CS170A — Mathematical Modeling & Methods for Computer Science
HW#2 *SVD, LSQ, PCA, FFT*
Due: 11:55 pm Wednesday February 10, 2016

D. Stott Parker, Yunzhong He, Luis Angel Larios-Cardenas, Yun Lu

Using CCLE, please upload PDF (with enough information to demonstrate that programs are working).

1. **Singular Value Decomposition (20 points)**
One of the most important and useful results in computational linear algebra is that any $n \times p$ matrix A has a *Singular Value Decomposition (SVD)* $A = U\,S\,V'$ where U and V are unitary matrices and, assuming $n > p$, S is a diagonal matrix of nonnegative real *singular values* $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_p \geq 0$:

$$A = \begin{pmatrix} \mathbf{u}_1 & \cdots & \mathbf{u}_k & \cdots & \mathbf{u}_n \end{pmatrix} \begin{pmatrix} \sigma_1 & & & & \\ & \ddots & & & \\ & & \sigma_k & & \\ & & & \ddots & \\ & & & & \sigma_p \end{pmatrix} \begin{pmatrix} \mathbf{v}_1 & \cdots & \mathbf{v}_k & \cdots & \mathbf{v}_p \end{pmatrix}'.$$

The singular values are a lot like eigenvalues, but they are always real values, and they are never negative.

In Matlab, the command `[U,S,V] = svd(A)` finds the SVD of `A`.

(a) Implement the SVD as a function

```
[U,S,V] = mysvd(A)   %%  A can be any REAL rectangular matrix
```

Your implementation can build upon the Jacobi eigendecomposition procedure described in the course notes, but however you do this, your implementation of the SVD must be self-contained, without using builtin functions other than trigonometric functions (*sin*, *cos*, *arctan*, ...).

Important: your function must return singular values in descending order.

For each of the following five matrices, find the SVD using your implementation:

```
C = gallery('chow', 8)      %  also available as Chow8.csv
F = hadamard(8)             %  also available as Hadamard8.csv
H = hilb(8)                 %  also available as Hilbert8.csv
P = pascal(8)               %  also available as Pascal8.csv
```

(b) The condition number of a matrix A is the ratio of its largest to smallest singular values: $\kappa(A) = \sigma_1(A) \,/\, \sigma_n(A)$. Using the singular values returned by your program, find the condition number of each matrix.

(c) The script `imagesvd.m` from `www.mathworks.com/moler/ncmfilelist.html` shows how the SVD can be used to compress images. `imagesvd` works by using the *rank-k SVD approximation* of an image matrix A to keep only the first k columns in U and V, and keep only the upper left $k \times k$ portion of S:

$$A^{(k)} = U^{(k)}\,S^{(k)}\left(V^{(k)}\right)'.$$

`imagesvd` displays $A^{(k)}$, where k is the slider value. The schematic of $A^{(k)}$ simply has some columns zeroed out:

$$\begin{pmatrix} \mathbf{u}_1 & \cdots & \mathbf{u}_k & \end{pmatrix} \begin{pmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_k \end{pmatrix} \begin{pmatrix} \mathbf{v}_1 & \cdots & \mathbf{v}_k & \end{pmatrix}' = \begin{pmatrix} \mathbf{u}_1 & \cdots & \mathbf{u}_k \end{pmatrix} \begin{pmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_k \end{pmatrix} \begin{pmatrix} \mathbf{v}_1 & \cdots & \mathbf{v}_k \end{pmatrix}'.$$

The error in this approximation is then the matrix difference:

$$(A - A^{(k)}) = U\,(S - S^{(k)})\,V'.$$

Display this error matrix as an image for $k = 5,\ 10,\ 20,\ 50$ for the 298×264 matrix A obtained from the grayscale (monochromatic, not RGB) image `Joseph_Fourier_1820.jpg` (also available as `Joseph_Fourier_1820.csv`).

