AMATH 582 Homework Three: Principal Component Analysis

Daniel W. Crews

February 24, 2021

Abstract

Measurements generically include superfluous components (from the perspective of an analyst at least). An old, efficient, and quite spectacular method to extract particular dynamics within a dataset is called principal component analysis (PCA). Basically, PCA is an eigendecomposition of the covariance matrix. This report utilizes PCA to analyze simple harmonic motion recorded on separate video cameras, assigning different oscillatory components to particular modal coordinates rather their motional ones.

1 Introduction and Overview

Here four experiments are analyzed of videos taken from three perspectives of a mass oscillating at the end of a string. The four experiments consisted of simple one-dimensional harmonic motion in gravity from the string tension, then introduction of an additional swinging pendulum motion, and thirdly also rotation of the oscillating mass. Finally data was collected with camera shake in order to introduce measurement noise. The measurements are composed of multiple relatively independent simultaneously occuring oscillations to demonstrate the mathemagical properties of the principal component analysis (PCA) method, which separates oscillatory modes by an eigendecomposition (thus, diagonalization) of the data covariance matrix.

2 Theoretical Background

2.1 Principal component analysis

Any particular data matrix $X \in \mathbb{C}^{m\times n}$ has an associated covariance matrix [1]

$$\mathcal{C}_X = \mathrm{cov}(X) \equiv \frac{1}{n-1}(\delta X)(\delta X^\dagger) \tag{1}$$

where $\delta X = X - \langle X\rangle$. Each component $\mathcal{C}_{ij}$ expresses the covariance $\langle\langle X_{ik}X^*_{kj}\rangle\rangle \equiv \langle(X_{ik}-\langle X\rangle)(X_{kj}-\langle X\rangle)^*\rangle$, or correlation coefficient of the component X_{ij} in relation to the entire dataset[1]. Note that in the case of complex data one may also split the component vectors of X into real and imaginary parts and calculate a real correlation matrix between those components with doubled dimension. For example, a complex function $\psi(t)$ with autocorrelation function $\kappa(\tau) = \langle\langle\psi(0)^*\psi(\tau)\rangle\rangle$ has relation to its 2×2 autocorrelation matrix K,

$$\kappa(\tau) = K_{00} + K_{11} + i(K_{01} - K_{10}). \tag{2}$$

Note that $\kappa(\tau)$ has reduced information from $K(\tau) = \mathrm{cov}(\{\mathrm{Re}(\psi), \mathrm{Im}(\psi)\})$ [2]. Now X represents a linear operation between vector spaces $\mathbb{V}^m \to \mathbb{V}^n$. However, $\mathrm{cov}(X) \in \mathbb{C}^{m\times m}$ is by definition self-adjoint. Therefore its eigenvalues are distinct and there exists a choice of basis $q \in \mathbb{V}$ diagonalizing $\mathrm{cov}(X)$,

$$(\delta X)(\delta X^\dagger) = Q\Lambda Q^{-1} \tag{3}$$

where Λ is the diagonal of eigenvalues and Q is unitary. In the projected variable $Y = Q^T X$, $\mathrm{cov}(Y) = \frac{1}{n-1}\Lambda$, so that these *principal components* Y represent the data in a decorrelated basis.

[1] Geometrically, the inner product of components represents a correlation by measuring the extent of their alignment.

2.1.1 Singular value decomposition

A limitation of the eigenvalue approach is that the matrix X itself may not be square and so not diagonalizable in that particular sense. The generalized eigendecomposition/diagonalization of X is called the *singular value decomposition* (SVD). The idea is to measure the action of X on vectors V spanning a unit hypersphere in the base space into unit vectors U of resultant lengths Σ tracing a hyperellipse in the target space,

$$XV = U\Sigma, \quad V^{-1} = V^\dagger, \quad \implies \quad X = U\Sigma V^\dagger \tag{4}$$

In this picture $U \in \mathbb{C}^{m\times n}$, but in most implementations $m-n$ additional orthonormal columns are added to make it square $m \times m$ (i.e. through Gram-Schmidt procedure), and thus Σ is a diagonal $m \times n$ matrix with silent $m-n$ rows (possibly more if rank r of X is less than n). As matrices of orthonormal basis vectors both U and V are unitary. The stretching components Σ are called *singular values*, apparently here in the sense of "particular" or "characteristic" rather than "explody singular".

