Last time: Cholesky factorization, QR factorization

Today: how to compute the QR factorization, the Singular Value Decomposition

Note: the final project is due at 11:59pm on Thursday December 13.\(^1\) Since submission is done online, there is no requirement that you be physically present.

---

\(^1\)All Harvard courses have a final exam group assigned to them. For courses with a final project instead, Harvard recommends the deadline be on the same date as when the exam would have been scheduled. AM205 is in the FAS04_C group. Dec 13 is based on the exam timetable that was recently posted.
QR Factorization

A matrix $A \in \mathbb{R}^{m \times n}$, $m \geq n$, can be factorized into

$$ A = QR $$

where

- $Q \in \mathbb{R}^{m \times m}$ is orthogonal
- $R \equiv \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix} \in \mathbb{R}^{m \times n}$
- $\hat{R} \in \mathbb{R}^{n \times n}$ is upper-triangular

QR is very good for solving overdetermined linear least-squares problems, $Ax \approx b$.

\[ ^2 \text{QR can also be used to solve a square system } Ax = b, \text{ but requires } \sim 2 \times \text{ as many operations as Gaussian elimination hence not the standard choice} \]
To see why, consider the 2-norm of the least squares residual:

\[ \| r(x) \|_2^2 = \| b - Ax \|_2^2 = \| b - Q \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix} x \|_2^2 \]

\[ = \| Q^T \left( b - Q \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix} x \right) \|_2^2 \]

\[ = \| Q^T b - \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix} x \|_2^2 \]

(We used the fact that \( \| Q^T z \|_2 = \| z \|_2 \) in the second line)
Then, let $Q^T b = [c_1, c_2]^T$ where $c_1 \in \mathbb{R}^n$, $c_2 \in \mathbb{R}^{m-n}$, so that

$$
\| r(x) \|_2^2 = \| c_1 - \hat{R}x \|_2^2 + \| c_2 \|_2^2
$$

**Question:** Based on this expression, how do we minimize $\| r(x) \|_2$?
Answer: We can’t influence the second term, $\|c_2\|_2^2$, since it doesn’t contain an $x$

Hence we minimize $\|r(x)\|_2^2$ by making the first term zero

That is, we solve the $n \times n$ triangular system $\hat{R}x = c_1$ — this what Python does in its `lstsq` function for solving least squares

Also, this tells us that $\min_{x \in \mathbb{R}^n} \|r(x)\|_2 = \|c_2\|_2$
QR Factorization

Recall that solving linear least-squares via the normal equations requires solving a system with the matrix $A^T A$

But using the normal equations directly is problematic since $\text{cond}(A^T A) = \text{cond}(A)^2$ (this is a consequence of the SVD, which we’ll cover soon)

The QR approach avoids this condition-number-squaring effect and is much more \textit{numerically stable}!
How do we compute the QR Factorization?

There are three main methods

- Gram–Schmidt Orthogonalization
- Householder Triangularization
- Givens Rotations

We will cover Gram–Schmidt and Givens rotations in class.
Gram–Schmidt Orthogonalization

Suppose $A \in \mathbb{R}^{m \times n}$, $m \geq n$

One way to picture the QR factorization is to construct a sequence of orthonormal vectors $q_1, q_2, \ldots$ such that

$$\text{span}\{q_1, q_2, \ldots, q_j\} = \text{span}\{a(:,1), a(:,2), \ldots, a(:,j)\}, \quad j = 1, \ldots, n$$

We seek coefficients $r_{ij}$ such that

$$a(:,1) = r_{11}q_1,$$
$$a(:,2) = r_{12}q_1 + r_{22}q_2,$$
$$\vdots$$
$$a(:,n) = r_{1n}q_1 + r_{2n}q_2 + \cdots + r_{nn}q_n.$$

This can be done via the Gram–Schmidt process, as we’ll discuss shortly.
Gram–Schmidt Orthogonalization

In matrix form we have:

\[
\begin{bmatrix}
a(:,1) & a(:,2) & \cdots & a(:,n)
\end{bmatrix}
= \begin{bmatrix}
q_1 & q_2 & \cdots & q_n
\end{bmatrix}
\begin{bmatrix}
r_{11} & r_{12} & \cdots & r_{1n} \\
r_{22} & r_{22} & \cdots & r_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
r_{nn} & & & r_{nn}
\end{bmatrix}
\]

This gives \( A = \hat{Q}\hat{R} \) for \( \hat{Q} \in \mathbb{R}^{m\times n}, \hat{R} \in \mathbb{R}^{n\times n} \)

This is called the reduced QR factorization of \( A \), which is slightly different from the definition we gave earlier.