To do this, e.g. `A = double(imread('Joseph_Fourier_1820.jpg'));` obtains a numeric matrix A from the image. The error matrix can then be displayed by first converting it to an image (with e.g., `uint8`) and then rendering it (with e.g., `imshow` or `image` or `imagesc`. In Python, `matplotlib.image` has these functions.)

2. **Rotations Using Quaternions (25 points)**

A quaternion is a four-tuple of real numbers $[s, x, y, z]$, or equivalently, $[s, \mathbf{v}]$, consisting of a scalar s and a three-dimensional vector $\mathbf{v} = [x, y, z]$. A quaternion can be used to represent both an orientation and a rotation since it encodes information abouth an axis and an angle. As a rotation, we can use quaternions to transform vectors and points in space; as an orientation, a quaternion can be used to interpolate the location of points relative to their initial state.

Before we implement quaternions, some of their basic math must be understood. In the equations that follow, $\cdot$ represents dot product, and $\times$ denotes cross product. Let $\mathbf{q}_1 = [s_1, \mathbf{v}_1]$ and $\mathbf{q}_2 = [s_2, \mathbf{v}_2]$ be two quaternions:

Addition : $\mathbf{q}_1 + \mathbf{q}_2 = [s_1 + s_2, \mathbf{v}_1 + \mathbf{v}_2]$
Product : $\mathbf{q}_1\mathbf{q}_2 = [s_1 s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2, s_1\mathbf{v}_2 + s_2\mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2]$
Multiplicative Identity : $[1, \mathbf{0}] = [1, 0, 0, 0]$ $[s, \mathbf{v}][1, \mathbf{0}] = [s, \mathbf{v}]$
Associativity : $(\mathbf{q}_1\mathbf{q}_2)\mathbf{q}_3 = \mathbf{q}_1(\mathbf{q}_2\mathbf{q}_3)$
Non Commutativity : $\mathbf{q}_1\mathbf{q}_2 \neq \mathbf{q}_2\mathbf{q}_1$
Norm : $||\mathbf{q}|| = \sqrt{s^2 + \mathbf{v} \cdot \mathbf{v}}$
Inverse : $\mathbf{q}^{-1} = \frac{1}{||\mathbf{q}^2||}[s, -\mathbf{v}]$ $\mathbf{q}^{-1}\mathbf{q} = \mathbf{q}\mathbf{q}^{-1} = [1, 0, 0, 0]$
Inverse Distribution : $(\mathbf{q}_1\mathbf{q}_2)^{-1} = \mathbf{q}_2^{-1}\mathbf{q}_1^{-1}$
Normalization : $\hat{\mathbf{q}} = \frac{\mathbf{q}}{||\mathbf{q}||}$

A *rotation* is represented in quaternion form by encoding axis-angle information. The quaternion

$$\text{Rot}_q = \mathbf{q} = (\cos(\theta/2), \sin(\theta/2)\widehat{\mathbf{v}})$$

represents a rotation by an angle, θ, around the unit axis $\widehat{\mathbf{v}} = (x, y, z)$. To rotate a vector, $\mathbf{u}$, using quaternion math, we represent the vector $\mathbf{u}$ as $(0, \mathbf{u})$, and the rotation as a *unit* quaternion $\mathbf{q}$. Thus

$$\mathbf{w} = \text{Rot}_q(\mathbf{u}) = \mathbf{q}\mathbf{u}\mathbf{q}^{-\mathbf{1}}$$

yields $\mathbf{w}$, which is the result of rotating $\mathbf{u}$ by an angle θ around the unit axis $\widehat{\mathbf{v}}$ (the last two are encoded in $\mathbf{q}$).

