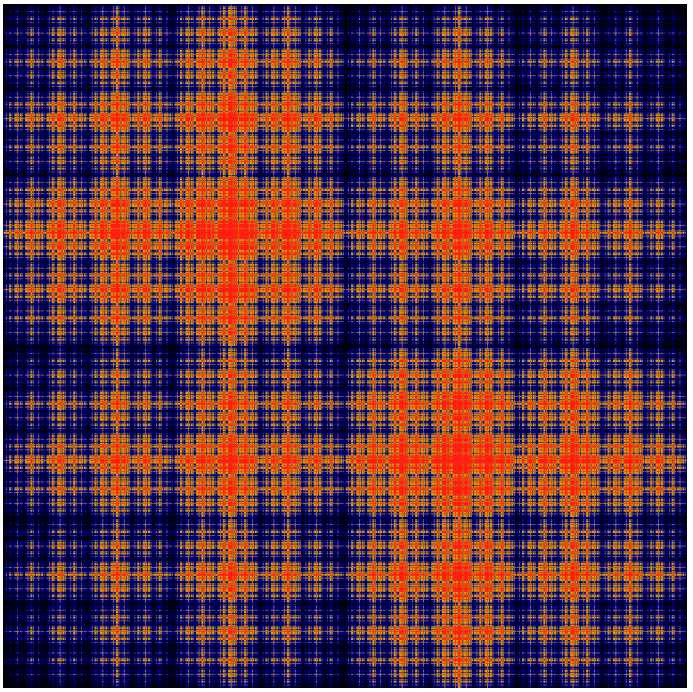
Applied Mathematics
and
Computing

Volume I



List of Contributors

J. Humpherys
Brigham Young University

J. Webb
Brigham Young University

R. Murray
Brigham Young University

J. West
University of Michigan

R. Grout
Brigham Young University

K. Finlinson
Brigham Young University

A. Zaitzeff
Brigham Young University

Preface

This lab manual is designed to accompany the textbook *Foundations of Applied Mathematics* by Dr. J. Humpherys.

Lab 16

Applications: Image Compression (SVD)

Lesson Objective: *Explore the SVD as a method of image compression*

The singular value decomposition is very useful. In this lab, we are going to explore how the SVD can be used to compress image data. Recall that the SVD is a decomposition of an $m \times n$ matrix A of rank r into the product $A = U\Sigma V^H$, where U and V are unitary matrices having dimensions $m \times m$ and $n \times n$, respectively, and Σ is an $m \times n$ diagonal matrix

$$\Sigma = \text{diag}(\sigma_1, \sigma_2, \ldots, \sigma_r, 0, \ldots, 0)$$

where $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r > 0$ are the singular values of A. Upon closer inspection, we can write

$$U = \begin{pmatrix} U_1 & U_2 \end{pmatrix}, \quad \Sigma = \begin{pmatrix} \Sigma_r & 0 \\ 0 & 0 \end{pmatrix}, \quad V = \begin{pmatrix} V_1 & V_2 \end{pmatrix},$$

where U_1 and V_1 have dimensions $m \times r$ and $n \times r$ respectively and Σ_r is the $r \times r$ diagonal matrix of (nonzero) singular values. Multiplying this out yields the reduced form of the SVD

$$A = \begin{pmatrix} U_1 & U_2 \end{pmatrix} \begin{pmatrix} \Sigma_r & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} V_1^H \\ V_2^H \end{pmatrix} = U_1 \Sigma_r V_1^H$$

Low rank data storage

If the rank of a given matrix is significantly smaller than its dimensions, the reduced form of the SVD offers a way to store A with less memory. Without the SVD, an $m \times n$ matrix requires storing $m * n$ values. By decomposing the original matrix into the SVD reduced form, U_1, Σ_r and V_1 together require $(m * r) + r + (n * r)$ values. Thus if r is much smaller than both m and n, we can obtain considerable efficiency. For example, suppose $m = 100$, $n = 200$ and $r = 20$. Then the original

