AMATH 482 HW1
Principle Component Analysis

Oliver Speltz

February 21, 2019

I Introduction

In this report, I will discuss and make use of an important topic in linear algebra, the Singular Value Decomposition (SVD). The SVD is helpful in pointing out redundancies in data and finding lower dimensional representations of a dataset. In this report I take data of a simple system, a mass bouncing on a spring, but under the assumption that I do not know what dynamics govern the system. I have multiple probes, cameras, giving me data on this system and I want to extract the dynamics in a lower dimensional space.

II Theoretical Background

The Singular Value Decomposition powerful tool in analyzing data. Mathematically, it is a decomposition of *any* matrix A into three matrices, such that,

$$A = U\Sigma V^*. \tag{1}$$

What is so special about this decomposition is that it breaks down how a square matrix transforms a vector into three steps: a rotation, a stretching, and another rotation. If A is real, both U and V are real, and they are unitary matrices, that is,

$$U^*U = UU^* = I \qquad V^*V = VV^* = I. \tag{2}$$

In other words, the complex conjugate transpose (denoted by A^*, sometimes A^H) of U and V are also their inverses. Further, U and V are orthonormal, that is, their columns have length one, and are all orthogonal to each other. Σ is a diagonal matrix, where the diagonal entries are real and positive. In fact, the number of non-zero σ_j, called singular values, is equal to the rank(A) or the span of A's columns. In other words, if there are redundant, rows, of the matrix A, there will be zeros on the diagonal of Σ.

Unitary matrices are easily understood because they just rotate vectors. Unitary matrices are really just another basis to represent a vector in. Diagonal matrices are intuitive because they just stretch vectors. Putting this all together, this means for a square A acting on any vector $\mathbf{x}$, it can be broken down as,

$$\begin{aligned} A\mathbf{x} &= (U\Sigma V^*)\mathbf{x} \\ &= U\Sigma(V^*\mathbf{x}) && \text{a rotation by } V^*, \\ &= U(\Sigma\hat{\mathbf{x}}) && \text{a stretching by } \Sigma, \\ &= U\tilde{\mathbf{x}} && \text{and another rotation by } U. \end{aligned}$$

Applied to data analysis, this tool is very powerful because it can point out redundant data, and it can give us new coordinate systems to look at things in. If you have multiple sensors of the same phenomenon, you

can arrange them in a wide data matrix X like

$$X = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \vdots \\ \mathbf{x}_m \end{bmatrix} \tag{3}$$

Where all m rows, $\mathbf{x}_i$, are data streams from a different sensor, and all n columns represent measurements taken at the each point in time (or space, etc). If X is of low rank, that means the rows have redundant data, there is a lot of covariance between them, which is exactly what SVD will tell us.

For a data matrix of this shape, one can consider the covariance matrix, which after subtracting out the mean of each row is,

$$C_X = \frac{1}{n-1} X X^T \tag{4}$$

Where each element c_{ij} is the covariance between $\mathbf{x}_i$ and $\mathbf{x}_j$. If all of our data streams are statistically independent, then the off diagonals of this matrix would be small, close to zero, while the diagonals would be larger (diagonals being each data streams variance). This would indicate that each sensor is relaying different information and that information is something worth noting.

Now if we take the SVD of X,

$$X = U \Sigma V^* \tag{5}$$

we can take a projection of our data matrix into the U coordinate system.

$$Y = U^* X \tag{6}$$

Now, if we examine the covariance matrix of this Y matrix, the representation of our data in the U basis. Called the *principle component basis*.

$$\begin{aligned} C_Y &= \frac{1}{n-1} Y Y^T \\ &= \frac{1}{n-1} (U^* X)(U^* X)^T \\ &= \frac{1}{n-1} (U^* X)(X^T U) && \text{if } U \text{ is real, } (U^*)^T = U \\ &= \frac{1}{n-1} (\Sigma V^*)(V \Sigma) && \text{from (5)} \\ &= \frac{1}{n-1} \Sigma^2 \end{aligned}$$

So the covariance matrix of our data in this U basis is diagonal. That means the covariance between each of the rows is 0, i.e. each row is statistically independent! Further, the variance of each row of Y is the associated singular value squared, so if any of them are 0, indicating X is of low rank, then that row of Y has no variance and can be essentially ignored.

