scaling: $O(kn^2)$ 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 \( \perp \) set
4 Compute the norm of “B”
5 Begin iterating
6 Randomly draw “ℓ” more columns
7 Obtain a new \( \perp \) set
8 Construct the “G” matrix using the \( \perp \) 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
- Takes either AbstractMatrices or DArrays
  - Algorithm proceeds differently depending on the array type

- Compactly store matrices as,
  \[
  \mathbf{B} = x_1 \mathbf{y}_1^T + x_2 \mathbf{y}_2^T + \ldots + x_k \mathbf{y}_k^T
  \]

- \( k \) 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 \( \mathbf{B} \)
Two validation cases: “rand \((N_x, N_y)\)” & “randn \((N_x, N_y)\)”

- True error: \[ \epsilon_T = \frac{||A - A_k||_F}{||A||_F} \]

- Approximation error: \[ \epsilon_A = \frac{||A - B||_F}{||A||_F} \]

- 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

![Singular Values Graph]

Error

![Error Graph]

Validation
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

![Graph of Singular Values](image1)

Error

![Graph of Percent Error](image2)
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))
```

<table>
<thead>
<tr>
<th>ITER 0:</th>
<th>Function</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rand_cols</td>
<td>43.20%</td>
</tr>
<tr>
<td></td>
<td>run_orth</td>
<td>2.44%</td>
</tr>
<tr>
<td></td>
<td>compute_B</td>
<td>48.75%</td>
</tr>
<tr>
<td></td>
<td>compute_norm</td>
<td>5.61%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.65529299 seconds</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ITER 1:</th>
<th>Function</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rand_cols</td>
<td>3.82%</td>
</tr>
<tr>
<td></td>
<td>run_orth</td>
<td>2.32%</td>
</tr>
<tr>
<td></td>
<td>compute_G</td>
<td>44.73%</td>
</tr>
<tr>
<td></td>
<td>eig_G</td>
<td>2.10%</td>
</tr>
<tr>
<td></td>
<td>svd_G</td>
<td>0.33%</td>
</tr>
<tr>
<td></td>
<td>compute_B</td>
<td>43.40%</td>
</tr>
<tr>
<td></td>
<td>compute_norm</td>
<td>3.29%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.21842003 seconds</td>
</tr>
</tbody>
</table>

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:
    $$ B = \sum_{i=1}^{k} x_i \left( A^T x_i \right)^T $$
  - 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

\[
B_y = \begin{bmatrix}
(A^T x_1) & (A^T x_2) & \cdots & (A^T x_k)
\end{bmatrix}
\]
Distributed Memory: A broken into $p$ parts, $k$ serial DGEMVs

\[
B_y = \begin{bmatrix}
\begin{pmatrix}
A^T x_1 \\
\vdots
\end{pmatrix} \\
\begin{pmatrix}
A^T x_2 \\
\vdots
\end{pmatrix} \\
\cdots
\begin{pmatrix}
A^T x_k \\
\vdots
\end{pmatrix}
\end{bmatrix}
\]
Approximate SVD does exhibit better scaling

- `svd_approx(A)` should scale as \(O(kmn)\)
- `svd(A)` should scale as \(O(mn^2)\)

Only use approximate SVD for large matrices

Testing 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 \(O(kmn)\)

Speedup: Matrix Size vs. Walltime [s]
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)