As a generalized eigendecomposition the direct SVD of X also accomplishes diagonalization of the covariance matrix $\mathrm{cov}(X)$ by representing the *two* characteristic problems of X in the case $m \neq n$, [1]

$$X^\dagger X V = V\Sigma^2, \tag{5}$$

$$XX^\dagger U = U\Sigma^2. \tag{6}$$

The latter guarantees diagonal covariance of the projected variable $Y = U^\dagger X$, as $\mathrm{cov}(Y) = \frac{1}{n-1}\Sigma^2$.

3 Algorithm Implementation and Development

The provided data consists of color video frames. The RGB color was first mapped to grayscale in order to work with scalar data. For each of the four data sets the video frames were then synchronized in time to equal frame-lengths by choosing start and end frames by eye. As a first reduction of the data to its relevant parts, coordinates of the bucket (oscillating mass) then had to be tracked. The first attempt at bucket tracking consisted of calculating temporal differences (effectively a differentiation). This differencing procedure was noisy and unreliable for the shaky camera, so a method based on "color filtering" was utilized.

The color filtering method starts with the observation that the brightest part of the image is mostly the white paint-bucket, so the grayscale camera dataset can be set to zero below a certain brightness threshold using masking notation, `cam1[cam1 <= t] = 0` where `cam1` is the dataset and `t` the threshold value (for example, with a maximum grayscale value of 255, `t`~ 240 was appropriate). As other bright spots happened to be on the periphery of the camera field-of-view, these values were then masked to zero manually (see Fig. 1 for example). The (x, y) trajectory of the bucket was then found using the mean coordinates of all points above the threshold per frame, e.g. `np.mean(X[np.where(cam[:,:,i] > 0)])` where `X` is a meshgrid of x-pixel coordinates. Finally, as there are fluctuations per frame in the particular parts of the bucket passing the color filter, the coordinates are smoothed with a moving average filter (typically three-five frames).

After the coordinates were determined, the data matrix was assembled as $X = \{x_1; y_1; \cdots\}$, its SVD calculated (of $\delta X = X - \langle X\rangle$), and the projected components $Y = U^T X$ plotted (just transpose as the data is real-valued). For datasets with multiple oscillation components FFTs were done to compare frequencies.

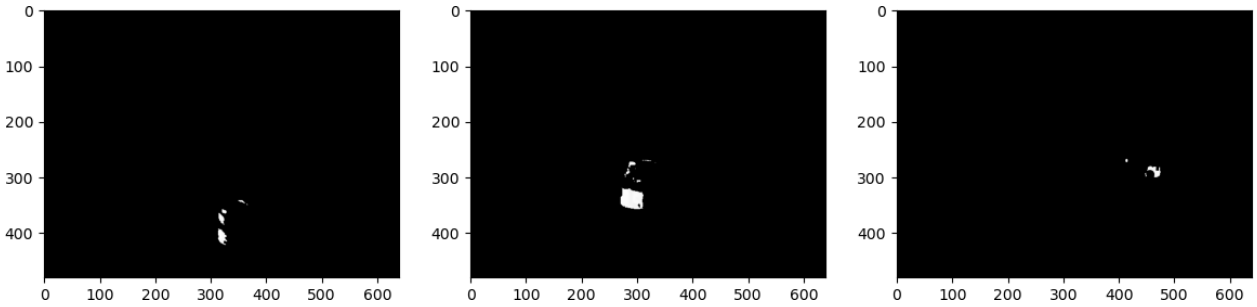

Figure 1: Appearance of the bucket in first dataset (no shake) after intensity masking.