Note that for \( m > n \), \( \hat{Q}^T\hat{Q} = I \), but \( \hat{Q}\hat{Q}^T \neq I \) (the latter is why the full QR is sometimes nice).
The full QR factorization (defined earlier)

\[ A = QR \]

is obtained by appending \( m - n \) arbitrary orthonormal columns to \( \hat{Q} \) to make it an \( m \times m \) orthogonal matrix.

We also need to append rows of zeros to \( \hat{R} \) to “silence” the last \( m - n \) columns of \( Q \), to obtain

\[ R = \begin{bmatrix} \hat{R} \\ 0 \end{bmatrix} \]
Full vs Reduced QR Factorization

Full QR

\[ A = Q \]

Reduced QR

\[ A = \hat{Q} \hat{R} \]
Exercise: Show that the linear least-squares solution is given by \( \hat{R}x = \hat{Q}^T b \) by plugging \( A = \hat{Q}\hat{R} \) into the Normal Equations

This is equivalent to the least-squares result we showed earlier using the full QR factorization, since \( c_1 = \hat{Q}^T b \)
In Python, `numpy.linalg.qr` gives the reduced QR factorization by default.

```python
>>> import numpy as np
>>> a = np.random.random((5, 3))
>>> q, r = np.linalg.qr(a)
```

```
>>> q
array([[-0.58479903,  0.18604305, -0.21857883],
       [-0.59514318, -0.34033765, -0.23588693],
       [-0.26381403,  0.14702842, -0.47775682],
       [-0.39324594, -0.43393772,  0.70189805],
       [-0.28208948,  0.79977432,  0.41912791]])
```

```
>>> r
array([[ 1.50223926, -1.44239112, -1.0813288 ],
       [ 0.     ,  0.49087707,  0.4207912 ],
       [ 0.     , -0.     ,  0.65436304]])
```
Full versus Reduced QR Factorization

In Python, supplying the mode='complete' option gives the complete QR factorization

Python 2.7.10 (default, Feb 7 2017, 00:08:15)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.34)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> a=np.random.random((5,3))
>>> (q,r)=np.linalg.qr(a,mode='complete')
>>> q
array([[ -0.58479903,  0.18604305, -0.21857883, -0.18819304, -0.73498623],
       [ -0.59514318, -0.34033765, -0.23588693, -0.40072023,  0.56013886],
       [ -0.26381403,  0.14702842, -0.47775682,  0.80433265,  0.18325448],
       [ -0.39324594, -0.43393772,  0.70189805,  0.39138159, -0.10590212],
       [ -0.28208948,  0.79977432,  0.41912791, -0.06225843,  0.31818586]])
>>> r
array([[ 1.50223926,  1.44239112, -1.0813288 ],
        [ 0.29074706,  0.49087707,  0.4207912 ],
        [ 0.12545186, -0. ,  0.65436304],
        [ 0.17686716,  0. , -0.29276124],
        [ 0. , -0. , -0. ]])
Gram–Schmidt Orthogonalization

Returning to the Gram–Schmidt process, how do we compute the \( q_i, \ i = 1, \ldots, n \)?

In the \( j \)th step, find a unit vector \( q_j \in \text{span}\{a(:,1), a(:,2), \ldots, a(:,j)\} \) that is orthogonal to \( \text{span}\{q_1, q_n, \ldots, q_{j-1}\} \)

We set

\[
v_j \equiv a(:,j) - (q_1^T a(:,j))q_1 - \cdots - (q_{j-1}^T a(:,j))q_{j-1},
\]

and then \( q_j \equiv v_j/\|v_j\|_2 \) satisfies our requirements

We can now determine the required values of \( r_{ij} \)
Gram–Schmidt Orthogonalization