A key theorem associated with the SVD is that if a matrix A has rank r then

$$A = \sum_{j=1}^{r} \sigma_j \mathbf{u}_j \mathbf{v}_j^* \tag{7}$$

That is, A is the sum of rank 1 matrices (a row vector times a column vector), each scaled by its singular value. If you have noisy data, the noise may cause a data matrix to have a higher dimensionality than the phenomenon actually does. If this is the case, you can use low rank approximations of your data where you sum only the first k principle components for a less messy look at your data.

$$A^k = \sum_{j=1}^{k} \sigma_j \mathbf{u}_j \mathbf{v}_j^* \tag{8}$$

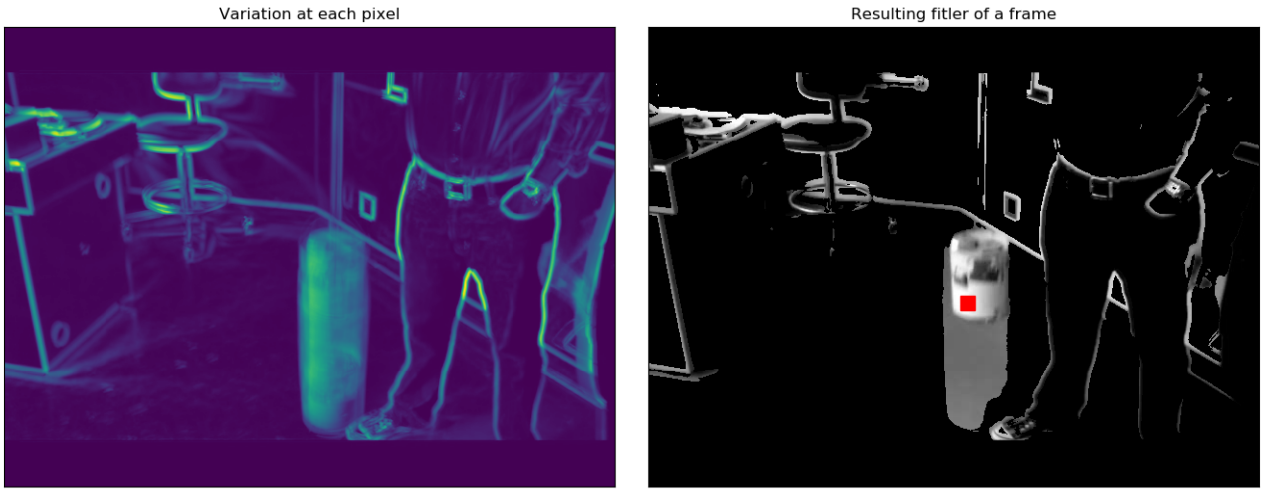

Figure 1: On the left, the variation by frame in each pixel, bright colors corresponding to high variation. On the right, the resulting filter of the image by setting low variance pixels to zero. The red square is the size of the box of pixels searched for with the highest values.

This is useful in getting low dimensional representations of your data. For example if $k = 1$, all rows of A^k would simply be scaled versions of each other, so you can look at one to see a 1 dimensional representation of whats happening. In the example of the data matrix X where the columns are different data points in the time, $\mathbf{u}_j$ would be column vectors which form "the best" basis to view the system in and $\mathbf{v}_j^*$ are row vectors which represent the temporal dynamics along each of the components of that basis. So looking at the dynamics of the significant components, i.e., the ones with large singular values, can give a low dimensional representation of your data.

III Algorithm Implementation and Development

For this report, we were given videos of a mass on a spring being held up by a person while it bounced around. The format of the videos was a MATLAB matrix, `[640,480,3,n_frames]` for the horizontal direction, the vertical direction, the RGB bands, and the number of frames in the recording. There were four different tests done, in the first test, the camera is stable and the mass is moving only in the z direction. In the second test, the mass is still only moving in the z direction, except now there is noise due to camera shake introduced. The third test introduces horizontal motion of the mass, but with a stable camera. The final test also has a stable camera, but now introduces rotation, as well as horizontal and vertical displacement, to the dynamics of the mass. Each different test has a view of the mass from three different angles, for a total of 12 videos.

We are interested in the dynamics of the spring, and what PCA can tell us about these dynamics, despite difference reference frames of the videos. In order to do so, the pixel coordinates of the mass on the spring must be extracted from each frame. To do so, first each frame should be converted to a gray scale, for reduction of dimensions. This can be done as a linear combination of the RGB bands with the weights $0.3r + 0.6g + 0.1b$. Now a method must be found to pick out where the mass is. One thing we can take advantage of in this instance is that the mass is painted white. This means it will constitute a large clump of pixels that are higher in value than most. Another fact we can take advantage of in the stable cases is that the variation at any particular background pixel is going to be rather low, because nothing is moving through that pixel.