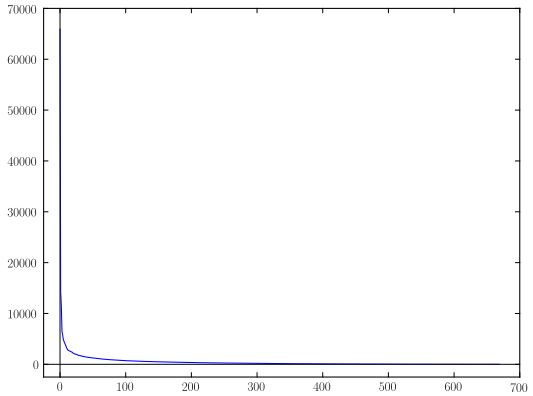
matrix would require storing $20,000$ values whereas the reduces form of the SVD only requires storing 6020 values.

Low rank approximation

The reduced form of the SVD also provides a way to approximate a matrix with another one of lower rank. This idea is used in many areas of applied mathematics including signal processing, statistics, semantic indexing (search engines), and control theory. If we are given a matrix A of rank r, we can find an approximate matrix $\widehat{A}$ of rank $s < r$ by taking the SVD of A and setting all of its singular values after σ_s to zero, that is,

$$\Sigma_{\widehat{A}} = \sigma_1, \sigma_2, \ldots, \sigma_s, \sigma_{s+1} = 0, \ldots, \sigma_r = 0$$

and then multiplying the matrix back together again. The more singular values we keep, the closer our approximation is to A. The number of singular values we decide to preserve depends on how close of an approximation we need and what our size requirements are for U_1, $\Sigma_{\widehat{A}}$, and V_1. Try plotting the the singular values. We have plotted the singular values to the image below. Matrix rank is on the x-axis and the eigenvalues are the y-axis. Note that SVD orders the singluar values from greatest to least. The greatest eigenvalues contribute most to the image while the smallest eigenvalues hardly contribute anything to the final approximation. By looking at the graph we can have a rough idea of how many singular values we need to preserve to have a good approximation of A. The matrix rank of the image below is 670. However, as the plot shows, we could easily approximate the image using only the first half of the singular values.




```
: import scipy as sp
: import numpy.linalg as nla
: A = sp.array
    ([[1,1,3,4],[5,4,3,7],[9,10,10,12],[13,14,15,16],[17,18,19,20]])
: nla.matrix_rank(A)
: U,s,Vt = nla.svd(A)
: S = sp.diag(s)
: Ahat = sp.dot(sp.dot(U[:,0:3], S[0:3,0:3]), Vt[0:3,:])
: nla.matrix_rank(Ahat)
: nla.norm(A)-nla.norm(Ahat)
```

Note that $\widehat{A}$ is "close" to the original matrix A, but that its rank is 3 instead of 4.

Application to Imaging

Enter the following into IPython (note that any image you might have will work):

```
: import matplotlib.pyplot as plt
: X = sp.misc.imread('fingerprint.png')[:,:,0].astype(float)
: X.nbytes       #number of bytes needed to store X
: sp.misc.imshow(X)
```

Computing the SVD of your image is simple. Remember to make the singluar values a diagonal matrix before multiplying.

```
: U,s,Vt = la.svd(X)
: S = sp.diag(s)
```

In the next code block, n repsents the desired rank of the output.

```
: n=50
: u1, s1, vt1 = U[:,0:n], S[0:n,0:n], Vt[0:n,:]
: Xhat = sp.dot(sp.dot(u1, s1), vt1)
: (u1.nbytes+sp.diag(s1).nbytes+vt1.nbytes) - X.nbytes    #should be
    negative
```

```
: sp.misc.imshow(Xhat)
```

Problem 1 A law enforcement agency has been needing to efficiently store over 50,000 fingerprints. They have decided to use an SVD based compression algorithm. Your job is to try several parameters for the SVD algorithm and recommend those parameters that retain the highest quality but compress the most. There should be no smearing or blocking in reconstructed final image and fingerprint detail must be retained (otherwise the fingerprint is worthless). As part of your recommendation, calculate how much memory would be needed on average to store each compressed fingerprint. Expand your results to say how much space could be saved if the entire database of fingerprints were compressed using your algorithm.