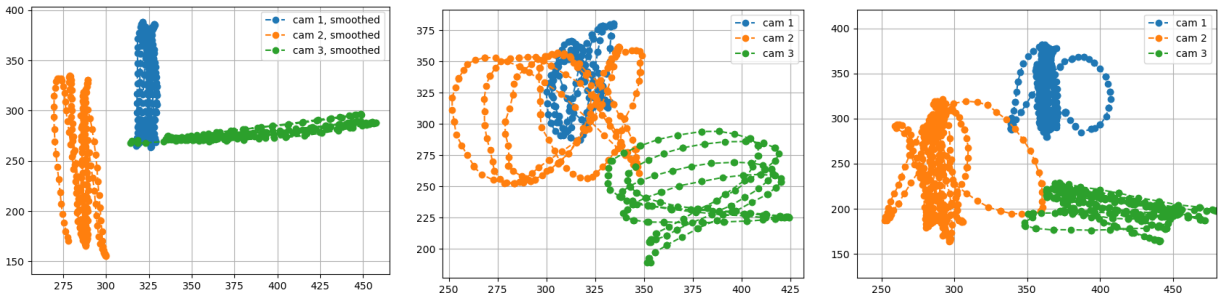

Figure 2: Bucket coordinates of three perspectives superimposed, with x and y axes pixels (arb. units). Left: Base case, SHM with no shake. Middle: SHM plus pendulum motion. Right: SHM plus bucket spin.

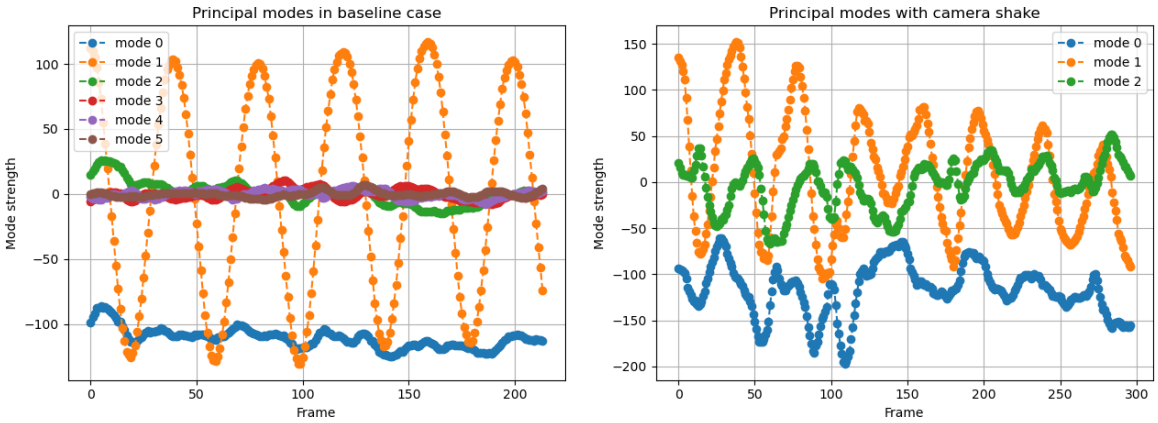

Figure 3: Projected modes $Y = U^T X$ for the bucket coordinates without (left) and with (right) camera shake. The six modes account for the three camera perspectives of two coordinates per frame. Evidently the oscillatory signature appears in the first dynamic mode, as the zeroth mode (a sort of DC component) describes the general location of the mass in the principal frame. It's clear that the addition of noise (right) splits the oscillation signature across different modes as they can no longer be "perfectly" decorrelated.

4 Computational Results

Figure 2 shows the results of tracking the bucket coordinates for the cases of simple harmonic motion (SHM) with no camera shake, with pendulum motion, and then with rotational motion. The case of camera shake is not shown for brevity, but is of course much noisier than any of these three cases. Then, Fig. 3 shows the PCA results for the baseline case (no camera shake) and with shake. According to the results, oscillation appears at first-order in the decorrelation and the addition of noise smears the oscillatory signature across multiple modes. This makes intuitive sense as noise ought to prevent ideal decorrelation of data components.

