MATH 494-A: Assignment 2
Identifying Dynamics in Low and High Noise Conditions

Myriam Boucher-Pinard (40109509)

March 12, 2023

Abstract

This study has the goal of analyzing the nonlinear motion of an object on a spring through space using Principal Component Analysis (PCA) and Sparse Identification of Nonlinear Dynamics (SINDy) on six videos, three for different perspectives of a low-noise ideal case and three for different perspectives of a high-noise case. This is accomplished through the learning of equations of motion, created by the SINDy algorithm, to approximate Newton's Second Law of Motion using polynomials. Particularly, it was found that two principal components were required to do so up to some error, most likely due to tracking method error.

1 Introduction and Overview

According to Newton's Second Law of Motion, for an object of constant mass, acceleration depends on the velocity and position of the object [2]. In this study, we shall be analyzing the motion of a can on a spring and approximating its trajectory through the use of the Sparse Identification of Nonlinear Dynamics (SINDy) algorithm, as the motion of the can is oscillatory in space. As three angles of the motion shall be analyzed for two videos (one with low noise and one with high noise, in that the camera shakes quite a bit), the data will first have to be pruned for redundancy, accomplished through the Singular Value Decomposition (SVD) method. This shall identify the dimensions containing the most amount of information, efficiently determining where the majority of the algorithm will need to draw its information from [1]. Then, using the Principal Component Analysis (PCA) will allow for these high-dimension pieces of information to be translated to linear functions, identifying orthogonal directions; these preserve as much variability as possible by ordering the orthogonal functions by variance and are computed through the covariance matrix [3]. The study is completed through the use of the SINDy algorithm, allowing us to take these principal components and apply them to the model as an approximation of the motion of the object. Furthermore, the reason for maintaining two dimensions (principal components) shall be discussed.

2 Theoretical Background

2.1 Principal Component Analysis (PCA)

The mathematical understanding of the principal component analysis draws from its use in determining the dominant spatial directions of motion, or, the way in which the can moves around in the projected plane. It thus serves to determine the direction of the motion, and order the orthogonal direction of the motion by variance [1]. This is accomplished through variance first, and then covariance.

$$\sigma_x^2 = \frac{1}{n-1} xx^T \tag{1}$$

$$\sigma_y^2 = \frac{1}{n-1} yy^T \tag{2}$$

$$\sigma_{xy}^2 = \frac{1}{n-1} xy^T \tag{3}$$

Due to the size of the data set, considering the computation of variance, or the spread of the data, on a larger scale is important. Thus, computing the population variance, using an unbiased estimator, is key. Then, considering two sets of data, the variance may be computed for either using equations 1 and 2; in our case, the x- and y-directions. Computing the covariance conversely depends on the way in which the variables vary from one another, and is thus computed using equation 3 [1]. This covariance is essential in the PCA, as it measures the redundancy of the data set; thus, if the motion in either x- or y-directions is quite similar, then the covariance shall be quite similar to the variance of the individual data. As such, PCA shall prune the redundant information to ensure that the most important information is defined in our later model.

As abovementioned, the goal of the PCA is to nullify any redundancies, such that each principal component proposes a new piece of information. This is accomplished through the diagonalization of the covariance matrix, through Singular Value Decomposition (SVD).

$$X = U\Sigma V^* \tag{4}$$

Here, equation 4 defines matrix X, composed of unitary (complex orthogonal) matrices U and V, respectively representing space and time [1].

$$Y = U^T X \tag{5}$$

Returning to PCA, we are essentially looking for a change in basis. If we let X in equations 4 and 5 be the x- and y-directional data, then Y represents the same data, but in the principal component basis, and the transpose of U makes this basis independent of time.

$$\sigma_Y = \Sigma^2 \tag{6}$$

The final step of the PCA is thus to compute the covariance of Y, in order to determine the dominant spatial directions of motion. This is apparent, as equation 6 provides a diagonal matrix, the diagonal being composed of the principal components of the data set. These are equivalent to the square of the singular values in the sigma matrix, hence the equality in equation 6.

