Homework 3

Math 5424

Numerical Linear Algebra

Alexander Novotny
Zuriah Quinton

October 23, 2023

1. Let A be $n \times m$ with $n > m$. Show that $\|\boldsymbol{A}^\top \boldsymbol{A}\|_2 = \|\boldsymbol{A}\|_2^2$ and $\kappa_2(\boldsymbol{A}^\top \boldsymbol{A}) = \kappa_2(\boldsymbol{A})^2$.

   Let M be $n \times n$ and positive definite and L be its Cholesky factor so that $M = \boldsymbol{L}\boldsymbol{L}^\top$. Show that $\|M\|_2 = \|L\|_2^2$ and $\kappa_2(\boldsymbol{M}) = \kappa_2(\boldsymbol{L})^2$.

   Proof. We have

   $$\left\|\boldsymbol{A}^\top \boldsymbol{A}\right\|_2 = \max_i \left|\lambda_i(\boldsymbol{A}^\top \boldsymbol{A})\right|$$

   since $\boldsymbol{A}^\top \boldsymbol{A}$ is normal, and

   $$\|\boldsymbol{A}\|_2^2 = \sqrt{\max_i \lambda_i(\boldsymbol{A}^\top \boldsymbol{A})}^2 = \max_i \left|\lambda_i(\boldsymbol{A}^\top \boldsymbol{A})\right|.$$

   So $\|\boldsymbol{A}^\top \boldsymbol{A}\|_2 = \|\boldsymbol{A}\|_2^2$.

   Next, we have

   $$\begin{aligned}
   \kappa_2(\boldsymbol{A}^\top \boldsymbol{A}) &= \left\|(\boldsymbol{A}^\top \boldsymbol{A})^{-1}\right\|_2 \left\|\boldsymbol{A}^\top \boldsymbol{A}\right\|_2 \\
   &= \max_i \left|\lambda_i((\boldsymbol{A}^\top \boldsymbol{A})^{-1})\right| \max_i \left|\lambda_i(\boldsymbol{A}^\top \boldsymbol{A})\right| \\
   &= \frac{1}{\sigma_m^2}\sigma_1^2 \\
   &= \frac{\sigma_1^2}{\sigma_m^2},
   \end{aligned}$$

   where σ_i is the i^{th} singular value of $\boldsymbol{A}$ in descending order. Further,

   $$\begin{aligned}
   \kappa_2(\boldsymbol{A})^2 &= \left(\frac{\sigma_1}{\sigma_m}\right)^2 \\
   &= \frac{\sigma_1^2}{\sigma_m^2}.
   \end{aligned}$$

   Note, we may have $\sigma_m = 0$ if TODO CHECK WORDING $\boldsymbol{A}$ is not full rank. In this case, we still have $\kappa_2(\boldsymbol{A}^\top \boldsymbol{A}) = \infty = \kappa_2(\boldsymbol{A})^2$.

   Next, for $\boldsymbol{M} = \boldsymbol{L}\boldsymbol{L}^\top$, we have

   $$\begin{aligned}
   \|\boldsymbol{M}\|_2 = \left\|\boldsymbol{L}\boldsymbol{L}^\top\right\|_2 &= \max_i \left|\lambda_i(\boldsymbol{L}\boldsymbol{L}^\top)\right| \\
   \text{and } \|\boldsymbol{L}\|_2^2 = \sqrt{\max_i \lambda_i(\boldsymbol{L}^\top \boldsymbol{L})}^2 &= \max_i \left|\lambda_i(\boldsymbol{L}^\top \boldsymbol{L})\right|.
   \end{aligned}$$

   By lemma 1 these are equivalent.

Finally, for the condition number of $\boldsymbol{M}$ we have

$$\begin{aligned}\kappa_2(\boldsymbol{M}) &= \kappa_2(\boldsymbol{L}\boldsymbol{L}^\top)\\ &= \left\|(\boldsymbol{L}\boldsymbol{L}^\top)^{-1}\right\|_2 \left\|\boldsymbol{L}\boldsymbol{L}^\top\right\|_2\\ &= \max_i \left|\lambda_i((\boldsymbol{L}\boldsymbol{L}^\top)^{-1})\right| \max_i \left|\lambda_i(\boldsymbol{L}\boldsymbol{L}^\top)\right|\\ &= \frac{1}{\lambda_m}\lambda_1\\ &= \frac{\sigma_1^2}{\sigma_m^2}.\end{aligned}$$

Since $\boldsymbol{M}$ is positive definite, we know $\sigma_i > 0$ for all i, so this value is finite. Further,

$$\begin{aligned}\kappa_2(\boldsymbol{L})^2 &= \left(\frac{\sigma_1}{\sigma_m}\right)^2\\ &= \frac{\sigma_1^2}{\sigma_m^2}.\end{aligned}$$

Thus, $\kappa_2(\boldsymbol{M}) = \kappa_2(\boldsymbol{L})^2$. ☺

Lemma 1. If $\vec{v}$ is an eigenvector of $\boldsymbol{L}\boldsymbol{L}^\top$, with λ its eigenvalue, then λ is an eigenvalue of $\boldsymbol{L}^\top\boldsymbol{L}$, with eigenvector $\boldsymbol{L}^\top\vec{v}$.

Proof. We have

$$\begin{aligned}\boldsymbol{L}\boldsymbol{L}^\top\vec{v} &= \lambda\vec{v} \qquad (1)\\ \implies (\boldsymbol{L}^\top\boldsymbol{L})(\boldsymbol{L}^\top\vec{v}) &= \lambda(\boldsymbol{L}^\top\vec{v}).\end{aligned}$$

For $\vec{v} \notin \ker(\boldsymbol{L}^\top)$. Suppose for contradiction that $\vec{v} \in \ker(\boldsymbol{L}^\top)$. Then eq. (1) is

$$\boldsymbol{L}\underbrace{\boldsymbol{L}^\top\vec{v}}_{=0} = \lambda\vec{v} \implies \lambda = 0,$$

which is absurd since the eigenvalues of a positive definite matrix are non-zero. Thus, we have $\vec{v} \notin \ker(\boldsymbol{L}^\top)$, and so λ is always an eigenvalue of $\boldsymbol{L}^\top\boldsymbol{L}$. ☺

2. In this question we will ask how to solve $\boldsymbol{B}\vec{y} = \vec{c}$ given a fast way to solve $\boldsymbol{A}\vec{x} = \vec{b}$, where $\boldsymbol{A} - \boldsymbol{B}$ is "small" in some sense.

   (a) Prove the *Sherman-Morrison formula*: Let $\boldsymbol{A}$ be nonsingular, $\vec{u}$ and $\vec{v}$ be column vectors, and $\boldsymbol{A} + \vec{u}\vec{v}^\top$ be nonsingular. Then $(\boldsymbol{A} + \vec{u}\vec{v}^\top)^{-1} = \boldsymbol{A}^{-1} - (\boldsymbol{A}^{-1}\vec{u}\vec{v}^\top\boldsymbol{A}^{-1})/(1 + \vec{v}^\top\boldsymbol{A}^{-1}\vec{u})$.

   More generally, prove the *Sherman-Morrison-Woodbury formula*: Let $\boldsymbol{U}$ and $\boldsymbol{V}$ be $n \times k$ rectangular matrices, where $k < n$ and $\boldsymbol{A}$ is $n \times n$. Then $\boldsymbol{T} = \boldsymbol{I} + \boldsymbol{V}^\top\boldsymbol{A}^{-1}\boldsymbol{U}$ is nonsingular if and only if $\boldsymbol{A} + \boldsymbol{U}\boldsymbol{V}^\top$ is nonsingular, in which case $(\boldsymbol{A} + \boldsymbol{U}\boldsymbol{V}^\top)^{-1} = \boldsymbol{A}^{-1} - \boldsymbol{A}^{-1}\boldsymbol{U}\boldsymbol{T}^{-1}\boldsymbol{V}^\top\boldsymbol{A}^{-1}$.