Using these two pieces of information, we can first take the standard deviation at each pixel coordinate. Those pixels which do not have enough variation will be set to zero for all frames. For a threshold, I found 20

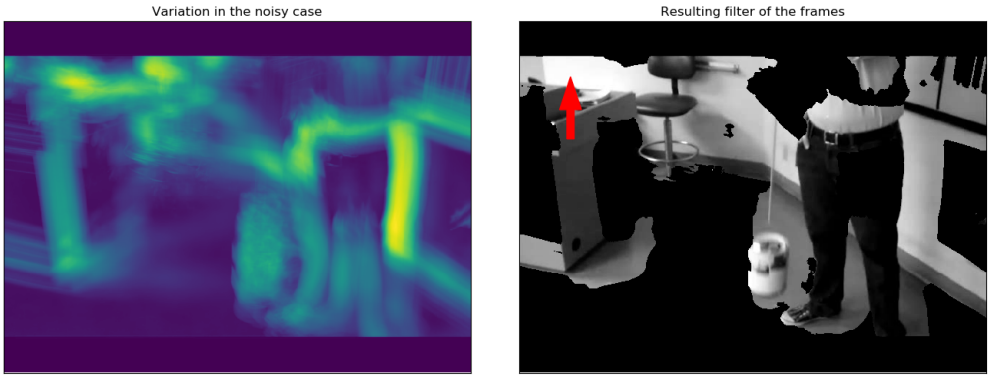

Figure 2: On the left, the variation of each pixel coordinate in one of the unstable cases. On the right, the resulting filter of the video. The bright area indicated by the red arrow was often erroneously determined to be the location of the mass.

worked fairly well across the different tests at blocking out the background without compromising the path of the mass. See Figure 1 for a heatmap of the variation at each pixel coordinate in one of the views. Then in each frame, we can search for the box of pixels which gives us the highest sum of pixel values, corresponding to the bright patch that is the mass. This is done by adding together values next to each other the size of the `box` parameter in the `extractPos()` function of the attached code. I used a box size of 15 because after examination of the videos, the mass is about a 15 by 15 block of pixels.

This strategy worked fairly well for the non-noisy cases. Producing data like in Figure 3 However, in the noisy case, the camera shake kills this technique. Shaking introduced very high variability in areas where there is highly contrasted edges, for example, the portion in Figure 2 where a leg in black pants is standing up against a white wall. As a result, the piece we are interested in is actually of lower variability than the background and we cannot filter out a significant portion of it. This led to erroneous extraction of the location, in particular, the bright wall in the left of the frame was being selected often.

To combat this, we must use a more supervised technique. By observing the videos, we can pick out a more confined frame that the mass is bouncing in and crop out as much background information as possible. With this we are able to extract fairly accurate position data from the noisy cases. The cropping also had a nice effect of drastically speeding up the search for the brightest cluster of pixels. Although cropping changed the pixel coordinates of the mass by a translation of however much was cropped off, this wouldn't matter because later we will subtract the mean of each data stream.

Now with our position data in hand, we can go about constructing a data matrix X_i for each test $i = 1, 2, 3, 4$ with the horizontal position and vertical position from each view point aligned in rows and stacked on top of each other. Note that the views of any one test are not guaranteed to be the same length of frames. So assuming the videos are started at the same time, if we take the minimum n frames of the three views, we can use that as the number of columns of our data matrix. Finally, we must subtract the mean from each row, so that the statistical principles of the SVD hold. Using the SVD method that is built into `numpy.linalg`, we can examine the rows of V to see a lower dimensional representation of our data along the principle components.