2.2 Sparse Identification of Nonlinear Dynamics (SINDy)

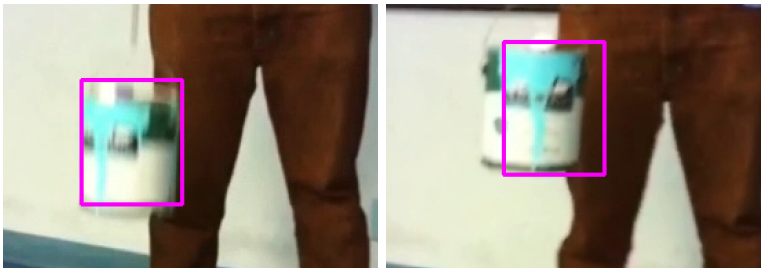
Now that the diagonal matrix of principal components has been determined, the Sparse Identification of Nonlinear Dynamics (SINDy) algorithm must be created in order to approximate the harmonic motion of the can on a spring. In this algorithm, we are effectively fitting a function to its Taylor expansion, such that the complex motion in space is approximately reproduced by polynomials. A dictionary is created, composed of as many elements (or more) necessary to describe the nonlinear motion; in our case, polynomial terms [1]. Considering we are dealing with motion in a continuous-time system, we must therefore learn an ordinary differential equation, due to the nature of the motion. As we wish to illustrate a physical law containing velocity and acceleration, solving at least one differential equation to describe the motion becomes necessary [2, 1].

$$x'(t) = F(x) \tag{7}$$

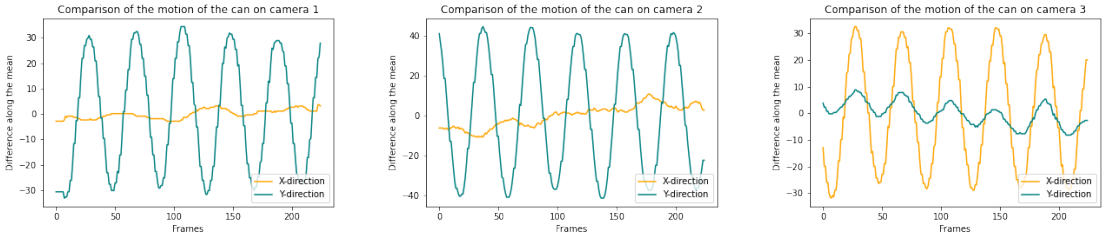
$$x(t) = x(0) + \int_0^t F(x(s))ds \tag{8}$$

Applying the Riemann sum approximation, through equations 7 and 8, the function becomes smooth.

A final element to note from the SINDy algorithm is that of the sparsity parameter; indeed, here, we are essentially applying a coefficient of zero to any element of the dictionary that adds little information to our model [1]. Thus, this limits the complexity of the model, by ensuring that the most important elements have the most effect.



Figures 1 and 2: Tracker centred on the can (left) versus tracker not centred on the can (right).



Figures 3, 4, and 5: Tracking the motion of the paint can in the low-noise condition.

3 Algorithm Implementation and Development

3.1 Importing the Data and Generating a Video

The first component of the study was of converting the data from a Matlab array to that of a python array, and extracting the frames in order to be able to convert these frames to a video. The integration of the scipy.io collection was essential here, in order to extend the abilities of Numpy, as well as then installing OpenCV in order to convert the arrays to videos. This procedure was conducted on all six videos, included below for the three videos of the low-noise condition.