   Proof. First, we have

$$\begin{aligned}(\boldsymbol{A} + \vec{u}\vec{v}^\top)\left(\boldsymbol{A}^{-1} - \frac{\boldsymbol{A}^{-1}\vec{u}\vec{v}^\top\boldsymbol{A}^{-1}}{1 + \vec{v}^\top\boldsymbol{A}^{-1}\vec{u}}\right) &= \cancel{\boldsymbol{A}\boldsymbol{A}^{-1}} - \frac{\boldsymbol{I}\,\cancel{\boldsymbol{A}\boldsymbol{A}^{-1}}\vec{u}\vec{v}^\top\boldsymbol{A}^{-1}}{1 + \vec{v}^\top\boldsymbol{A}^{-1}\vec{u}} + \vec{u}\vec{v}^\top\boldsymbol{A}^{-1} - \frac{\vec{u}\vec{v}^\top\boldsymbol{A}^{-1}\vec{u}\vec{v}^\top\boldsymbol{A}^{-1}}{1 + \vec{v}^\top\boldsymbol{A}^{-1}\vec{u}}\\ &= \boldsymbol{I} - \frac{\vec{u}\vec{v}^\top\boldsymbol{A}^{-1} - \vec{u}\vec{v}^\top\boldsymbol{A}^{-1}(1 + \vec{v}^\top\boldsymbol{A}^{-1}\vec{u}) + \vec{u}\vec{v}^\top\boldsymbol{A}^{-1}\vec{u}\vec{v}^\top\boldsymbol{A}^{-1}}{1 + \vec{v}^\top\boldsymbol{A}^{-1}\vec{u}}\\ &= \boldsymbol{I} - \frac{\cancel{\vec{u}\vec{v}^\top\boldsymbol{A}^{-1}} - \cancel{\vec{u}\vec{v}^\top\boldsymbol{A}^{-1}} - \vec{u}\vec{v}^\top\boldsymbol{A}^{-1}\vec{v}^\top\boldsymbol{A}^{-1}\vec{u} + \vec{u}\vec{v}^\top\boldsymbol{A}^{-1}\vec{u}\vec{v}^\top\boldsymbol{A}^{-1}}{1 + \vec{v}^\top\boldsymbol{A}^{-1}\vec{u}}\\ &= \boldsymbol{I} - \frac{\vec{v}^\top\boldsymbol{A}^{-1}\vec{u}(-\cancel{\vec{u}\vec{v}^\top\boldsymbol{A}^{-1}} + \cancel{\vec{u}\vec{v}^\top\boldsymbol{A}^{-1}})}{1 + \vec{v}^\top\boldsymbol{A}^{-1}\vec{u}}\\ &= \boldsymbol{I},\end{aligned}$$

so $(\boldsymbol{A}+\vec{u}\vec{v}^\top)^{-1} = \boldsymbol{A}^{-1} - (\boldsymbol{A}^{-1}\vec{u}\vec{v}^\top\boldsymbol{A}^{-1})/(1+\vec{v}^\top\boldsymbol{A}^{-1}\vec{u})$. Then, assume $\boldsymbol{T} = \boldsymbol{I} + \boldsymbol{V}^\top\boldsymbol{A}^{-1}\boldsymbol{U}$ is nonsingular. We have

$$\begin{aligned}(\boldsymbol{A}+\boldsymbol{U}\boldsymbol{V}^\top)(\boldsymbol{A}^{-1} - \boldsymbol{A}^{-1}\boldsymbol{U}\boldsymbol{T}^{-1}\boldsymbol{V}^\top\boldsymbol{A}^{-1}) &= \cancel{\boldsymbol{A}\boldsymbol{A}^{-1}}^{\boldsymbol{I}} - \cancel{\boldsymbol{A}\boldsymbol{A}^{-1}}\boldsymbol{U}\boldsymbol{T}^{-1}\boldsymbol{V}^\top\boldsymbol{A}^{-1} \\ &\quad + \boldsymbol{U}\boldsymbol{V}^\top\boldsymbol{A}^{-1} - \boldsymbol{U}\boldsymbol{V}^\top\boldsymbol{A}^{-1}\boldsymbol{U}\boldsymbol{T}^{-1}\boldsymbol{V}^\top\boldsymbol{A}^{-1} \\ &= \boldsymbol{I} - \boldsymbol{U}(\boldsymbol{T}^{-1} - \boldsymbol{I} + \underbrace{\boldsymbol{V}^\top\boldsymbol{A}^{-1}\boldsymbol{U}}_{\boldsymbol{T}-\boldsymbol{I}}\boldsymbol{T}^{-1})\boldsymbol{V}^\top\boldsymbol{A}^{-1} \\ &= \boldsymbol{I} - \boldsymbol{U}(\cancel{\boldsymbol{T}^{-1}} - \cancel{\boldsymbol{I}} + \cancel{\boldsymbol{T}\boldsymbol{T}^{-1}} - \cancel{\boldsymbol{T}^{-1}})\boldsymbol{V}^\top\boldsymbol{A}^{-1} \\ &= \boldsymbol{I},\end{aligned}$$

so $\boldsymbol{A}+\boldsymbol{U}\boldsymbol{V}^\top$ is nonsingular and $(\boldsymbol{A}+\boldsymbol{U}\boldsymbol{V}^\top)^{-1} = \boldsymbol{A}^{-1} - \boldsymbol{A}^{-1}\boldsymbol{U}\boldsymbol{T}^{-1}\boldsymbol{V}^\top\boldsymbol{A}^{-1}$. Finally, assume $\boldsymbol{A}+\boldsymbol{U}\boldsymbol{V}^\top$ is nonsingular. Then we have

$$\begin{aligned}\boldsymbol{T}(\boldsymbol{I} - \boldsymbol{V}^\top(\boldsymbol{A}+\boldsymbol{U}\boldsymbol{V}^\top)^{-1}\boldsymbol{U}) &= (\boldsymbol{I}+\boldsymbol{V}^\top\boldsymbol{A}^{-1}\boldsymbol{U})(\boldsymbol{I} - \boldsymbol{V}^\top(\boldsymbol{A}+\boldsymbol{U}\boldsymbol{V}^\top)^{-1}\boldsymbol{U}) \\ &= \boldsymbol{I} - \boldsymbol{V}^\top(\boldsymbol{A}+\boldsymbol{U}\boldsymbol{V}^\top)^{-1}\boldsymbol{U} + \boldsymbol{V}^\top\boldsymbol{A}^{-1}\boldsymbol{U} - \boldsymbol{V}^\top\boldsymbol{A}^{-1}\boldsymbol{U}\boldsymbol{V}^\top(\boldsymbol{A}+\boldsymbol{U}\boldsymbol{V}^\top)^{-1}\boldsymbol{U} \\ &= \boldsymbol{I} - \boldsymbol{V}^\top\boldsymbol{A}^{-1}\big[\boldsymbol{A}(\boldsymbol{A}+\boldsymbol{U}\boldsymbol{V}^\top)^{-1} - \boldsymbol{I} + \boldsymbol{U}\boldsymbol{V}^\top(\boldsymbol{A}+\boldsymbol{U}\boldsymbol{V}^\top)^{-1}\big]\boldsymbol{U} \\ &= \boldsymbol{I} - \boldsymbol{V}^\top\boldsymbol{A}^{-1}\big[(\boldsymbol{A}+\boldsymbol{U}\boldsymbol{V}^\top)(\boldsymbol{A}+\boldsymbol{U}\boldsymbol{V}^\top)^{-1} - \boldsymbol{I}\big]\boldsymbol{U} \\ &= \boldsymbol{I},\end{aligned}$$

therefore $\boldsymbol{T}$ is invertible and non-singular. ☺

(b) If you have a fast algorithm to solve $\boldsymbol{A}\vec{x} = \vec{b}$, show how to build a fast solver for $\boldsymbol{B}\vec{y} = \vec{c}$, where $\boldsymbol{B} = \boldsymbol{A} + \vec{u}\vec{v}^\top$.