IV Computational Results

IV.1 Position Extraction

In Figure 3 are some examples of the position data this algorithm extracted from the videos. The oscillations of the dynamics show up pretty clearly and smoothly, however, it is far from perfect. For one, the location

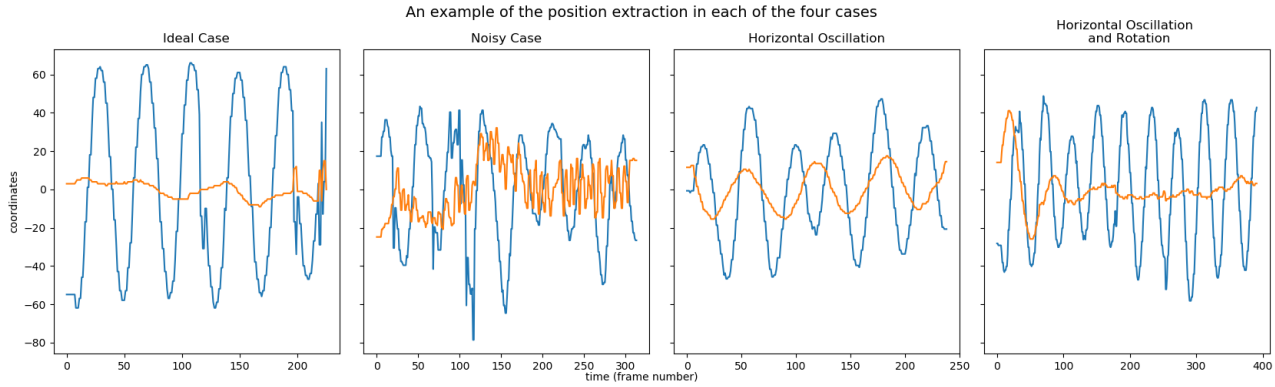

Figure 3: The position of the mass on the spring from one of the cameras in each test. The blue represents the first axis of the pixel coordinates, the orange is the second axis.

it returns is a pixel coordinate, so it is discrete. This leads to blocky looking waves. Another limitation of this method is that different spots on the mass can be selected as the location of the mass. The size of the box of brightest pixels searched for was 15^2, which is smaller than the actual size of the mass in the videos. See Figure 1 for a comparison of the box size to the bouncing mass. As a result, the top or bottom of the mass could be determined to be the center and throw off the results. Sometimes, completely different points in the frame than where the mass was were selected, especially when the background had a lot of white in it.

The extraction process took very long periods of time, sometimes around 10 minutes, mostly from the adding together each matrix to get the sum of the box pixel values. Although using a bigger box may have yielded a more consistent center of the mass, the computation time increased $\mathcal{O}(N^2)$ with box size N.

IV.2 Principle Component Analysis

Shown in Figure 4 are the singular values obtained from the data in each test from this procedure. In the ideal case, the first two singular values are dramatically larger than the rest. This is an indication that many of our data streams are redundant, which is what we should expect, as there is only one dimensional movement which can be captured with one camera. However, we know that the dynamics of the system can be encoded in only dimension. Upon closer inspection of the data, the third camera seems to have started recording significantly later, causing the peaks of the motion it captured to not be in phase with the others.

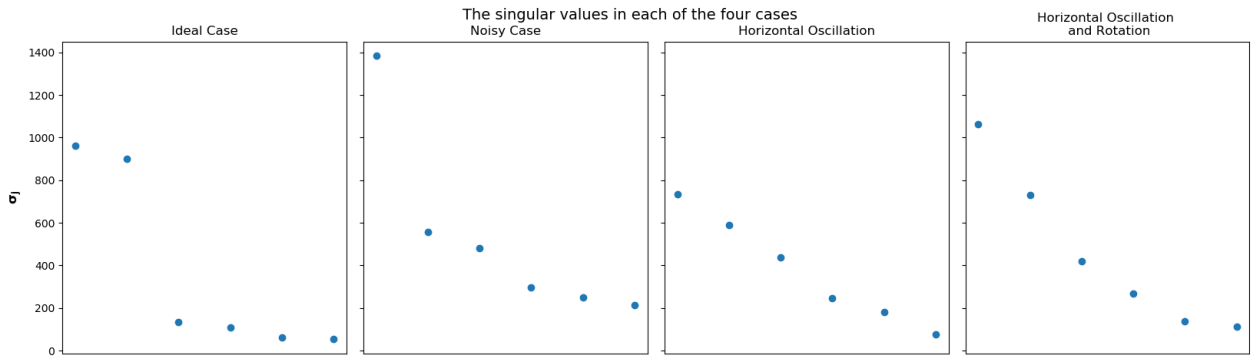

Figure 4: The singular values from the data matrices of each different test. In test 1 it is very clear that the dynamics can be represented in two dimensions.