Finally Fig. 4 shows the results of PCA on the SHM first with superimposed pendulum oscillation and secondly with bucket rotation. The interesting and seemingly magical result (with mathematical foundations of course, cf. Section 2!) is that the separate oscillatory components are well separated in the decorrelation. Some interference (weak modal coupling) is to be expected to hamper decorrelation, and this is perhaps seen in the zeroth mode (however this is merely speculation). FFTs on these datasets are not shown for conciseness, as the largest amplitude frequencies are clearly visible. However, one is shown in Fig. 5 to demonstrate that the "mode 0" waveform of the pendulum dataset is indeed composed of both frequencies.

5 Summary and Conclusions

This numerical experiment analyzed video clips of some household oscillations (tensile string oscillation and gravity pendulum) as illustration of how awesome the principal component analysis method is and the

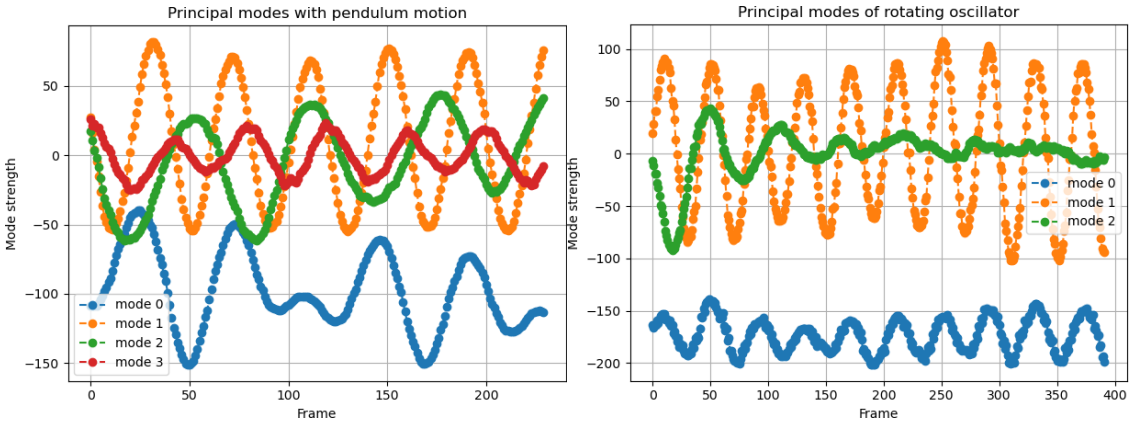

Figure 4: Projected modes of bucket coordinates for cases of SHM superimposed on pendulum motion (left) and bucket rotation (right). In comparison to the baseline case (Fig. 3) clearly "mode 1" still describes the bucket SHM as their frequencies match. The pendulum motion appears as a lower-frequency oscillation apparently in "mode 2". However, the occurence of both oscillations appears to leave a modulated trace (superposition?) in mode 0, in curious correlation with the Wigner function experiments of the last homework. The rotating oscillator case shows a damped rotation tendency in "mode 2," matching the video here.

usefulness of singular value decomposition. The author learned a great deal about this subject in the exercise, having never done an SVD before, and is very grateful for the learning opportunity. Theory was combined with image processing techniques to explore an analytical tool with broad scientific applications. As usual, Python was used for all implementations in this exercise and may be found at the author's Github page here

References

[1] J.N. Kutz. *Data-driven Modeling and Scientific Computation.* Oxford University Press, 1st edition, 2013.

[2] N.G. van Kampen. *Stochastic processes in physics and chemistry.* Elsevier, 3rd edition, 2007.

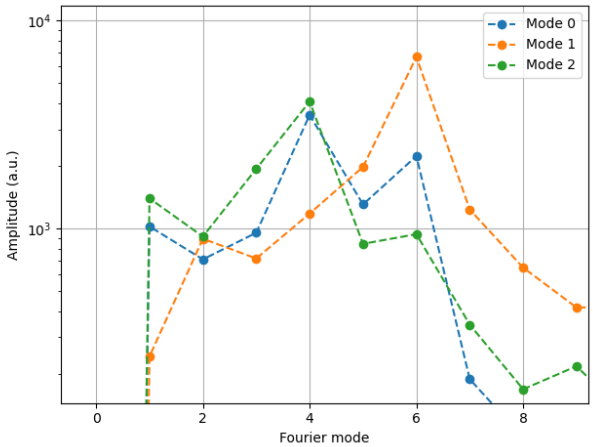

Figure 5: First few Fourier components from FFT of the pendulum case, showing the frequencies in the "mode 0" waveform to be the SHM and pendulum frequencies 4 and 6, and perhaps interference in mode 5.