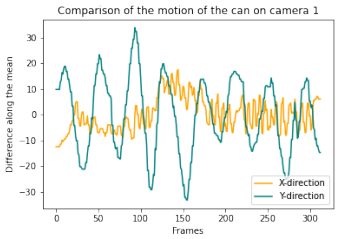
3.2 Tracking the Object

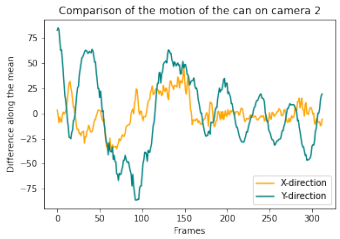
When declaring the library of trackers that were available to us, the MIL (Multiple Instance Learning) tracker was most interesting to select, as it is more resistant to noise, and as such, would allow for our code to be more generalized between the low- and high-noise conditions [4].

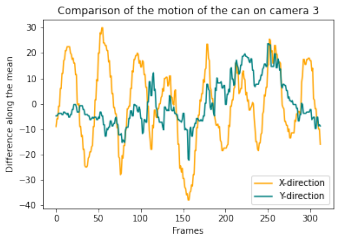
As discussed in Appendix A, the tracking was initiated through the calling forward of the video (more precisely, the very first frame), and a rectangle could be drawn around the object. Then, the program would track the bucket in every frame and transmit the coordinates of the rectangle, thus effectively creating a matrix of coordinates. However, viewing the tracking of the can demonstrated that the square could be quite off of its target when the environment around it changed drastically.

For instance, as observed in figures 1 and 2, when the pant of one of the professors came into view, the rectangle would focus on the coordinates of the pants rather than follow the paint can. This most definitely added noise to the data. Once all of the trackings were complete, all of the x- and y-coordinates were appended to a single matrix, and only the fewest number of coordinates were retained (the first video had the fewest number of frames).

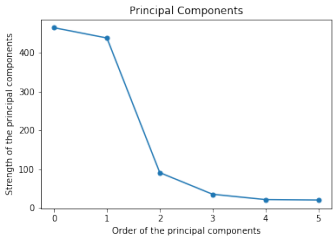
It can be observed in figures 3, 4, and 5 that the low-noise data has a clear distinction in the object motion along the x- and y-directions, as the paint can mostly travelled in the vertical direction. Thus, figures 3 and 4 have the more prominent oscillating y-direction motion for the paint can, and figure 5 has the oscillating x-direction motion as the camera was rotated, thus demonstrating horizontal motion. Nonetheless, as it is

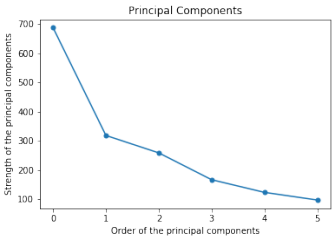






Figures 6, 7, and 8: Tracking the motion of the paint can in the high-noise condition.





Figures 9 and 10: Principal Components of the Low-Noise (9) and High-Noise (10).

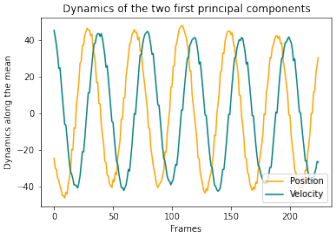
the difference in the motion and not the direction of the motion that determines the result of the PCA, this had no incidence.

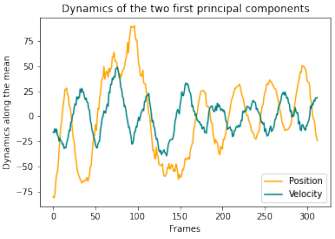
In figures 6, 7, and 8, the high-noise condition demonstrated a rather different pattern (although the more salient direction still retained a somewhat oscillatory trajectory). This is in larger part due to the error of the tracker in some of the frames. There were instances when the tracker would completely lose the object. Nonetheless, a similar pattern may be observed to that of the low-noise condition, in that the x-direction retained a more noticeable oscillating trajectory for cameras 1 and 2, and camera 3 had the more prominent y-direction trajectory.