Answer. We have

$$\begin{aligned}\vec{y} &= \boldsymbol{B}^{-1}\vec{c} \\ &= (\boldsymbol{A}+\vec{u}\vec{v}^\top)^{-1}\vec{c} \\ &\overset{2a}{=} (\boldsymbol{A}^{-1} - (\boldsymbol{A}^{-1}\vec{u}\vec{v}^\top\boldsymbol{A}^{-1})/(1+\vec{v}^\top\boldsymbol{A}^{-1}\vec{u}))\vec{c} \\ &= \boldsymbol{A}^{-1}\vec{c} - (\boldsymbol{A}^{-1}\vec{u})\vec{v}^\top(\boldsymbol{A}^{-1}\vec{c})/(1+\vec{v}^\top(\boldsymbol{A}^{-1}\vec{u})).\end{aligned}$$

Then, an algorithm to quickly solve this problem can be found in algorithm 1.

Algorithm 1: An algorithm to quickly solve $(\boldsymbol{A}+\vec{u}\vec{v}^\top)\vec{y} = \vec{c}$ given an algorithm to quickly solve $\boldsymbol{A}\vec{x} = \vec{b}$.

Data: $\boldsymbol{A}$, $\vec{c}$, $\vec{u}$, $\vec{v}$
Result: $\vec{y} = (\boldsymbol{A}+\vec{u}\vec{v}^\top)^{-1}\vec{c}$
1 Solve $\boldsymbol{A}\vec{w} = \vec{c}$;
2 Solve $\boldsymbol{A}\vec{x} = \vec{u}$;
3 $\vec{y} \leftarrow \vec{w} - \vec{x}\vec{v}^\top\vec{w}/(1+\vec{v}^\top\vec{x})$;

3. Consider the linear system $\boldsymbol{A}\vec{x} = \vec{b}$ where

$$\boldsymbol{A} = \begin{bmatrix}2 & 1 & 1 & 0\\ 4 & 3 & 3 & 1\\ 8 & 7 & 9 & 5\\ 6 & 7 & 9 & 8\end{bmatrix} \quad \text{and} \quad \vec{b} = \begin{bmatrix}4\\ 11\\ 29\\ 30\end{bmatrix}.$$

(a) Calculate the appropriate determinants to show that $\boldsymbol{A}$ can be written as $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}$ where $\boldsymbol{L}$ is a unit lower triangular matrix and $\boldsymbol{U}$ is a non-singular upper triangular matrix. You may use MATLAB to compute the determinants. But the remaining parts of this problem should be done by paper-and-pencil.

Answer.

$$\det \boldsymbol{A}_1 = |1| = 1$$
$$\det \boldsymbol{A}_2 = \begin{vmatrix} 2 & 1 \\ 4 & 3 \end{vmatrix} = 2$$
$$\det \boldsymbol{A}_3 = \begin{vmatrix} 2 & 1 & 1 \\ 4 & 3 & 3 \\ 8 & 7 & 9 \end{vmatrix} = (2)(6) - (1)(12) + (1)(4) = 4$$
$$\det \boldsymbol{A}_4 = \begin{vmatrix} 2 & 1 & 1 & 0 \\ 4 & 3 & 3 & 1 \\ 8 & 7 & 9 & 5 \\ 6 & 7 & 9 & 8 \end{vmatrix}$$

(b) Compute the LU decomposition $\boldsymbol{A} = \boldsymbol{L}\boldsymbol{U}$ and convert $\boldsymbol{A}x = b$ into $\boldsymbol{U}x = y$ where $\boldsymbol{U}$ is upper triangular and solve for x.

Answer. The LU decomposition is please

(c) Using $\boldsymbol{L}$ and $\boldsymbol{U}$ from the earlier steps, solve the new system $\boldsymbol{A}\tilde{x} = \tilde{b}$ where

$$\tilde{b} = \begin{bmatrix} 5 \\ 15 \\ 41 \\ 45 \end{bmatrix}.$$

Do NOT perform the Gaussian elimination from scratch. You already have $\boldsymbol{U}$ and $\boldsymbol{L}$.

Answer.

4. Let $\boldsymbol{A}$ have the following economy (thin) SVD:

$$\boldsymbol{A} = \begin{bmatrix} 1 & 0 \\ 0 & 0.6 \\ 0 & -0.8 \end{bmatrix} \begin{bmatrix} 4 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 0.6 & -0.8 \\ -0.8 & -0.6 \end{bmatrix}^\top.$$

Answer the following questions without forming $\boldsymbol{A}$ and without using Matlab. None of these questions require using numerical software. At most, you need to be able to perform simple matrix arithmetic using paper-and-pencil.

(a) Find $\operatorname{rank}(\boldsymbol{A})$, $\|\boldsymbol{A}\|_2$ and $\|\boldsymbol{A}\|_F$.

Answer. We have

$$\begin{aligned} \operatorname{rank}(\boldsymbol{A}) &= 2, \\ \|\boldsymbol{A}\|_2 &= \max_i \sqrt{\lambda_i(A^\top A)} \\ &= \sqrt{\sigma_1^2(\boldsymbol{A})} \\ &= |\sigma_1(\boldsymbol{A})| = 4, \\ \|\boldsymbol{A}\|_F &= \sqrt{\operatorname{tr}((\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top)^\top(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top))} \\ &= \sqrt{\operatorname{tr}(\cancel{\boldsymbol{V}^\top}\cancel{\boldsymbol{V}}\boldsymbol{\Sigma}^\top\cancel{\boldsymbol{U}^\top}\cancel{\boldsymbol{U}}\boldsymbol{\Sigma})} \\ &= \sqrt{\operatorname{tr}^2(\boldsymbol{\Sigma})} \\ &= |\operatorname{tr}(\boldsymbol{\Sigma})| \\ &= 4 + 3 = 7. \end{aligned}$$

(b) Find an orthonormal basis for $\operatorname{Range}(\boldsymbol{A})$, $\operatorname{Null}(\boldsymbol{A})$, $\operatorname{Range}(\boldsymbol{A}^\top)$, and $\operatorname{Null}(\boldsymbol{A}^\top)$.

Answer. We have

$$\begin{aligned}
\text{Range}(\boldsymbol{A}) &= \text{span}\left\{\begin{bmatrix}1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0.6\\-0.8\end{bmatrix}\right\},\\
\text{Null}(\boldsymbol{A}) &= \text{span}\,\emptyset,\\
\text{Range}(\boldsymbol{A}^\top) &= \text{span}\left\{\begin{bmatrix}0.6\\-0.8\end{bmatrix}, \begin{bmatrix}-0.8\\-0.6\end{bmatrix}\right\},\\
\text{Null}(\boldsymbol{A}^\top) &= \mathbb{R}^3 \setminus \text{span}\left\{\begin{bmatrix}1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0.6\\-0.8\end{bmatrix}\right\}\\
&= \text{span}\left\{\begin{bmatrix}0\\0.8\\0.6\end{bmatrix}\right\}.
\end{aligned}$$

Note that the given sets consist of orthogonal unit vectors, so they are orthonormal bases for the sets which they span.

(c) Form the optimal rank-1 approximation $\boldsymbol{A}_1$ to $\boldsymbol{A}$ in the 2-norm? What is the error matrix $\boldsymbol{A} - \boldsymbol{A}_1$, and what is the norm of the error $\|\boldsymbol{A} - \boldsymbol{A}_1\|_2$? The error matrix and the norm of the error follow directly from the SVD; no computation is needed.