We then write our set of equations for the $q_i$ as

\[
q_1 = \frac{a(:,1)}{r_{11}},
\]

\[
q_2 = \frac{a(:,2) - r_{12}q_1}{r_{22}},
\]

\[
\vdots
\]

\[
q_n = \frac{a(:,n) - \sum_{i=1}^{n-1} r_{in}q_i}{r_{nn}}.
\]

Then from the definition of $q_j$, we see that

\[
 r_{ij} = q_i^T a(:,j), \quad i \neq j
\]

\[
|r_{jj}| = \|a(:,j) - \sum_{i=1}^{j-1} r_{ij}q_i\|_2
\]

The sign of $r_{jj}$ is not determined uniquely, e.g. we could choose $r_{jj} > 0$ for each $j$. 
The Gram–Schmidt algorithm we have described is provided in the pseudocode below.

```
1: for j = 1 : n do
2:   v_j = a(:,j)
3:   for i = 1 : j - 1 do
4:     r_{ij} = q_i^T a(:,j)
5:     v_j = v_j - r_{ij} q_i
6:   end for
7:   r_{jj} = \|v_j\|_2
8:   q_j = v_j / r_{jj}
9: end for
```

This is referred to the classical Gram–Schmidt (CGS) method.
The only way the Gram–Schmidt process can fail is if
\[ |r_{jj}| = \|v_j\|_2 = 0 \] for some \( j \)

This can only happen if \( a(:,j) = \sum_{i=1}^{j-1} r_{ij} q_i \) for some \( j \), i.e. if
\[ a(:,j) \in \text{span}\{q_1, q_n, \ldots, q_{j-1}\} = \text{span}\{a(:,1), a(:,2), \ldots, a(:,j-1)\} \]

This means that columns of \( A \) are linearly dependent

Therefore, Gram–Schmidt fails \( \implies \) cols. of \( A \) linearly dependent
Gram–Schmidt Orthogonalization

Equivalently, by contrapositive: cols. of $A$ linearly independent $\implies$ Gram–Schmidt succeeds

**Theorem:** Every $A \in \mathbb{R}^{m \times n} (m \geq n)$ of full rank has a unique reduced QR factorization $A = \hat{Q} \hat{R}$ with $r_{ii} > 0$

The only non-uniqueness in the Gram–Schmidt process was in the sign of $r_{ii}$, hence $\hat{Q} \hat{R}$ is unique if $r_{ii} > 0$
Theorem: Every $A \in \mathbb{R}^{m \times n}(m \geq n)$ has a full QR factorization.

Case 1: $A$ has full rank

- We compute the reduced QR factorization from above
- To make $Q$ square we pad $\hat{Q}$ with $m - n$ arbitrary orthonormal columns
- We also pad $\hat{R}$ with $m - n$ rows of zeros to get $R$

Case 2: $A$ doesn’t have full rank

- At some point in computing the reduced QR factorization, we encounter $\|v_j\|_2 = 0$
- At this point we pick an arbitrary $q_j$ orthogonal to $\text{span}\{q_1, q_2, \ldots, q_{j-1}\}$ and then proceed as in Case 1
Modified Gram–Schmidt Process

The classical Gram–Schmidt process is **numerically unstable!**
(sensitive to rounding error, orthogonality of the $q_j$ degrades)

The algorithm can be reformulated to give the **modified Gram–Schmidt process**, which is numerically more robust

**Key idea:** when each new $q_j$ is computed, orthogonalize each remaining column of $A$ against it
Modified Gram–Schmidt Process

Modified Gram–Schmidt (MGS):

1: **for** $i = 1 : n$ **do**
2: $v_i = a(:,i)$
3: **end for**
4: **for** $i = 1 : n$ **do**
5: $r_{ii} = \|v_i\|_2$
6: $q_i = v_i / r_{ii}$
7: **for** $j = i + 1 : n$ **do**
8: $r_{ij} = q_i^T v_j$
9: $v_j = v_j - r_{ij} q_i$
10: **end for**
11: **end for**
Modified Gram–Schmidt Process

Key difference between MGS and CGS:

- In CGS we compute orthogonalization coefficients $r_{ij}$ wrt the “raw” vector $a(:,j)$
- In MGS we remove components of $a(:,j)$ in $\text{span}\{q_1, q_2, \ldots, q_{i-1}\}$ before computing $r_{ij}$