Compound rotations can be implemented by premultiplying the corresponding quaternions, similar to what is routinely done when rotations matrices are used:

$$\begin{aligned} \mathbf{w} = \text{Rot}_q(\text{Rot}_p(\mathbf{u})) &= \mathbf{q}(\mathbf{p}\mathbf{u}\mathbf{p}^{-\mathbf{1}})\mathbf{q}^{-1} \\ &= (\mathbf{q}\mathbf{p})\mathbf{u}(\mathbf{p}^{-1}\mathbf{q}^{-1}) \\ &= (\mathbf{q}\mathbf{p})\mathbf{u}(\mathbf{q}\mathbf{p})^{-1} \\ &= \text{Rot}_{qp}(\mathbf{u}) \end{aligned}$$

The latter represents a rotation of $\mathbf{u}$ by $\mathbf{p}$, followed by a rotation expressed by $\mathbf{q}$ (the rightmost transformation is always applied first).

Using the programming language of your choice:

(a) Create a function that implements the *quaternion multiplication* of 2 input quaternions $\mathbf{p}$ and $\mathbf{q}$. Your function has to return the product $\mathbf{r} = \mathbf{p}\mathbf{q}$, having all quaterions expressed as a 4-tuple (i.e. vector of four elements). What is $\mathbf{r} = \mathbf{p}\mathbf{q}$ when $\mathbf{p} = (3, 2, 5, 4)$ and $\mathbf{q} = (4, 5, 3, 1)$?

(b) Write a function to compute and return the *inverse quaternion* $\mathbf{q}^{-1}$ of the input quaternion $\mathbf{q}$. What is the inverse of $\mathbf{q} = (8, -2, 3, -1)$? What is the result of multiplying $||\mathbf{q}||$ by the inverse you just got?

(c) Write a function $\text{Rot}_q(\cdot)$ to build a *rotation quaternion*. Your function must receive as input the angle θ and the axis of rotation $\mathbf{v}$, and return the rotation quaternion $\mathbf{q}$. Recall that $\mathbf{q}$ should have *unit length* (that is, it should be $\widehat{\mathbf{q}}$ built with $\widehat{\mathbf{v}} = \mathbf{v}/||\mathbf{v}||$). What is the result of rotating the $\widehat{\mathbf{x}} = (1, 0, 0)$ axis by an angle of $\theta = \pi/6$ around the $\widehat{\mathbf{z}} = (0, 0, 1)$ axis? *(Note: Check your result with a conventional 2D rotation matrix transforming the x axis)*

(d) Use the function you wrote in the previous step to rotate the point $\mathbf{u} = (1, 2, 3)$ by 45 degrees ($\pi/4$) around the axis $\mathbf{v} = (1, 1, 1)$. Generate a 3D plot showing the original $\mathbf{u}$ vector, and its rotated version $\mathbf{w} = \text{Rot}_q(\mathbf{u})$. To aid in the visualization, display the rotated cartesian axes ($\widehat{\mathbf{x}}$, $\widehat{\mathbf{y}}$, and $\widehat{\mathbf{z}}$) in your 3D plot.

(e) Use the function you wrote in step 3 to show that a rotation given by a quaternion $\mathbf{q}$ is the **same** as the rotation given by $-\mathbf{q} = (-s, -\mathbf{v})$. That is, obtain $\mathbf{q}$ for a rotation of $\theta = \pi/3$ around the axis $\mathbf{v} = (1, 1, 1)$. Next, use $\mathbf{q}$ to rotate the 3 coordinate axes ($\widehat{\mathbf{x}}$, $\widehat{\mathbf{y}}$, and $\widehat{\mathbf{z}}$) and plot them. Then, let $\mathbf{p} = -\mathbf{q}$ be the **negated** $\mathbf{q}$. Finally, in a different plot, show again the 3 coordinate axes when rotated by $\mathbf{p}$.