3.3 Running PCA and SVD

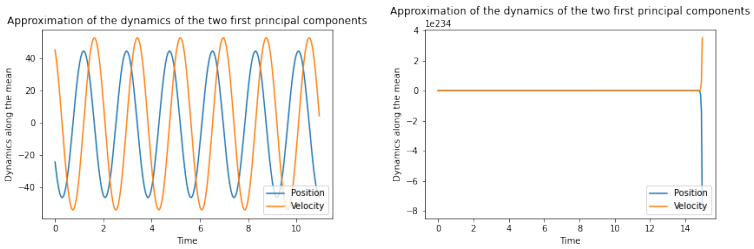
Prior to running the PCA, the mean was taken of every row of the x- and y-coordinates matrix, in order to subtract the mean from the coordinates. This would later allow for the PCA to be along the x-axis rather than in the range of much larger y-values. Then, using the preexisting SVD and covariance function from Numpy, equation 5 could be applied.

Figures 9 and 10 demonstrate the principal components of the low-noise condition (9) and the high-noise condition (10). We observed that the first two components in the low-noise condition were indeed much more salient, but in the high-noise condition, it could be problematic. In Figures 11 and 12, the dynamics for the low-noise condition (11) and the high-noise condition (12) were established, in the former the position and





Figures 11 and 12: PCA Dynamics of the Low-Noise (11) and High-Noise (12).



Figures 13 and 14: SINDy Models of the Dynamics of the Low-Noise (13) and High-Noise (14).

velocity trajectories being quite similar but shifted. In the latter, the trajectories were quite different, this again most likely due to the existing noise of the video and the added noise of the tracking.

3.4 Implementing SINDy

$$\frac{dx}{dt} = -2.454 - 3.004(z[1]) \tag{9}$$

$$\frac{dy}{dt} = 4.509 + 4.143(z[0]) \tag{10}$$

$$\frac{dz}{dt} = [\frac{dx}{dt}, \frac{dy}{dt}] \tag{11}$$

The SINDy algorithm was finally integrated, having only two elements and two differential equations to describe the motion of the paint can on a spring, as demonstrated by equations 9 and 10. It was implemented using the coefficients from the PCA x- and y-coordinates matrix, using the first two values from the x-coordinates in the first equation and the first two values from the y-coordinates in the second equation (which were determined from the PCA demonstrating that two principal components were needed). The sparsity parameter provided a threshold to limit the number of elements found in the model, such that this threshold of 0.10 provided the model with two elements: the position and the velocity, in relation to the initial conditions. Equation 11 provides the model for describing the motion of the paint can along the z-axis; note that this SINDy algorithm is for the low-noise case.

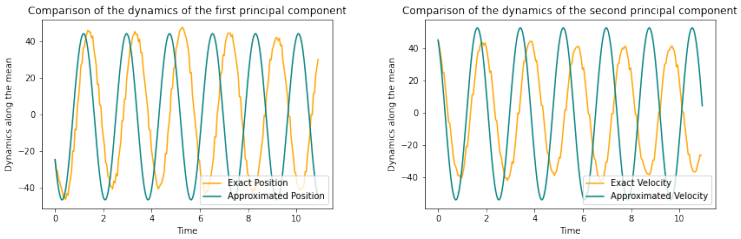
Figure 13, as an approximation of the actual position and velocity trajectories from figure 11 was quite successful. However, figure 14 was unable to reproduce the very complex trajectories of figure 12. This shall be further discussed below.

4 Computational Results

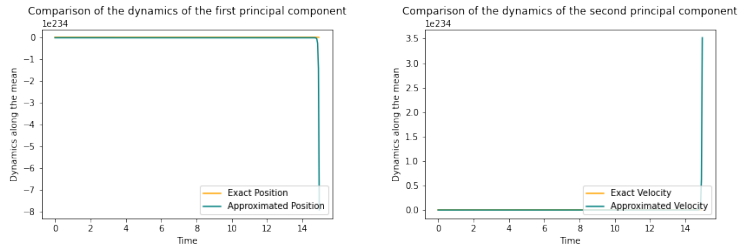
Overall, the low-noise condition provided the expected model; one including the position and velocity, and is quite descriptive of Newton's Second Law of Motion. Indeed, had we only included a single principal component, then we would have only been able to consider the positional argument, and would not be able to attribute acceleration to the model. We would have been able to differentiate for velocity, but the model itself would have been composed of a constant alone and this would not have been descriptive enough of the motion of the paint can in space.