This makes no difference mathematically: In exact arithmetic components in $\text{span}\{q_1, q_2, \ldots, q_{i-1}\}$ are annihilated by $q_i^T$

But in practice it reduces degradation of orthogonality of the $q_j$ $\implies$ superior numerical stability of MGS over CGS
Operation Count

Work in MGS is dominated by lines 8 and 9, the innermost loop:

\[ r_{ij} = q_i^T v_j \]
\[ v_j = v_j - r_{ij} q_i \]

First line requires \( m \) multiplications, \( m - 1 \) additions; second line requires \( m \) multiplications, \( m \) subtractions

Hence \( \sim 4m \) operations per single inner iteration

Hence total number of operations is asymptotic to

\[ \sum_{i=1}^{n} \sum_{j=i+1}^{n} 4m \sim 4m \sum_{i=1}^{n} i \sim 2mn^2 \]
Alternative QR computation methods

The QR factorization can also be computed using Householder triangularization and Givens rotations.

Both methods take the approach of applying a sequence of orthogonal matrices $Q_1, Q_2, Q_3, \ldots$ to the matrix that successively remove terms below the diagonal (similar to the method employed by the LU factorization).

We will discuss Givens rotations.
A Givens rotation

For \( i < j \) and an angle \( \theta \), the elements of the \( m \times m \) Givens rotation matrix \( G(i, j, \theta) \) are

\[
\begin{align*}
g_{ii} &= c, & g_{jj} &= c, & g_{ij} &= s, & g_{ji} &= -s, \\
g_{kk} &= 1 & & \text{for } k \neq i, j, \\
g_{kl} &= 0 & & \text{otherwise,}
\end{align*}
\]  

(1)

where \( c = \cos \theta \) and \( s = \sin \theta \).
A Givens rotation

Hence the matrix has the form

\[ G(i, j, \theta) = \begin{pmatrix}
1 & \ldots & 0 & \ldots & 0 & \ldots & 0 \\
\vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\
0 & \ldots & c & \ldots & s & \ldots & 0 \\
\vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\
0 & \ldots & -s & \ldots & c & \ldots & 0 \\
\vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\
0 & \ldots & 0 & \ldots & 0 & \ldots & 1
\end{pmatrix} \]

It applies a rotation within the space spanned by the \( i \)th and \( j \)th coordinates.
Effect of a Givens rotation

Consider a $m \times n$ rectangular matrix $A$ where $m \geq n$

Suppose that $a_1$ and $a_2$ are in the $i$th and $j$th positions in a particular column of $A$. Assume at least one $a_i$ is non-zero.

Restricting to just $i$th and $j$th dimensions, a Givens rotation $G(i, j, \theta)$ for a particular angle $\theta$ can be applied so that

$$
\begin{pmatrix}
c & s \\
-s & c
\end{pmatrix}
\begin{pmatrix}
a_1 \\
a_2
\end{pmatrix} = \begin{pmatrix}
\alpha \\
0
\end{pmatrix},
$$

where $\alpha$ is non-zero, and the $j$th component is eliminated.
Stable computation

\( \alpha \) is given by \( \sqrt{a_1^2 + a_2^2} \). We could compute

\[
c = a_1/\sqrt{a_1^2 + a_2^2}, \quad s = a_2/\sqrt{a_1^2 + a_2^2}
\]

but this is susceptible to underflow/overflow if \( \alpha \) is very small.

A better procedure is as follows:

- if \(|a_1| > |a_2|\), set \( t = \tan \theta = a_2/a_1 \), and hence
  \[
c = \frac{1}{\sqrt{1+t^2}}, \ s = ct.
\]

- if \(|a_2| \geq |a_1|\), set \( \tau = \cot \theta = a_1/a_2 \), and hence
  \[
s = \frac{1}{\sqrt{1+\tau^2}}, \ c = s\tau.
\]
Givens rotation algorithm

To perform the Givens procedure on a dense \( m \times n \) rectangular matrix \( A \) where \( m \geq n \), the following algorithm can be used:

\[
\begin{align*}
1: \quad R &= A, \quad Q = I \\
2: \quad \textbf{for } k = 1 : n \textbf{ do} \\
3: \quad \quad \textbf{for } j = m : k + 1 \textbf{ do} \\
4: \quad \quad \quad \text{Construct } G = G(j - 1, j, \theta) \text{ to eliminate } a_{jk} \\
5: \quad \quad \quad R &= GR \\
6: \quad \quad \quad Q &= QG^T \\
7: \quad \quad \textbf{end for} \\
8: \quad \textbf{end for}
\end{align*}
\]
Givens rotation advantages

In general, for dense matrices, Givens rotations are not as efficient as the other two approaches (Gram–Schmidt and Householder).