Answer. We have

$$\begin{aligned}
\boldsymbol{A}_1 &= \begin{bmatrix}1 & 0\\0 & 0.6\\0 & -0.8\end{bmatrix}\begin{bmatrix}4 & 0\\0 & 0\end{bmatrix}\begin{bmatrix}0.6 & -0.8\\-0.8 & -0.6\end{bmatrix}^\top\\
&= \begin{bmatrix}1\\0\\0\end{bmatrix}[4]\begin{bmatrix}0.6\\-0.8\end{bmatrix}^\top,\\
\boldsymbol{A} - \boldsymbol{A}_1 &= \begin{bmatrix}1 & 0\\0 & 0.6\\0 & -0.8\end{bmatrix}\left(\begin{bmatrix}4 & 0\\0 & 3\end{bmatrix} - \begin{bmatrix}4 & 0\\0 & 0\end{bmatrix}\right)\begin{bmatrix}0.6 & -0.8\\-0.8 & -0.6\end{bmatrix}^\top\\
&= \begin{bmatrix}1 & 0\\0 & 0.6\\0 & -0.8\end{bmatrix}\begin{bmatrix}0 & 0\\0 & 3\end{bmatrix}\begin{bmatrix}0.6 & -0.8\\-0.8 & -0.6\end{bmatrix}^\top,\\
\|\boldsymbol{A} - \boldsymbol{A}_1\|_2 &= 3.
\end{aligned}$$

5. (From Golub and van Loan) Given $\boldsymbol{A} \in \mathbb{R}^{m\times n}$, let σ_1 denote the largest singular value of $\boldsymbol{A}$. Prove that

$$\sigma_1(\boldsymbol{A}) = \max_{\substack{\vec{y}\in\mathbb{R}^m,\\ \vec{x}\in\mathbb{R}^n}} \frac{\vec{y}^\top \boldsymbol{A}\vec{x}}{\|\vec{y}\|_2\|\vec{x}\|_2}.$$

Proof. For $\vec{y}, \vec{x} \neq 0$ we have

$$\begin{aligned}
\max_{\substack{\vec{y}\in\mathbb{R}^m,\\ \vec{x}\in\mathbb{R}^n}} \frac{\vec{y}^\top \boldsymbol{A}\vec{x}}{\|\vec{y}\|_2\|\vec{x}\|_2} &= \max_{\substack{\vec{y}\in\mathbb{R}^m,\\ \vec{x}\in\mathbb{R}^n}} \left(\frac{\vec{y}}{\|\vec{y}\|_2}\right)^\top \frac{\boldsymbol{A}\vec{x}}{\|\vec{x}\|_2}\\
&= \max_{\substack{\vec{y}\in\mathbb{R}^n\\ \|\vec{y}\|=1,\\ \vec{x}\in\mathbb{R}^n}} \vec{y}^\top \frac{\boldsymbol{A}\vec{x}}{\|\vec{x}\|_2} \qquad (2)
\end{aligned}$$

which is the inner product, or the projection of a unit vector $\vec{y}$ onto $\frac{\boldsymbol{A}\vec{x}}{\|\vec{x}\|_2}$. The maximum of this quantity is achieved

when $\vec{y}$ is the unit vector in the direction of $\frac{\boldsymbol{A}\vec{x}}{\|\vec{x}\|_2}$. In other words,

$$\begin{aligned}\vec{y} &= \frac{\dfrac{\boldsymbol{A}\vec{x}}{\|\vec{x}\|_2}}{\left\|\dfrac{\boldsymbol{A}\vec{x}}{\|\vec{x}\|_2}\right\|_2} \\ &= \frac{\boldsymbol{A}\vec{x}}{\|\boldsymbol{A}\vec{x}\|_2}.\end{aligned}$$

Then in eq. (2) we have

$$\begin{aligned}\max_{\substack{\vec{y}\in\mathbb{R}^m,\\ \vec{x}\in\mathbb{R}^n}} \frac{\vec{y}^\top \boldsymbol{A}\vec{x}}{\|\vec{y}\|_2\|\vec{x}\|_2} &= \max_{\substack{\vec{y}\in\mathbb{R}^n\\ \|\vec{y}\|=1,\\ \vec{x}\in\mathbb{R}^n}} \vec{y}^\top \frac{\boldsymbol{A}\vec{x}}{\|\vec{x}\|_2} \\ &= \max_{\vec{x}\in\mathbb{R}^n} \left(\frac{\boldsymbol{A}\vec{x}}{\|\boldsymbol{A}\vec{x}\|_2}\right)^\top \frac{\boldsymbol{A}\vec{x}}{\|\vec{x}\|_2} \\ &= \max_{\vec{x}\in\mathbb{R}^n} \frac{\|\boldsymbol{A}\vec{x}\|_2^2}{\|\boldsymbol{A}\vec{x}\|_2\|\vec{x}\|_2} \\ &= \max_{\vec{x}\in\mathbb{R}^n} \frac{\|\boldsymbol{A}\vec{x}\|_2}{\|\vec{x}\|_2} \\ &= \|\boldsymbol{A}\|_2 \\ &= \sigma_1(\boldsymbol{A}).\end{aligned}$$

6. Let $\boldsymbol{A} \in \mathbb{R}^{m\times n}$ have the singular value decomposition

$$\boldsymbol{A} = \sum_{j=1}^{r} \sigma_j \vec{u}_j \vec{v}_j^\top.$$

We showed in class that the matrix

$$\boldsymbol{A}_k = \sum_{j=1}^{k} \sigma_j \vec{u}_j \vec{v}_j^\top$$

is an optimal rank-k approximation to $\boldsymbol{A}$ in the 2-norm and $\|\boldsymbol{A} - \boldsymbol{A}_k\| = \sigma_{k+1}$. But this minimizer (optimal approximant) is not unique. In other words, using truncated SVD we can find another rank-k matrix $\tilde{\boldsymbol{A}}_k \in \mathbb{R}^{m\times n}$ such that $\|\boldsymbol{A} - \tilde{\boldsymbol{A}}_k\| = \sigma_{k+1}$. In this problem, you will prove this fact.
Define

$$\tilde{\boldsymbol{A}}_k = \sum_{j=1}^{k} (\sigma_j - \eta_j)\vec{u}_j \vec{v}_j^\top \quad \text{where} \quad 0 \le |\eta_j| < \sigma_{k+1}. \tag{3}$$

Show that has $\tilde{\boldsymbol{A}}_k$ in (3) has rank-k and minimizes the error, i.e, $\|\boldsymbol{A} - \tilde{\boldsymbol{A}}_k\| = \sigma_{k+1}$.

Proof. Observe that $\sum_{j=1}^{k}(\sigma_j - \eta_j)\vec{u}_j\vec{v}_j^\top$ is the SVD of $\tilde{\boldsymbol{A}}_k$, with singular values of $(\sigma_j - \eta_j)$. Since $|\eta_j| < \sigma_{k+1} \le \sigma_j$ for all $1 \le j \le k$, then $\sigma_j - \eta_j > 0$, and $\tilde{\boldsymbol{A}}_k$ has k non-zero singular values, so it has rank k. As well, we have

$$\begin{aligned}\boldsymbol{A} - \tilde{\boldsymbol{A}}_k &= \sum_{j=1}^{r} \sigma_j \vec{u}_j \vec{v}_j^\top - \sum_{j=1}^{k} (\sigma_j - \eta_j)\vec{u}_j\vec{v}_j^\top \\ &= \sum_{j=1}^{k} \left[\sigma_j \vec{u}_j \vec{v}_j^\top - (\sigma_j - \eta_j)\vec{u}_j\vec{v}_j^\top\right] + \sum_{j=k+1}^{r} \sigma_j \vec{u}_j \vec{v}_j^\top \\ &= \sum_{j=1}^{k} \eta_j \vec{u}_j \vec{v}_j^\top + \sum_{j=k+1}^{r} \sigma_j \vec{u}_j \vec{v}_j^\top.\end{aligned}$$

Note, then, that this is the SVD of $\boldsymbol{A} - \tilde{\boldsymbol{A}}_k$, with eigenvalues of $|\eta_j|$ for $1 \leq j \leq k$ and σ_j for $k+1 \leq j \leq r$. Since $|\eta_j| < \sigma_{k+1}$ for all j and $\sigma_{k+1} \geq \sigma_k$ for all j, we must have

$$\|\boldsymbol{A} - \tilde{\boldsymbol{A}}_k\|_2 = \sigma_{k+1},$$

as required. ☺

7. Approximation of a Fingerprint Image via SVD: You will complete the Matlab Script `Homework3Problem7.m` to answer this question. You will attach the completed script to the .pdf file.

   Before start completing your code, it will help use the commands `help svd`, `help subplot`, `help semilogy` to understand how to use these functions.