Furthermore, as abovementioned, the low-noise condition models turned out to be quite successful, as demonstrated in figures 15 and 16. Figure 15 demonstrates the position as computed by the PCA and figure 16 the position by the SINDy model. Although they do not match perfectly, they are both oscillatory in nature; the discord may be due to the noise from the tracking, as the SINDy model has a very periodic trajectory, while the PCA trajectory is much less periodic.

Meanwhile, the high-noise condition models failed, which we believe to be due to the sparsity parameter (this is observed in figures 17 and 18). Due to the high level of noise in the video and in the tracking



Figures 15 and 16: Comparison of the PCA and SINDy Position and Velocity Trajectories in Low-Noise.



Figures 17 and 18: Comparison of the PCA and SINDy Position and Velocity Trajectories in High-Noise.

(the tracking box lost the target on several occasions), there may have been a need for more elements from the dictionary within the model, thus having a larger threshold to allow for more coefficients to be non-zero. Perhaps then a future direction would be to attempt the SINDy algorithm with four or even six elements in the model, considering positional and velocity arguments from other directions, as the principal components were not as clearly cut-off in terms of salience. In figure 10, the first principal component is considered quite important, but then components 2 and 3 are much closer and thus the latter may have been required for a more accurate model.

5 Summary and Conclusions

To conclude, the goal of the study was to model the motion of a paint can on a spring using the PCA and SINDy algorithm, in order to learn equations governing dynamics similar to Newton's Second Law of Motion. Two principal components were used within the model in order to represent position and velocity, both used in the calculation of acceleration in motion at a particular time in space. Although the low-noise condition demonstrated a more accurate representation, the high-noise condition lacked accuracy due to the nature of the model. We consider this to be due to not enough elements being used within the model to approximate the more complex motion captured by the videos.

References

[1] Jason Bramburger. *Data-Driven Methods for Dynamic Systems: A new perspective on old problems.* 2023.

[2] NASA - Glenn Research Center. "Newton's Laws of Motion". In: (2022).

[3] Ian T. Jollife and Jorge Cadima. "Principal component analysis: A review and recent developments". In: (2016).

[4] Brouton Lab. *A Complete Review of the OpenCV Object Tracking Algorithms.*

Appendix A Python Premade Functions

- `scipy.io.loadmat` Imports the Matlab array.

- `cv.VideoWriter` Converts an array into a video.
- `tracker types, cv.legacy` Declaring the tracker types from an existing library, using OpenCV.
- `np.zeros` Generates an array of zeros.
- `cv.VideoCapture` Calls the video forward and reads it.
- `cv.selectROI` Allows us to generate a box around the object of interest.
- `tracker.init` Initializes the tracker selected.
- `cv.rectangle` Draws the rectangle at every frame.
- `cv.destroyAllWindows()` Used to close the windows opened for object tracking.
- `np.resize` Resizes the specified matrix to the specified size.
- `np.reshape` Reshapes the specified matrix to the specified shape.
- `np.append` Stacks an array into an existing array.
- `np.mean` Generates the mean of the input matrix.
- `np.save` Saves a matrix as a file.
- `np.load` Loads a saved matrix back into the program.
- `lib.plot` Generates a plot (lib has many applications, for instance, generating labels and legends).
- `np.cov` Computes the covariance matrix of the argument.
- `np.linalg.svd` Generates the SVD of a matrix.
- `np.transpose` Generates the transpose of the input matrix.
- `np.matmul` Computes the matrix multiplication of two arguments.
- `py.SINDy` Defines the SINDy algorithm.
- `py.STLSQ` Defines the optimizer of the SINDy algorithm.
- `np.arange` Generates an array with evenly spaced values.
- `model.fit` Defining the data to the model.
- `odeint` Solves the ODE through integration.