However, they are advantageous for sparse matrices, since non-zero entries can be eliminated one-by-one. They are also amenable to parallelization. Consider the $6 \times 6$ matrix:

$$
\begin{pmatrix}
\times & \times & \times & \times & \times & \times \\
5 & \times & \times & \times & \times & \times \\
4 & 6 & \times & \times & \times & \times \\
3 & 5 & 7 & \times & \times & \times \\
2 & 4 & 6 & 8 & \times & \times \\
1 & 3 & 5 & 7 & 9 & \times \\
\end{pmatrix}
$$

The numbers represent the steps at which a particular matrix entry can be eliminated. *e.g.* on step 3, elements $(4, 1)$ and $(6, 2)$ can be eliminated concurrently using $G(3, 4, \theta_a)$ and $G(5, 6, \theta_b)$, respectively, since these two matrices operate on different rows.
The Singular Value Decomposition (SVD) is a very useful matrix factorization.

Motivation for SVD: image of the unit sphere, $S$, from any $m \times n$ matrix is a hyperellipse.

A hyperellipse is obtained by stretching the unit sphere in $\mathbb{R}^m$ by factors $\sigma_1, \ldots, \sigma_m$ in orthogonal directions $u_1, \ldots, u_m$. 
Singular Value Decomposition

For $A \in \mathbb{R}^{2 \times 2}$, we have
Singular Value Decomposition

Based on this picture, we make some definitions:

- **Singular values**: \( \sigma_1, \sigma_2, \ldots, \sigma_n \geq 0 \) (we typically assume \( \sigma_1 \geq \sigma_2 \geq \ldots \))

- **Left singular vectors**: \( \{u_1, u_2, \ldots, u_n\} \), unit vectors in directions of principal semiaxes of \( AS \)

- **Right singular vectors**: \( \{v_1, v_2, \ldots, v_n\} \), preimages of the \( u_i \) so that \( Av_i = \sigma_i u_i, \ i = 1, \ldots, n \)

(The names “left” and “right” come from the formula for the SVD below)
Singular Value Decomposition

The key equation above is that

$$Av_i = \sigma_i u_i, \quad i = 1, \ldots, n$$

Writing this out in matrix form we get

\[
\begin{bmatrix}
    \vdots \\
    A \\
    \vdots \\
\end{bmatrix}
\begin{bmatrix}
    v_1 \\
    v_2 \\
    \vdots \\
    v_n \\
\end{bmatrix} =
\begin{bmatrix}
    \vdots \\
    u_1 \\
    u_2 \\
    \vdots \\
    u_n \\
\end{bmatrix}
\begin{bmatrix}
    \sigma_1 \\
    \sigma_2 \\
    \vdots \\
    \sigma_n \\
\end{bmatrix}
\]

Or more compactly:

$$AV = \hat{U}\hat{\Sigma}$$
Singular Value Decomposition

Here

- \( \hat{\Sigma} \in \mathbb{R}^{n \times n} \) is diagonal with non-negative, real entries
- \( \hat{U} \in \mathbb{R}^{m \times n} \) with orthonormal columns
- \( V \in \mathbb{R}^{n \times n} \) with orthonormal columns

Therefore \( V \) is an orthogonal matrix \( (V^T V = VV^T = I) \), so that we have the reduced SVD for \( A \in \mathbb{R}^{m \times n} \):

\[
A = \hat{U} \hat{\Sigma} V^T
\]
Singular Value Decomposition

Just as with QR, we can pad the columns of $\hat{U}$ with $m - n$ arbitrary orthogonal vectors to obtain $U \in \mathbb{R}^{m \times m}$

We then need to “silence” these arbitrary columns by adding rows of zeros to $\hat{\Sigma}$ to obtain $\Sigma$