   Lines 7-10 load the image into Matlab and plot it. The resulting matrix $\boldsymbol{A}$ in your workspace is a 1133×784 matrix with entries consisting of only ones and zeroes; ones correspond to the white spots in the image and zeroes to the black spots.

   (a) Compute the short (reduced) SVD of $\boldsymbol{A}$ using Matlab. Then plot the normalized singular values under `subplot(2,1,2)`. This figure might be hard to read due to the scale of the y-axis. Then plot only the leading 700 normalized singular values under `subplot(2,1,2)`. What do you observe? What would be your decision for the (numerical) rank of $\boldsymbol{A}$ based on these comments? Justify your reasoning. Does the result of the built-in rank command coincide with your decision?

   Answer. Plots of the singular values of $\boldsymbol{A}$ can be found in fig. 1. We notice a significant drop off in singular values after the 654^{th} singular value. Because of this, the numerical rank of $\boldsymbol{A}$ should be 654. Indeed, this is the computed rank of $\boldsymbol{A}$.

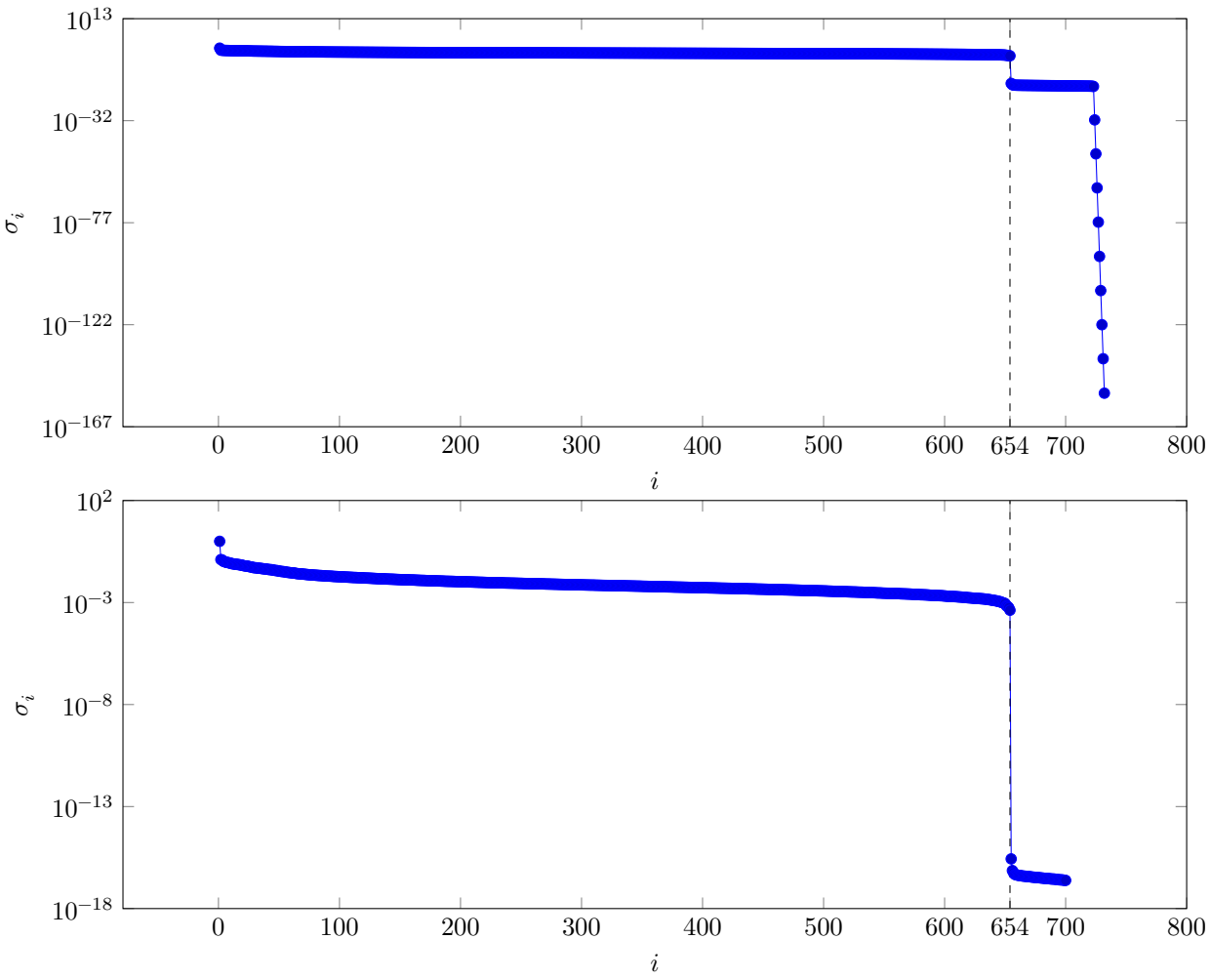
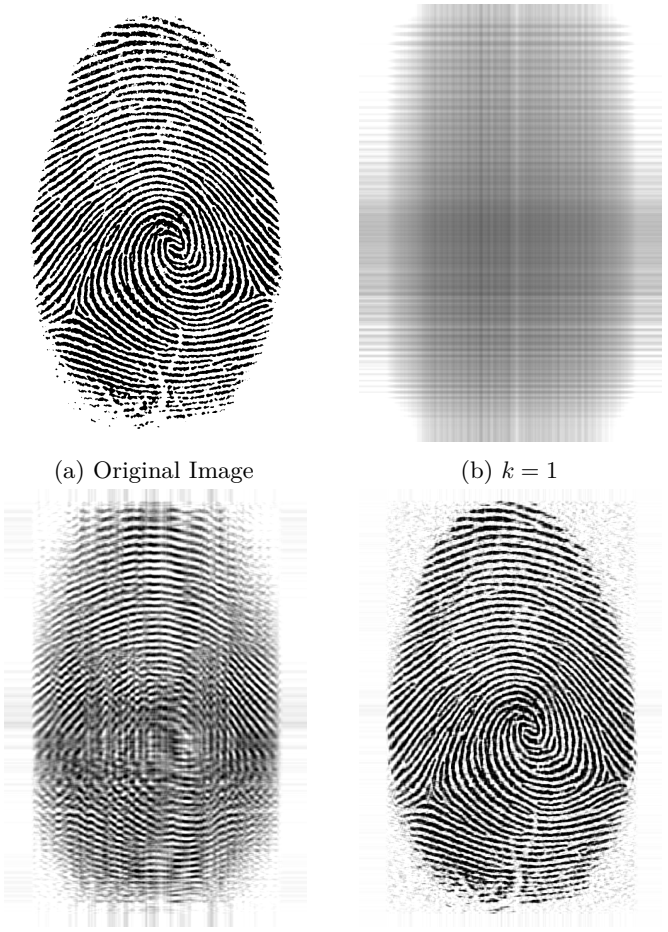


Figure 1: Plots showing all singular values of $\boldsymbol{A}$ and just the first 700 singular values of $\boldsymbol{A}$. Note the large gap between values that occurs at the calculated rank of $\boldsymbol{A}$ (654).

(b) Construct the optimal rank-k approximation $\boldsymbol{A}_k$ to $\boldsymbol{A}$ in the 2-norm for $k = 1, k = 10$, and $k = 50$. In each case, compute the relative error $\frac{\|\boldsymbol{A}-\boldsymbol{A}_k\|_2}{\|\boldsymbol{A}\|_2}$. Note that you do NOT need to form $\boldsymbol{A} - \boldsymbol{A}_k$ to compute these error values; the singular values are all you need. Use `subplot(2,2,1)` to plot the original image in the top left corner. Then, use the `imshow` command on the three low-rank approximations and plot them in the `subplot(2,2,2)`, `subplot(2,2,3)`, and `subplot(2,2,4)` spots. Put appropriate titles on each plot, e.g., "original image", "rank-1 approximation" etc., using the title command. Plots without appropriate labels and explanations will lose points. These plots need to be attached to the .pdf file.