Appendix B Python Code

The code for both videos is quite similar. The low noise code is the one included below.

```
##PROCEDURE:

# 1. set-up - import data
# 2. tracking
# 3. PCA / SVD
# 4. SINDy

! pip install numpy
import numpy as np
import scipy.io
import matplotlib.pyplot as lib
from PIL import Image as im
import cv2 as cv
```

```
##EXTRACTING THE VIDEOS FROM THE .MAT FILES - FOR EACH CAMERA (THREE ANGLES): Merci pour ton aide, Nada !

cam1 = scipy.io.loadmat("cam1_1.mat")
datacam1 = cam1["vidFrames1_1"]
#"vidFrames1_1" is the name of the element within the .mat file dictionary that is of interest
#extracting this for all three cameras

print("The dimensions of camera 1 are: ", datacam1.shape)
#print(cam1)
#print(datacam1)

cam2 = scipy.io.loadmat("cam2_1.mat")
datacam2 = cam2["vidFrames2_1"]

print("The dimensions of camera 2 are: ", datacam2.shape)
#print(cam2)
#print(datacam2)

cam3 = scipy.io.loadmat("cam3_1.mat")
datacam3 = cam3["vidFrames3_1"]

print("The dimensions of camera 3 are: ", datacam3.shape)
#print(cam3)
#print(datacam3)

#the dimensions are each printed as each camera shall have a different number of frames

##CONVERTING THE DATA INTO VIDEOS:
#the data should not be made into greyscale

#camera 1:

frame_height, frame_width, RGB, frames = datacam1.shape
fourcc = cv.VideoWriter_fourcc(*"mp4v")
filename = "vidcam1.mp4"
output = cv.VideoWriter(filename, fourcc = fourcc, fps = 20, frameSize = (frame_width, frame_height))
#cv.VideoWriter orders width by height

for i in range(frames):
    output.write(datacam1[:,:,:,i])

#camera 2:

frame_height, frame_width, RGB, frames = datacam2.shape
fourcc = cv.VideoWriter_fourcc(*"mp4v")
filename = "vidcam2.mp4"
output = cv.VideoWriter(filename, fourcc = fourcc, fps = 20, frameSize = (frame_width, frame_height))

for i in range(frames):
    output.write(datacam2[:,:,:,i])

#camera 3:

frame_height, frame_width, RGB, frames = datacam3.shape
fourcc = cv.VideoWriter_fourcc(*"mp4v")
filename = "vidcam3.mp4"
output = cv.VideoWriter(filename, fourcc = fourcc, fps = 20, frameSize = (frame_width, frame_height))

for i in range(frames):
    output.write(datacam3[:,:,:,i])

##DECLARING THE TRACKERS:
#reference: https://broutonlab.com/blog/opencv-object-tracking

tracker_types = ['BOOSTING', 'MIL', 'KCF', 'TLD', 'MEDIANFLOW', 'MOSSE', 'CSRT']
tracker_type = tracker_types[1]
```

```
if tracker_type == 'BOOSTING':
    tracker = cv.legacy.TrackerBoosting_create()
if tracker_type == 'MIL':
    tracker = cv.TrackerMIL_create()
if tracker_type == 'KCF':
    tracker = cv.TrackerKCF_create()
if tracker_type == 'TLD':
    tracker = cv.legacy.TrackerTLD_create()
if tracker_type == 'MEDIANFLOW':
    tracker = cv.legacy.TrackerMedianFlow_create()
if tracker_type == 'MOSSE':
    tracker = cv.legacy.TrackerMOSSE_create()
if tracker_type == "CSRT":
    tracker = cv.TrackerCSRT_create()
#to make the motion tracking more generalized, MIL was selected, as it has a better resistance to noise

##SAVING THE XY-COORDINATES OF THE TRACKING RECTANGLE:
#creating a matrix of zeros to then enter the coordinates of the box