This gives the full SVD for $A \in \mathbb{R}^{m \times n}$:

$$A = U \Sigma V^T$$
Full vs Reduced SVD

Full SVD

\[ A = U \Sigma V^T \]

Reduced SVD

\[ A = \hat{U} \hat{\Sigma} V^T \]
Theorem: Every matrix $A \in \mathbb{R}^{m \times n}$ has a full singular value decomposition. Furthermore:

- The $\sigma_j$ are uniquely determined
- If $A$ is square and the $\sigma_j$ are distinct, the $\{u_j\}$ and $\{v_j\}$ are uniquely determined up to sign
Singular Value Decomposition

This theorem justifies the statement that the image of the unit sphere under any $m \times n$ matrix is a hyperellipse.

Consider $A = U \Sigma V^T$ (full SVD) applied to the unit sphere, $S$, in $\mathbb{R}^n$:

1. The orthogonal map $V^T$ preserves $S$
2. $\Sigma$ stretches $S$ into a hyperellipse aligned with the canonical axes $e_j$
3. $U$ rotates or reflects the hyperellipse without changing its shape
SVD in Python

Python's `numpy.linalg.svd` function computes the full SVD of a matrix.

```python
>>> import numpy as np
>>> a=np.random.random((4,2))
>>> (u,s,v)=np.linalg.svd(a)
>>> u
array([[-0.38627868, 0.3967265 , -0.44444737, -0.70417569],
       [-0.4748846 , -0.845594 , -0.23412286, -0.06813139],
       [-0.47511682, 0.05263149, 0.84419597, -0.24254299],
       [-0.63208972, 0.35328288, -0.18704595, 0.663828 ]])
>>> s
array([ 1.56149162, 0.24419604])
>>> v
array([[-0.67766849, -0.73536754],
       [-0.73536754, 0.67766849]])
```
SVD in Python

The `full_matrices=0` option computes the reduced SVD

Python 2.7.8 (default, Jul 13 2014, 17:11:32)
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.40)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> a=np.random.random((4,2))
>>> (u,s,v)=np.linalg.svd(a,full_matrices=0)
>>> u
array([[-0.38627868,  0.3967265 ],
       [-0.4748846 , -0.845594  ],
       [-0.47511682,  0.05263149],
       [-0.63208972,  0.35328288]])

>>> s
array([ 1.56149162,  0.24419604])

>>> v
array([[-0.67766849, -0.73536754],
       [-0.73536754,  0.67766849]])
Matrix Properties via the SVD

- The rank of $A$ is $r$, the number of nonzero singular values.$^3$

**Proof:** In the full SVD $A = U \Sigma V^T$, $U$ and $V^T$ have full rank, hence it follows from linear algebra that $\text{rank}(A) = \text{rank}(\Sigma)$

- $\text{image}(A) = \text{span}\{u_1, \ldots, u_r\}$ and $\text{null}(A) = \text{span}\{v_{r+1}, \ldots, v_n\}$

**Proof:** This follows from $A = U \Sigma V^T$ and

\[
\begin{align*}
\text{image}(\Sigma) &= \text{span}\{e_1, \ldots, e_r\} \in \mathbb{R}^m \\
\text{null}(\Sigma) &= \text{span}\{e_{r+1}, \ldots, e_n\} \in \mathbb{R}^n
\end{align*}
\]

---

$^3$This also gives us a good way to define rank in finite precision: the number of singular values larger than some (small) tolerance
Matrix Properties via the SVD

• $\|A\|_2 = \sigma_1$

Proof: Recall that $\|A\|_2 \equiv \max_{\|v\|_2=1} \|Av\|_2$. Geometrically, we see that $\|Av\|_2$ is maximized if $v = v_1$ and $Av = \sigma_1 u_1$.

• The singular values of $A$ are the square roots of the eigenvalues of $A^T A$ or $AA^T$

Proof: (Analogous for $AA^T$)

$$A^T A = (U\Sigma V^T)^T (U\Sigma V^T) = V\Sigma U^T U\Sigma V^T = V(\Sigma^T \Sigma) V^T,$$

hence $(A^T A) V = V(\Sigma^T \Sigma)$, or $(A^T A)v(:,j) = \sigma_j^2 v(:,j)$