Answer. The plots of the rank-k approximations, for $k = 1, 10, 50$, can be found in fig. 2, with program output displayed below:

```
Computed Rank of A = 654
Relative error for A_1 = 0.12761236428030379
Relative error for A_10 = 0.08040029616965558
Relative error for A_50 = 0.03400080504768164
```



(a) Original Image (b) $k = 1$

(c) $k = 10$ (d) $k = 50$

Figure 2: Original fingerprint image compared to its optimal rank-k approximations.

The code for this question is given below:

```rust
#![feature(float_next_up_down)]
use image::{DynamicImage::ImageLuma8, GrayImage};
use nalgebra::{
    allocator::Allocator, ComplexField, DMatrix, DefaultAllocator, Dim, DimMin, DimMinimum, Matrix,
    RawStorage, Storage, SVD,
};
use num_traits::AsPrimitive;
use show_image::create_window;
use std::{
    error::Error,
    fs::{self, File},
    io::Write,
};

#[show_image::main]
fn main() -> Result<(), Box<dyn Error>> {
    // Open original .pbm image
    let img = image::open("HW4_Prob6_fingerprint.pbm")?;
    // Display image
    let window = create_window("Original Image", Default::default())?;
    window.set_image("Figure 1", img.clone())?;

    // Assert image is 8-bit monochrome
    let ImageLuma8(img) = img else { unreachable!() };
    // Convert image to matrix of doubles
    let a = DMatrix::from_row_iterator(
```

```
        img.height() as usize,
        img.width() as usize,
        img.as_raw().iter().map(|x| *x as f64),
    );

    // Save size of A for later
    let (m, n) = a.shape();

    // check matrix is still the same image
    mat_to_img_show(&a, "Original matrix as image check")?;

    // Compute SVD of A
    let svd = SVD::new(a, true, true);

    // Create output file
    fs::create_dir_all("./out")?;
    let mut out = File::create("./out/all_singular_values7.dat")?;
    // Print normalized singular values to output file
    writeln!(out, "#all singular values of A")?;
    for s in &svd.singular_values {
        writeln!(out, "{}", s / svd.singular_values[0])?;
    }
    // notify Terminal of completed SVD printing
    eprintln!("finished printing");

    // The rank() function in matlab uses a default tolerance value. The nalgebra version doesn't - it needs to be provided. So we replicate matlab's default tolerance.
    // tol from MatLab = max(size(A))*eps(norm(A))
    // From matlab's documentation, eps(x) is the positive distance between |x| and the next largest float.
    // Rust's f64::next_up(x) returns that next largest float. As well, we are using the already computed largest singular value as norm(A).
    let tol = (m.max(n) as f64) * (f64::next_up(svd.singular_values[0]) - svd.singular_values[0]);

    // Compute the Rank of A with the computed tolerance
    println!("Computed Rank of A = {}", svd.rank(tol));

    // Compute optimal rank-k approximations, display them, and save them
    let mut approx = DMatrix::zeros(m, n);
    let mut prev_k = 0;
    for k in [1, 10, 50] {
        approx = rank_k_approx(&svd, k, &approx, prev_k);
        prev_k = k;
        mat_to_img_show(&approx, format!("Rank_{k}_Approximation"))?;
        // Print relative errors
        println!(
            "Relative error for A_{k} = {}",
            // Rust is 0 indexed, need to subtract 1
            svd.singular_values[(k + 1) - 1] / svd.singular_values[1 - 1]
        );
    }

    // keep images up until original window is closed
    for _event in window.event_channel()? {}
    Ok(())
}

/// Calculates the optimal rank `k` approximation of a matrix represented by its singular value decomosition.
/// Uses previously-computed optimal approximation `prev`, which is rank `prev_k`, which must be less than `k`.
/// For new approximation, pass zero matrix for `prev` and 0 for `prev_k`
fn rank_k_approx<T: ComplexField, R: DimMin<C>, C: Dim>(
    svd: &SVD<T, R, C>,
    k: usize,
    prev: &Matrix<T, R, C, impl Storage<T, R, C>>,
    prev_k: usize,
) -> Matrix<T, R, C, <nalgebra::DefaultAllocator as nalgebra::allocator::Allocator<T, R, C>>::Buffer>
where
    DefaultAllocator: Allocator<T, DimMinimum<R, C>, C>
        + Allocator<T, R, DimMinimum<R, C>>
        + Allocator<T::RealField, DimMinimum<R, C>>
```

```rust
        + Allocator<T, R, C>,
{
    let mut u = svd.u.as_ref().unwrap().clone();
    // Multiply σ_i u_i for i ∈ [1, k]
    for i in prev_k..k {
        let val = svd.singular_values[i].clone();
        u.column_mut(i).scale_mut(val);
    }
    // Multiply Σ_{i=1}^{k} (σ_i u_i) v_i^⊤
    prev + u.columns(prev_k, k - prev_k) * svd.v_t.as_ref().unwrap().rows(prev_k, k - prev_k)
}

/// Displays and saves the image stored in mat to the file "./out/<`wind_name`>.png"
fn mat_to_img_show<T: AsPrimitive<u8>, R: Dim, C: Dim, S: RawStorage<T, R, C>>(
    mat: &Matrix<T, R, C, S>,
    wind_name: impl AsRef<str>,
) -> Result<(), Box<dyn Error>> {
    // Convert matrix to 8-bit monochrome image
    // Annoyingly, image crate and matrix crate use different size types, so converting is required
    let im2 = GrayImage::from_fn(mat.ncols() as u32, mat.nrows() as u32, |c, r| {
        image::Luma([mat[(r as usize, c as usize)].as_()])
    });

    // Create window and display image
    let window2 = create_window(wind_name.as_ref(), Default::default())?;
    window2.set_image("f", im2.clone())?;

    // Save image as png in output folder
    im2.save_with_format(
        format!("./out/{}.png", wind_name.as_ref()),
        image::ImageFormat::Png,
    )?;

    Ok(())
}
```

8. In the previous problem you used SVD to compress a black-and-white image. In this example, you will use SVD to compute a low-rank approximation to the color image hokiebirdwithacat.jpg. You will complete the Matlab Script Homework3Problem8.m to answer this question. You will attach the completed script to the .pdf file.

   Lines 7-20 load the image into Matlab, plot the original image, and extract the images correspond to every color layer, and convert these three layer images to double precision matrices A1, A2, and A3.

   (a) Compute the SVDs of every layer A1, A2, and A3. Then compute the vector of normalized singular values for every layer. Using the subplot comment (and logarithmic y-axis), plot all three vectors in Figure 2.

   Answer. The normalized singular values for each channel can be found in fig. 3. Note that the color of the plot indicates which channel the singular values are from.