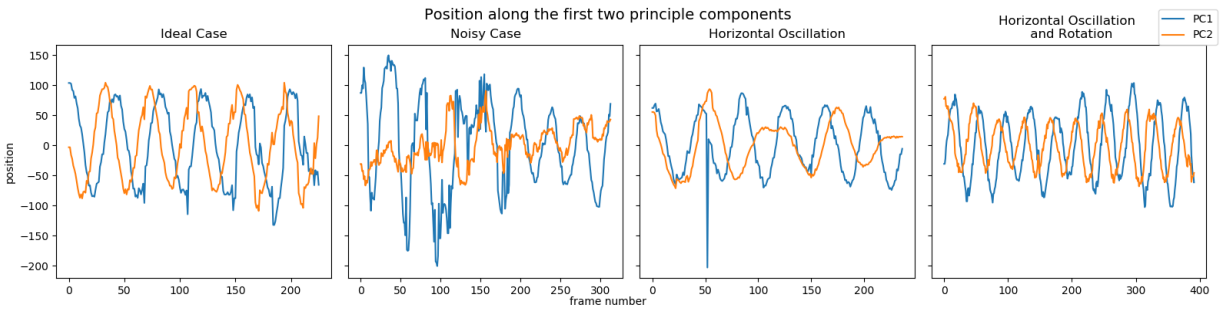

Figure 5: The projection of the data onto the first two principle components. Despite the noise, the projections reveal some of the underlying dynamics. Blue is the motion along the first principle component, orange is the motion along the second.

This leads to low covariance between it and the other cameras, leading to two significant singular values. In the other cases, it is not so clear and the singular values decrease in size more steadily. I think that this can also be attributed to noise from the position selection, as well as the cameras not having started recording at the same time.

Shown above in Figure 5 are the first two rows $\mathbf{v}_j^*$ of the SVD of the different data matrices. These represent the temporal activity along the first two of the principle components of the data. Although the singular values do not decay to zero quickly, we can still see what we know is going on in the principle components. In the first case, we can see that the motion along the second principle component is almost exactly the same as the first, just out of phase. This is because the cameras are out of sync. In the noisy case, the oscillations are clear in the first principle component direction, the second component's motion seems to be largely correlated with the first, except for the peaks of noise between frames 75 and 175. In the last two cases, we see that there is oscillation along the second component out of phase and at a different frequency than the first. This reflects what we know about the real system because in the last two cases the mass is oscillating in both the horizontal and vertical directions, not necessarily at the same frequency.

V Summary and Conclusions

In this report, I explored the use of principle component analysis by attempting to reduce the dimensionality of some "unknown" system. Having received multiple data streams of the same phenomenon from different views, I had to find what is the best view where the dynamics are captured by as few dimensions, or data streams, as possible. This can be applied to more than just a mass bouncing on a spring but to genomic data, image data, weather data, and more. In this specific application, a way it could be improved upon would be to find a way to line up the oscillations captured from the different cameras so that they are synced. One way this could be done would be to line up the peaks of the oscillations, however, one would have to be careful in the cases where the mass is oscillating in the horizontal and vertical directions, because the peaks do not necessarily coincide in time, especially if the camera is angled differently in different views. One could Fourier transform the data and line up the the different rows that have the most similar frequency components, because that would be a good indicator that they are capturing the same direction of motion. However, this is using a lot of our knowledge of what the results should be, and finding a way that makes no assumptions about the true nature of the data would be challenging.

Appendix A: Python Functions used

`numpy.linalg.svd(a,full_matrices)`
The SVD algorithm built into `numpy`'s linear algebra submodule. Returns the U, Σ and V^* of the SVD of `a` as `numpy` arrays, the Σ returned is not actually a diagonal matrix, but rather a 1D array of the diagonal elements. `full_matrices` is a boolean that if true the function returns the full SVD, if false, it returns the reduced SVD.

`numpy.argmax(a)`
Returns the index of the maximum of `a`, the index is into the flattened array.

`numpy.unravel_index(inds,dims)`
Takes an index into a flattened array, or array of indices, and returns the corresponding index into the un-flattened array with shape in the tuple `dims`.