xymat1 = np.zeros((0,225)) #for the first camera with 226 frames - the camera shall only generate 225 data
↪  points
xymat2 = np.zeros((0,225)) #for the second camera - but we need to compensate for the number of frames in the
↪  first camera
xymat3 = np.zeros((0,225)) #for the third camera - but we need to compensate for the number of frames in the
↪  first camera

##OBJECT TRACKING WITH OPENCV: CAMERA 1
#this code allows us to select a box around the can of paint

#get the video file and read it
video1 = cv.VideoCapture('vidcam1.mp4')
ret, frame = video1.read()

#selecting the bounding box in the first frame:
bbox = cv.selectROI(frame, False)
ret = tracker.init(frame, bbox)

##STORING THE XY-COORDINATES: CAMERA 1

X1 = np.zeros((226,1))
Y1 = np.zeros((226,1))

i = 0

while True:
    ret,frame = video1.read()
    if not ret:
        break
    (success,box) = tracker.update(frame)
    if success:
        (x,y,w,h) = [int(a) for a in box]
        cv.rectangle(frame, (x,y), (x+w,y+h), (250,0,250), 2) #image, start point, end point, colour (BGR),
        ↪  thickness
        XC1 = (x+w)/2
        YC1 = (y+h)/2
        print(XC1, YC1)
        X1[i] = XC1
        Y1[i] = YC1
    cv.imshow('Frame', frame) #need a positional argument, matrix form
    key = cv.waitKey(30)
    i += 1
    if key == ord('q'):
        break

video1.release()
cv.destroyAllWindows()

##OBJECT TRACKING WITH OPENCV: CAMERA 2
```

```
#this code allows us to select a box around the can of paint

#get the video file and read it
video2 = cv.VideoCapture('vidcam2.mp4')
ret, frame = video2.read()

#selecting the bounding box in the first frame:
bbox = cv.selectROI(frame, False)
ret = tracker.init(frame, bbox)

##STORING THE XY-COORDINATES: CAMERA 2
#as camera 1 has the fewest number of frames, we shall shape all

X2 = np.zeros((284,1))
Y2 = np.zeros((284,1))

i = 0

while True:
    ret,frame = video2.read()
    if not ret:
        break
    (success,box) = tracker.update(frame)
    if success:
        (x,y,w,h) = [int(a) for a in box]
        cv.rectangle(frame, (x,y), (x+w,y+h), (250,0,250), 2) #image, start point, end point, colour (BGR),
        ↪  thickness
        XC2 = (x+w)/2
        YC2 = (y+h)/2
        print(XC2, YC2)
        X2[i] = XC2
        Y2[i] = YC2
    cv.imshow('Frame', frame) #need a positional argument, matrix form
    key = cv.waitKey(30)
    i += 1
    if key == ord('q'):
        break

video2.release()
cv.destroyAllWindows()

##OBJECT TRACKING WITH OPENCV: CAMERA 3
#this code allows us to select a box around the can of paint

#get the video file and read it
video3 = cv.VideoCapture('vidcam3.mp4')
ret, frame = video3.read()

#selecting the bounding box in the first frame:
bbox = cv.selectROI(frame, False)
ret = tracker.init(frame, bbox)

##STORING THE XY-COORDINATES: CAMERA 3
#as camera 1 has the fewest number of frames, we shall shape all

X3 = np.zeros((232,1))
Y3 = np.zeros((232,1))

i = 0

while True:
    ret,frame = video3.read()
    if not ret:
        break
    (success,box) = tracker.update(frame)
    if success:
        (x,y,w,h) = [int(a) for a in box]
```

```
        cv.rectangle(frame, (x,y), (x+w,y+h), (250,0,250), 2) #image, start point, end point, colour (BGR),