Appendix A Python Functions

The following list compiles important Python functions used in implementation:

- The masking capability `array[bool condition] = x` is used to isolate bright parts of the image,
- The nice concise expression `np.mean(Y[np.where(m1[:,:,i]>0)])` is used to find the mean Y values of parts of the camera array `m1` which passed the color filter, where `X,Y=np.meshgrid(x,y)` and the coordinates are taken to be `x = np.linspace(0, m1.shape[0])`,
- `u, s, vh = np.linalg.svd(X, full_matrices=True)` is used to compute the full SVD of the data matrix after setting `X -= np.mean(X)` for covariance analysis.

Appendix B Python Implementation

A separate analysis script was used for each dataset and the files are quite similar. Only pendulum is shown for brevity (see Github page for other files).

```
import numpy as np
import matplotlib.pyplot as plt
import time
# to read .mat files
from scipy.io import loadmat
from scipy.interpolate import RegularGridInterpolator
from scipy import ndimage

### RGB to grayscale
# def rgb2gray(rgb):
#     r, g, b = rgb[:,:,0,:], rgb[:,:,1,:], rgb[:,:,2,:]
#     gray = 0.2989*r + 0.5870*g + 0.1140*b

#     return gray

# ### Read camera .mat files
# data11 = loadmat('data/cam1_3.mat')
# data12 = loadmat('data/cam2_3.mat')
# data13 = loadmat('data/cam3_3.mat')

# ### Turn into numpy arrays using list comprehension
# dmtrx1 = np.array([[val for val in elements]
#                    for elements in data11['vidFrames1_3']])
# dmtrx2 = np.array([[val for val in elements]
#                    for elements in data12['vidFrames2_3']])
# dmtrx3 = np.array([[val for val in elements]
#                    for elements in data13['vidFrames3_3']])

# ### Convert to grayscale
# dmtrx1g = rgb2gray(dmtrx1)
# dmtrx2g = rgb2gray(dmtrx2)
# dmtrx3g = rgb2gray(dmtrx3)

# ### Save
# with open('data/pendulum1.npy', 'wb') as f:
#     np.save(f, dmtrx1g)
```

```
# with open('data/pendulum2.npy', 'wb') as f:
#     np.save(f, dmtrx2g)

# with open('data/pendulum3.npy', 'wb') as f:
#     np.save(f, dmtrx3g)

### Load
ta = time.time()
with open('data/pendulum1.npy', 'rb') as f:
    dm1 = np.load(f)

with open('data/pendulum2.npy', 'rb') as f:
    dm2 = np.load(f)

with open('data/pendulum3.npy', 'rb') as f:
    dm3 = np.load(f)

print('numpy time is ' + str(time.time() - ta))

# print(dm1.shape)
# print(dm2.shape)
# print(dm3.shape)

### Get them synchronized (so annoying!!)
# third clip
s3 = 0
e3 = dm3.shape[2]-5
m3 = dm3[:,:,s3:e3]
# first clip
s1 = 7
e1 = m3.shape[2]+s1
m1 = dm1[:,:,s1:e1]
# second clip
s2 = 31
e2 = m3.shape[2]+s2
m2 = dm2[:,:,s2:e2]
# Shapes
print(m1.shape)
print(m2.shape)
print(m3.shape)

# "color filter"
m1[m1 <= 243] = 0
m2[m2 <= 240] = 0
m3[m3 <= 243] = 0

# Clear other bright objects
m1[:220,:,:] = 0
m2[:190,:,:] = 0
m3[:,:100,:] = 0
m3[:160,:,:] = 0

# Coords
x = np.arange(m1.shape[0])