Appendix B: Main Python Code

```
import numpy as np
import scipy.io as spio
import matplotlib.pyplot as plt
import time
import winsound as ws

def loadCams(test):
    ''' test is a string for the test number (1-4), loads the frames for all
    the different perspectives for this test in a list, in grayscale'''

    camNames = [str(k) + '_' + test for k in range(1,4)]

    gray = [0.3, 0.6, 0.1]
    t = time.time()
    cams = [spio.loadmat('cams/cam' + c)['vidFrames' + c] for c in camNames]
    print('load mat time',time.time()-t)

    views = []
    for i,cam in enumerate(cams):
        xx,yy,rgb,fs = cam.shape
        t = time.time()
        out = np.zeros((xx,yy,fs))

        for f in range(fs):
            out[:,:,f] = np.dot(cam[:,:,:,f],gray)
        print('grayscale convert',i,time.time()-t)
        t = time.time()
        views.append(out)
    return views

def extractPos(view,thresh,box,crop=None):
    '''takes a single viewpoint of the bouncing mass and extracts the position
    of it as a numpy array (2,n_frames) with pos[0,:] = dim1, pos[0,:] = dim2,
    thresh is the threshold of variance for a pixel value to be kept, box is
    box size to search for with the brightest pixels, crop is a length 4 list
    [dim0 start, dim0 stop, dim1 start dim1 stop]'''
    if crop:
```

```
        view = view[crop[0]:crop[1],crop[2]:crop[3],:]
    x,y,fs = view.shape
    pos = -1 * np.ones((2,fs))
    t = time.time()
    pixel_var = np.std(view,axis=2)
    boo = (pixel_var > thresh).reshape((x,y,1))
    boo = np.dot(boo,np.ones((1,fs)))
    view = view * boo
    print('filter time:',time.time()-t)
    t = time.time()
    for f in range(fs):
        frame = view[:,:,f]
        s = np.zeros_like(frame)
        for i in range(box):
            for j in range(box):
                s[0:x-box+1,0:y-box+1] += \
                frame[i:x+i-box+1,j:y+j-box+1]

        pos[:,f] = np.unravel_index(np.argmax(s),(x,y))
    print('add time:',time.time()-t)
    return pos, view, pixel_var

# crop coordinates for the noisy case
crops = [[0,480,300,450],
         [0,480,200,400],
         [210,350,0,640]]

for test in ['1','2','3','4']:
    views = loadCams(test)
    for i in range(1):
        view = views[i]
        if test == '2':
            crop = crops[i]
        else:
            crop = None
        pos,filtered,pv = extractPos(view,20,15)
        ws.Beep(880,500)
        np.save('pos'+test+str(i),pos)

import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

def loadNpy(test):
    poss = [np.load('pos' + test + str(i) + '.npy')  for i in range(3)]
    n = min([pos.shape[1] for pos in poss])
    X = np.vstack([pos[:,0:n] for pos in poss])
    X = X - np.mean(X,axis=1).reshape((6,1))
    return X

# ideal case, vertical motion only and stable camera
X1 = loadNpy('1')
```

```
u1,s1,vs1 = np.linalg.svd(X1,full_matrices=False)
pca = PCA()
pca.fit(X1)
print(pca.singular_values_)
# vertical motion with a shakey camera
X2 = loadNpy('2')
u2,s2,vs2 = np.linalg.svd(X2,full_matrices=False)
# stable camera, now with vertical and horizontal motion
X3 = loadNpy('3')
u3,s3,vs3 = np.linalg.svd(X3,full_matrices=False)
# stable camera, with vertical horizontal, and rotation
X4 = loadNpy('4')
u4,s4,vs4 = np.linalg.svd(X4,full_matrices=False)

s = [s1,s2,s3,s4]
titles = ['Ideal Case','Noisy Case',
          'Horizontal Oscillation',
          'Horizontal Oscillation \n and Rotation']
fig,axes = plt.subplots(1,4,figsize=(16,4),sharex=True,sharey=True)
for i in range(4):
    axes[i].plot(s[i],'o')
    axes[i].set_title(titles[i])
    axes[i].set_xticks([])
axes[0].set_ylabel('$\mathbf{\sigma_j}$',fontsize=12)
fig.text(0.5,0.95,'The singular values in each of the four cases',
         ha='center',fontsize=14)

# project in principle component directions
Y = []
X = [X1,X2,X3,X4]
V = [vs1,vs2,vs3,vs4]
fig,axes = plt.subplots(1,4,figsize=(16,4),sharey=True)
for i in range(4):
    line1 = axes[i].plot(s[i][0]*V[i][0,:])
    line2 = axes[i].plot(s[i][1]*V[i][1,:])
    axes[i].set_title(titles[i])
axes[0].set_ylabel('position')
line1[0].set(label='PC1')
line2[0].set(label='PC2')
fig.text(0.5,0.95,'Position along the first two principle components',
         ha='center',fontsize=14)
fig.text(0.5,0.01,'frame number',ha='center')
fig.legend()
```