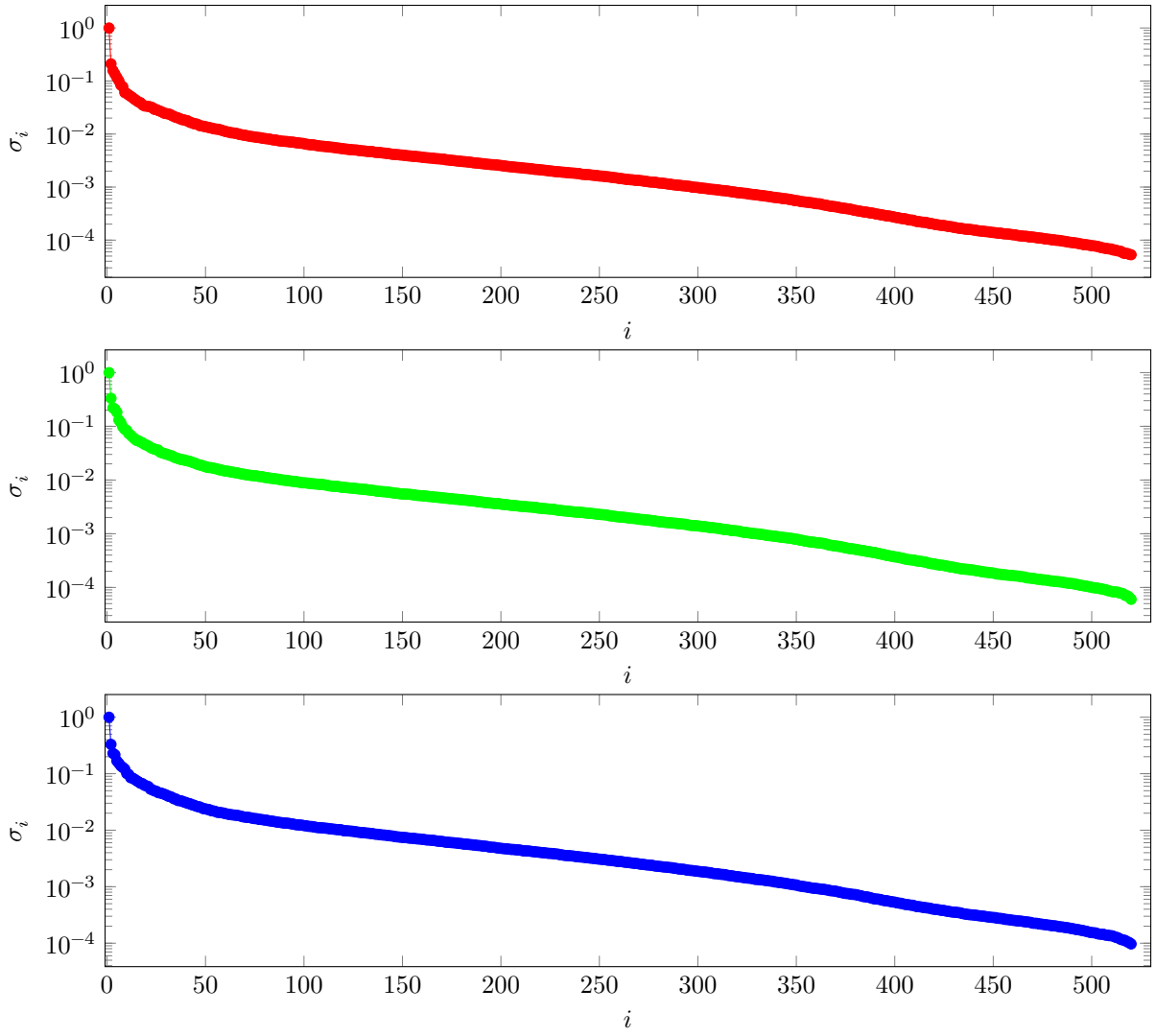


Figure 3: Plots showing all normalized singular values of hokiebirdwithacat, split into red, green, and blue components. The color of the plot indicates which component the singular values were calculated from

(b) Use the same error tolerance for every layer: Find the smallest rank-k for each layer such that the relative error in each layer is less than

(i) 10% (ii) 5% (iii) 1%

So, you will have three approximations. For each case, plot the low-rank image (either as a new figure or all in one plot using subplot). These plots need to be attached to the .pdf file. Make sure to clearly indicate the rank of ever layer and which plot corresponds to which error tolerance. Did you obtain the same k value for every layer? Comment on your results.

Answer. The approximations can be found in fig. 4. Note that different k values are chosen per channel, and that the red channel always uses the smallest k, while blue uses the largest.

(a) Original Image (b) $tol = 10\%,\ k = 6, 7, 10$ (c) $tol = 5\%,\ k = 12, 17, 23$ (d) $tol = 1\%,\ k = 66, 89, 119$

Figure 4: Original hokie bird image compared to its optimal approximations for different error tolerances. Selected k values for the given tolerances are shown.

(c) Use different error tolerances for different layers: Now we will choose different tolerances for different values. Pick three different selections:

- 50% error for Layer 1, 1% error for Layer 2, and 1% error for Layer 3,
- 1% error for Layer 1, 50% error for Layer 2, and 1% error for Layer 3,
- 1% error for Layer 1, 1% error for Layer 2, and 50% error for Layer 3,

So, you will have three approximations. For each case, plot the low-rank image (either as a new figure or all in one plot using subplot). These plots need to be attached to the .pdf file. Make sure to clearly indicate the rank of ever layer and which plot corresponds to which error tolerance. Comment on your results.

Answer. The approximations can be found in fig. 5. Note that the chosen k values are only for the channel with the 50% error tolerance - since the k values for 1% error tolerance in each channel are given in fig. 4d above. The 50% error in blue looks much clearer than the other two channels - this is probably due to the relative lack of blue in the original image compared to the other two channels. Interestingly, the chosen $k = 1$ for all channels - since the rank 1 approximation of each channel includes more than 50% of the original information.

(a) Red $k = 1$ (b) Green $k = 1$ (c) Blue $k = 1$

Figure 5: Optimal approximations with a single channel constrained to 50% error tolerance, with the rest of the channels at 1%. The channel with 50% tolerance is labeled, in addition to its chosen k value.