3. **Least-Squared Faces (30 points)**
Besides curve fitting, we can also use least squares as a means of *image compression*. In this context, given a set of *basis* images, we can obtain an approximation to any image by solving the least square problem:

$$A\mathbf{x} = \mathbf{b}$$

where A is a matrix whose columns are the (vectorized) unit basis images, $\mathbf{b}$ is the (vectorized) original image, and $\mathbf{x}$ is the coefficients vector that minimizes the squared error between the reconstructed image, $A\mathbf{x}$, and the original image, $\mathbf{b}$.

Figure 1: A basis face

Generally, $\mathbf{x}$ will be much smaller than the size of $\mathbf{b}$. Thus, instead of dealing with the large, original image, we can use the $\mathbf{x}$ coefficients to represent $\mathbf{b}$ and save space.

Inside the homework *.zip file you will find a folder `faces`. There are **13 basis images** located under the directory `basis`, and a few other test images under the directory `tests`. All images have been preprocessed and are 148×100 pixels in size (height and width, respectively).

(a) Your first task is to build matrix A. Load all basis images and convert them to gray-scale (monochromatic). Let E_i be the i^{th} basis image, then

$$A = \left(\begin{array}{c|c|c|c} \widehat{\mathbf{e}}_1 & \widehat{\mathbf{e}}_2 & \cdots & \widehat{\mathbf{e}}_{13} \end{array} \right) \tag{1}$$

where $\widehat{\mathbf{e}}_i$ is the unit column vector version of E_i —that is, you have to reshape E_i into a vector $\mathbf{e}_i$ and normalize it: $\widehat{\mathbf{e}}_i = \mathbf{e}_i / ||\mathbf{e}_i||$.

(b) Load the test image `t2.jpg`, gray-scale it, and reshape it into the column vector $\mathbf{b}$. Make sure all intensity values are in the range of [0, 1].

(c) Solve for $\mathbf{x}$ in $A\mathbf{x} = \mathbf{b}$. Then, compute the reconstructed (or approximation) image $\mathbf{v} = A\mathbf{x}$. Again, scale $\mathbf{v}$ so that its intensity values are in the range of [0, 1]. Plot your reconstructed image and the original image `t2.jpg`. What is the squared error, $||\mathbf{v} - \mathbf{b}||_2^2$, of your approximation?

(d) What are the coefficients of vector $\mathbf{x}$? This vector is now the "compressed" version of our original image `t2.jpg`. Furthermore, $\mathbf{x}$ contains the *weights* of the linear cobination of the basis images, $\widehat{\mathbf{e}}_i$, that yields `t2.jpg`. Which basis images correspond to the top three largest absolute-valued coefficients in $\mathbf{x}$?

(e) Which two basis images have highest correlation? (Determine this by computing the correlation matrix of A.)

(f) What are the first two principal components of the correlation matrix of A?

(g) Viewing these two components as (x, y) coordinates, which two images (among the 13) have (x, y) positions that are closest to each other?

4. **Exponential Growth (15 points)**
In the files `IXIC.csv`, is the history of NASDAQ index (from `ichart.finance.yahoo.com/table.csv?s=ÎXIC`). The Matlab function `read_stock.m` can read and plot this index history for you.

We want to determine whether the growth of the Nasdaq over certain periods was exponential or polynomial. (A function $f(t)$ is called **exponential** (or **geometric**) if $f(t) = O(c^t)$ for some nonzero constant c, and **polynomial** if $f(t) = O(t^n)$ for some positive integer n.)

(a) Read in the file `IXIC.csv` file (e.g., using `[time ixic] = read_stock('IXIC.csv');`
The `time` vector this produces is a sequence of dates on which index values were obtained;
the `ixic` vector is the actual index value ('adjusted close').

(b) For this IXIC data, we consider two time intervals as sequences of x values:

(1) 1971-02-05 — 2000-03-09 (lines 2:7350 in the `IXIC.csv` file)
(2) 2009-03-10 — 2015-11-04 (lines 9611:11287 in the `IXIC.csv` file).