```

```
y = np.arange(m2.shape[1])
X,Y = np.meshgrid(x,y, indexing='ij')

# mean pos
c1 = np.zeros((2, m1.shape[2]))
c2 = np.zeros_like(c1)
c3 = np.zeros_like(c2)
#fig, ax = plt.subplots(1,3, figsize=(15,10))
for i in range(m3.shape[2]):
    # cam1
    #ax[0].clear()
    #ax[0].imshow(m1[:,:,i], cmap='gray')
    # cam2
    #ax[1].clear()
    #ax[1].imshow(m2[:,:,i], cmap='gray')
    # cam3
    #ax[2].clear()
    #ax[2].imshow(m3[:,:,i], cmap='gray')
    #plt.pause(0.05)

    # Get mean positions
    # cam1
    c1[0,i] = np.mean(Y[np.where(m1[:,:,i]>0)])
    c1[1,i] = np.mean(X[np.where(m1[:,:,i]>0)])
    # cam2
    c2[0,i] = np.mean(Y[np.where(m2[:,:,i]>0)])
    c2[1,i] = np.mean(X[np.where(m2[:,:,i]>0)])
    # cam3
    c3[0,i] = np.mean(Y[np.where(m3[:,:,i]>0)])
    c3[1,i] = np.mean(X[np.where(m3[:,:,i]>0)])

### Kernel convolution (smooth with neighbors)
k = np.ones(3)/3 # box
#k = np.exp(-np.arange(-3, 3)**2.0)
c1 = np.array([np.convolve(c1[0,:], k, 'valid'), np.convolve(c1[1,:], k, 'valid')])
c2 = np.array([np.convolve(c2[0,:], k, 'valid'), np.convolve(c2[1,:], k, 'valid')])
c3 = np.array([np.convolve(c3[0,:], k, 'valid'), np.convolve(c3[1,:], k, 'valid')])

# Check coordinates
plt.figure()
plt.plot(c1[0,:], c1[1,:], 'o--', label='cam 1')
#0plt.plot(c1s[0,:], c1s[1,:], 'o', label='cam 1, smoothed')
plt.plot(c2[0,:], c2[1,:], 'o--', label='cam 2')
#plt.plot(c2s[0,:], c2s[1,:], 'o', label='cam 2, smoothed')
plt.plot(c3[0,:], c3[1,:], 'o--', label='cam 3')
#plt.plot(c3s[0,:], c3s[1,:], 'o', label='cam 3, smoothed')
plt.legend(loc='best')
plt.grid(True)
#plt.xlabel('x pixels')
plt.axis([0,480,0,640])
plt.show()

### Construct data matrix
X = np.array([[c1[0,:]],
```

```
                [c1[1,:]],
                [c2[0,:]],
                [c2[1,:]],
                [c3[0,:]],
                [c3[1,:]]]).reshape(6, c1.shape[1])

### SVD it
X -= np.mean(X)
u, s, vh = np.linalg.svd(X, full_matrices=True)
# Principal component projection
Y = np.matmul(u.T, X)
# Check it out...
plt.figure()
plt.plot(Y[0,:], 'o--', label='mode 0')
plt.plot(Y[1,:], 'o--', label='mode 1')
plt.plot(Y[2,:], 'o--', label='mode 2')
plt.plot(Y[3,:], 'o--', label='mode 3')
#plt.plot(Y[4,:], 'o--', label='mode 4')
#plt.plot(Y[5,:], 'o--', label='mode 5')
plt.grid(True)
plt.legend(loc='best')
plt.xlabel('Frame')
plt.ylabel('Mode strength')
plt.title('Principal modes with pendulum motion')

### FFT the PC proj.
M0f = np.fft.fft(Y[0,:] - np.mean(Y[0,:]))
M1f = np.fft.fft(Y[1,:] - np.mean(Y[1,:]))
M2f = np.fft.fft(Y[2,:] - np.mean(Y[2,:]))

plt.figure()
plt.semilogy(np.absolute(M0f), 'o--', label='Mode 0')
plt.semilogy(np.absolute(M1f), 'o--', label='Mode 1')
plt.semilogy(np.absolute(M2f), 'o--', label='Mode 2')
plt.grid(True)
plt.legend(loc='best')
plt.xlabel('Fourier mode')
plt.ylabel('Amplitude (a.u.)')
plt.tight_layout()

plt.show()
```