The code for this question is given below:

```
#![feature(float_next_up_down)]
#![feature(array_methods)]
#![feature(array_try_map)]
```

```rust
use std::error::Error;
use std::ops::MulAssign;
use std::{
    fs::{self, File},
    io::Write,
};

use image::{DynamicImage::ImageRgb8, ImageBuffer, Rgb, RgbImage};
use nalgebra::{
    allocator::Allocator, ComplexField, DMatrix, DefaultAllocator, Dim, DimMin, DimMinimum, Matrix,
    RawStorage, Storage, SVD,
};
use num_traits::AsPrimitive;
use show_image::create_window;

#[show_image::main]
fn main() -> Result<(), Box<dyn Error>> {
    // Open original .pbm image
    let img = image::open("hokiebirdwithacat.jpg")?;
    // Display image
    let window = create_window("Original Image", Default::default())?;
    window.set_image("Figure 1", img.clone())?;

    // Assert image is 8-bit Rgb
    let ImageRgb8(img) = img else { unreachable!() };
    // Convert image to 3 matrices of doubles
    let a = [0, 1, 2].map(|c| rgb_to_mat(&img, c));

    // Save size of A for later
    let (m, n) = a[0].shape();

    // check matrix is still the same image
    mat_to_img_show(a.each_ref(), "RGB Image Conversion Check")?;

    // Compute SVD of A
    let svd = a.map(|a| SVD::new(a, true, true));

    // Create output file
    fs::create_dir_all("./out")?;
    let mut out = File::create("./out/all_singular_values8.dat")?;
    // Print normalized singular values to output file
    writeln!(out, "{:8} {:8} {:8}", "A1", "A2", "A3")?;
    // Write normalized singular values out
    for sigma in zip_array(svd.each_ref().map(|svd| svd.singular_values.iter())) {
        for (sigma, svd) in sigma.into_iter().zip(&svd) {
            write!(out, "{:8.6} ", *sigma / svd.singular_values[0])?;
        }
        writeln!(out)?;
    }
    // notify Terminal of completed SVD printing
    eprintln!("finished printing");

    // Optimal rank-0 approximations of the channels. Used for computing iteratively higher rank
    ↪ approximations
    let mut approx = [
        (0, DMatrix::zeros(m, n)),
        (0, DMatrix::zeros(m, n)),
        (0, DMatrix::zeros(m, n)),
    ];

    for err in [0.10, 0.05, 0.01] {
        // Compute the next optimal approximation for each channel based on the target error, as well as the
        ↪ previous approximation and its rank
        for ((prev_k, approx), svd) in approx.iter_mut().zip(&svd) {
            (*prev_k, *approx) = rel_err_approx(svd, err, approx, *prev_k);
        }

        // Show and save approximation of image
        mat_to_img_show(
            approx.each_ref().map(|(_, approx)| approx),
```

```rust
            format!("err_all_{err}_Approximation"),
        )?;

        // Print rank and error information
        println!("Goal error = {err}");
        for (i, ((k, _), svd)) in approx.iter().zip(&svd).enumerate() {
            println!(
                "Smallest k for layer {i} = {k:3}  Actual error = {:8.6}  Error for k-1: {:8.6}",
                svd.singular_values[(k + 1) - 1] / svd.singular_values[1 - 1],
                svd.singular_values[(k) - 1] / svd.singular_values[1 - 1],
            );
        }
    }

    // 1% error approximations were already previously computed for each channel
    let per_1_approx = approx;
    // Compute 50% error approximations for each channel
    let per_50_approx = svd.map(|svd| rel_err_approx(&svd, 0.5, &DMatrix::zeros(m, n), 0));

    // Display all combinations of 50,1% error approximations where exactly a single channel uses its 50%
     ↪  error approximation,
    // and other channels use their 1% error approximation
    mat_to_img_show(
        [&per_50_approx[0].1, &per_1_approx[1].1, &per_1_approx[2].1],
        "50-err-red",
    )?;
    mat_to_img_show(
        [&per_1_approx[0].1, &per_50_approx[1].1, &per_1_approx[2].1],
        "50-err-green",
    )?;
    mat_to_img_show(
        [&per_1_approx[0].1, &per_1_approx[1].1, &per_50_approx[2].1],
        "50-err-blue",
    )?;

    // Print rank information for the 50% error approximations
    for (i, (k, _)) in per_50_approx.iter().enumerate() {
        println!("Err 50 approximation for channel {} k = {k}", i + 1)
    }

    Ok(())
}

/// Converts image buffer reference to matrix of f64 for given channel `c`
fn rgb_to_mat(img: &ImageBuffer<Rgb<u8>, Vec<u8>>, c: usize) -> DMatrix<f64> {
    DMatrix::from_row_iterator(
        img.height() as usize,
        img.width() as usize,
        img.enumerate_pixels().map(|(_, _, x)| x.0[c] as f64),
    )
}

/// Compute optimal approximation with relative error less than err.
/// Calls `rank_k_approx` after computing required k for error, then returns that rank and the approximation.
fn rel_err_approx<T: ComplexField, R: DimMin<C>, C: Dim>(
    svd: &SVD<T, R, C>,
    mut err: T::RealField,
    prev: &Matrix<T, R, C, impl Storage<T, R, C>>,
    prev_k: usize,
) -> (
    usize,
    Matrix<
        T,
        R,
        C,
        <nalgebra::DefaultAllocator as nalgebra::allocator::Allocator<T, R, C>>::Buffer,
    >,
)
where
    DefaultAllocator: Allocator<T, DimMinimum<R, C>, C>
```

```rust
        + Allocator<T, R, DimMinimum<R, C>>
        + Allocator<T::RealField, DimMinimum<R, C>>
        + Allocator<T, R, C>,
    T::RealField: for<'a> MulAssign<&'a T::RealField>,
{
    // Calculate the absolute error we're interested in finding
    err *= &svd.singular_values[0];

    // The singular values are already sorted, so we can binary search through them to find the one we're looking for.
    // We know that our previous approximation had less error than the one we're currently looking for, so we don't need to search through the first prev_k singular values.
    // We therefore exclude those from the search
    let k = match svd.singular_values.as_slice()[(prev_k + 1)..]
        // f64 isn't totally ordered, so we can't use a normal binary search.
        // Instead, it's partially ordered, so we use partial_cmp(...).unwrap() to assert that we are in a totally ordered subset (no NaNs).
        // As well, the singular values are sorted in descending order, so we reverse the comparison to err.partial_cmp(x).
        .binary_search_by(|x| err.partial_cmp(x).unwrap())
    {
        // Binary search can return two things:
        // Ok - it found a singular value which is exactly err. Then i is the index of that singular value.
        // Err - it didn't find a singular value which is exactly err. Then i is the index in the list where we could insert err and it would still be sorted.
        // In both cases, i is the choice of k we want, since it means that sigma_i < err and sigma_{i - 1} >= err - and the vector is 0-indexed.
        // i in this specific case is the index from (prev_k + 1) so we need to add that.
        Ok(i) | Err(i) => i + prev_k + 1,
    };

    // Calculate the approximation based on the k we chose above. Then return this along with that k.
    (k, rank_k_approx(svd, k, prev, prev_k))
}

/// Calculates the optimal rank `k` approximation of a matrix represented by its singular value decomosition.
/// Uses previously-computed optimal approximation `prev`, which is rank `prev_k`, which must be less than `k`.
/// For new approximation, pass zero matrix for `prev` and 0 for `prev_k`
fn rank_k_approx<T: ComplexField, R: DimMin<C>, C: Dim>(
    svd: &SVD<T, R, C>,
    k: usize,
    prev: &Matrix<T, R, C, impl Storage<T, R, C>>,
    prev_k: usize,
) -> Matrix<T, R, C, <nalgebra::DefaultAllocator as nalgebra::allocator::Allocator<T, R, C>>::Buffer>
where
    DefaultAllocator: Allocator<T, DimMinimum<R, C>, C>
        + Allocator<T, R, DimMinimum<R, C>>
        + Allocator<T::RealField, DimMinimum<R, C>>
        + Allocator<T, R, C>,
{
    let mut u = svd.u.as_ref().unwrap().clone();
    // Multiply \sigma_i u_i for i \in [1, k]
    for i in prev_k..k {
        let val = svd.singular_values[i].clone();
        u.column_mut(i).scale_mut(val);
    }
    // Multiply \sum_{i=1}^{k} (\sigma_i u_i) v_i^T
    prev + u.columns(prev_k, k - prev_k) * svd.v_t.as_ref().unwrap().rows(prev_k, k - prev_k)
}

/// Displays and saves the image stored in the three matrices (representing rgb channels) to the file "./out/<`wind_name`>.png"
fn mat_to_img_show<T: AsPrimitive<u8>, R: Dim, C: Dim, S: RawStorage<T, R, C>>(
    mats: [&Matrix<T, R, C, S>; 3],
    wind_name: impl AsRef<str>,
) -> Result<(), Box<dyn Error>> {
    // Convert matrix to 8-bit rgb image
    // Annoyingly, image crate and matrix crate use different size types, so converting is required
```

```rust
    let im2 = RgbImage::from_fn(mats[0].ncols() as u32, mats[0].nrows() as u32, |c, r| {
        let i = (r as usize, c as usize);
        image::Rgb(mats.map(|m| m[i].as_()))
    });

    // Create window and display image
    let window2 = create_window(wind_name.as_ref(), Default::default())?;
    window2.set_image("f", im2.clone())?;

    // Save image as png in output folder
    im2.save_with_format(
        format!("./out/{}.png", wind_name.as_ref()),
        image::ImageFormat::Png,
    )?;

    Ok(())
}

// Some helper code. Allows us to zip an array of iterators into an iterator over arrays
pub struct ZipArray<T, const N: usize> {
    array: [T; N],
}

pub fn zip_array<T: Iterator, const N: usize>(array: [T; N]) -> ZipArray<T, N> {
    ZipArray { array }
}

impl<T: Iterator, const N: usize> Iterator for ZipArray<T, N> {
    type Item = [T::Item; N];

    fn next(&mut self) -> Option<Self::Item> {
        self.array.each_mut().try_map(|i| i.next())
    }
}
```

As well, additional output information, such as the actual computed relative errors of approximations, as well as verification of the correctly chosen k values, can be found below:

```
Goal error = 0.1
Smallest k for layer 0 =   6  Actual error = 0.083742  Error for k-1: 0.100988
Smallest k for layer 1 =   7  Actual error = 0.097234  Error for k-1: 0.117101
Smallest k for layer 2 =  10  Actual error = 0.097061  Error for k-1: 0.100765
Goal error = 0.05
Smallest k for layer 0 =  12  Actual error = 0.048465  Error for k-1: 0.051466
Smallest k for layer 1 =  17  Actual error = 0.049079  Error for k-1: 0.050994
Smallest k for layer 2 =  23  Actual error = 0.049735  Error for k-1: 0.052005
Goal error = 0.01
Smallest k for layer 0 =  66  Actual error = 0.009872  Error for k-1: 0.010231
Smallest k for layer 1 =  89  Actual error = 0.009900  Error for k-1: 0.010012
Smallest k for layer 2 = 119  Actual error = 0.009810  Error for k-1: 0.010157
Err 50 approximation for channel 1 k = 1
Err 50 approximation for channel 2 k = 1
Err 50 approximation for channel 3 k = 1
```