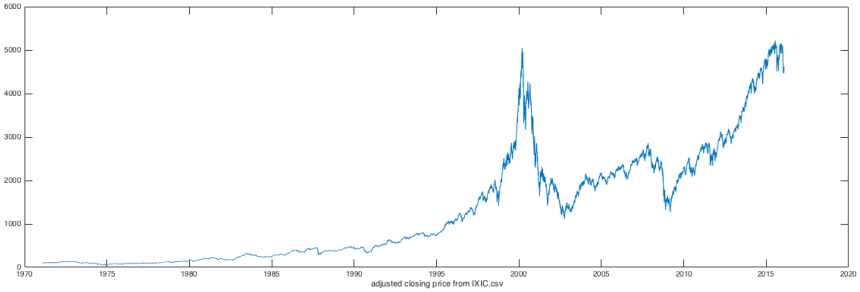
For each of these periods, fit two curves, where x is a sequence of time values and y is a sequence of ixic values:

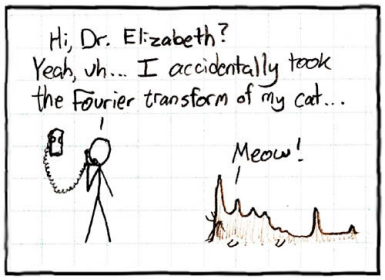
- find the coefficients of a degree-4 polynomial fit $y \sim \sum_{i=0}^{4} c_i x^i$.
- find the coefficients of an exponential fit $y \sim e^{c_0 + c_1 x}$ by fitting a linear model $\log(x) \sim c_0 + c_1 t$.
- Plot the the polynomial and exponential fits together along with the data, and include the coefficients and squared error in the title (e.g., with `title(sprintf(...))`).
- Determine whether a polynomial or exponential fit has lower squared error.

5. **Fourier Stock Analysis (10 points)**
You can download historical stock quotes on most U.S. securities from the same place as above. For example, you can download quotes for Tesla from finance.yahoo.com/q/hp?s=TSLA For this problem, however, we want you to do Fourier analysis on the NASDAQ data you analyzed above. Specifically, we want you to analyze the interval

2003-09-03 — 2008-09-02 (lines 8223:9481 in the `IXIC.csv` file)

— which is a history of 5 years' worth of data, between the two peaks in NASDAQ history:






```
[time ixic] = read_stock('IXIC.csv');

%% get the desired interval of NASDAQ quotes here...
desired_ixic  =     ..............
n = size(desired_ixic,1);
power_spectrum = abs(fft(desired_ixic)).^2;
log_power_spectrum = log(power_spectrum);   % power_spectrum values have large range, so we use a semilog plot

%% NOTE:  use day numbers (\verb"ixic" subscript values) as 'x' values --- not the 'time' values in IXIC.csv !!
frequencies = linspace(0, 1.0, n);
  %% stock quotes are made "once per day", so their frequency is  1/day.
  %% In the frequencies vector, the largest value 1.0 means stocks are sampled once per day.
  %% In other words, the frequency values have 1/day as their maximum.
plot(frequencies(2:floor(n/2)), log_power_spectrum(2:floor(n/2)));
  %% We plot only the first half of the data, since the second half is a mirror image;
  %% also we ignore the first Fourier coefficient, since it is just the sum of the input.
```

Compute the Fourier transform of this history. Remember that stocks are sampled at a frequency of 1/day (once per day), but there are only five business days per week, and with holidays and other exceptional situations altogether prices are sampled only on about 252 days per year. In other words, assume a sampling frequency of 1/day (once per day), and that 1/(252 days) = 1/year. (Remember that the far right value in the frequency range should be the sampling frequency, and for stocks this is once per day = 1/day.) This can matter since the stock market is sensitive to quarterly and annual reports.

First, in the power spectrum there is a modest spike at a frequency near `0.02` per day; find the exact frequency. Given that this frequency is in units of 1 per day, what monthly frequency (with 252/12 days per month) does it correspond to?