        ↪  thickness
        XC3 = (x+w)/2
        YC3 = (y+h)/2
        print(XC3, YC3)
        X3[i] = XC3
        Y3[i] = YC3
    cv.imshow('Frame', frame) #need a positional argument, matrix form
    key = cv.waitKey(30)
    i += 1
    if key == ord('q'):
        break

video3.release()
cv.destroyAllWindows()

##SETTING UP XYMATRIX:

print(xymat1.shape)
print(xymat2.shape)
print(xymat3.shape)

X1 = np.resize(X1, (225,1)) #due to the size of the xy-matrix
Y1 = np.resize(Y1, (225,1)) #due to the size of the xy-matrix

X2 = np.resize(X2, (225,1)) #due to the size of the xy-matrix
Y2 = np.resize(Y2, (225,1)) #due to the size of the xy-matrix

X3 = np.resize(X3, (225,1)) #due to the size of the xy-matrix
Y3 = np.resize(Y3, (225,1)) #due to the size of the xy-matrix

X1 = np.reshape(X1, (1,225)) #to match the matrix format
Y1 = np.reshape(Y1, (1,225)) #to match the matrix format

X2 = np.reshape(X2, (1,225)) #to match the matrix format
Y2 = np.reshape(Y2, (1,225)) #to match the matrix format

X3 = np.reshape(X3, (1,225)) #to match the matrix format
Y3 = np.reshape(Y3, (1,225)) #to match the matrix format

xymat1 = np.append(xymat1, X1, axis = 0)
xymat2 = np.append(xymat2, X2, axis = 0)
xymat3 = np.append(xymat3, X3, axis = 0)

xymat1 = np.append(xymat1, Y1, axis = 0)
xymat2 = np.append(xymat2, Y2, axis = 0)
xymat3 = np.append(xymat3, Y3, axis = 0)

print(xymat1.shape)
print(xymat1)
print(xymat2.shape)
print(xymat2)
print(xymat3.shape)
print(xymat3)

xymat = np.zeros((0,225))
xymat = np.append(xymat, xymat1, axis = 0)
xymat = np.append(xymat, xymat2, axis = 0)
xymat = np.append(xymat, xymat3, axis = 0)
print(xymat.shape)
print(xymat)

##COMPUTING THE MEAN MATRIX:
#this shall allow to plot the X and Y values along the origin

#identifying the separate rows of the xy-matrix:

row1X = xymat[0,:]
```

```
row1Y = xymat[1,:]
row2X = xymat[2,:]
row2Y = xymat[3,:]
row3X = xymat[4,:]
row3Y = xymat[5,:]

#computing the mean of every row separately:

meanX1 = np.mean(row1X)
meanY1 = np.mean(row1Y)
meanX2 = np.mean(row2X)
meanY2 = np.mean(row2Y)
meanX3 = np.mean(row3X)
meanY3 = np.mean(row3Y)

#computing the rows without their mean:

xymat[0,:] = xymat[0,:] - meanX1
xymat[1,:] = xymat[1,:] - meanY1
xymat[2,:] = xymat[2,:] - meanX2
xymat[3,:] = xymat[3,:] - meanY2
xymat[4,:] = xymat[4,:] - meanX3
xymat[5,:] = xymat[5,:] - meanY3

print(xymat)
#the mean should be removed to run PCA as the results will look awful otherwise

np.save('xymat', xymat)
np.load('xymat.npy')

lib.plot(xymat[0,:])
lib.xlabel('Frames')
lib.ylabel('Difference along the mean')
lib.title('Difference along the mean through time, camera 1-X')
lib.savefig("1-X.png")
lib.show()

lib.plot(xymat[1,:])
lib.xlabel('Frames')
lib.ylabel('Difference along the mean')
lib.title('Difference along the mean through time, camera 1-Y')
lib.savefig("1-Y.png")
lib.show()

lib.plot(xymat[0,:], c = 'orange', label = 'X-direction')
lib.plot(xymat[1,:], c = 'teal', label = 'Y-direction')
lib.xlabel('Frames')
lib.ylabel('Difference along the mean')
lib.title('Comparison of the motion of the can on camera 1')
lib.legend(loc = 'lower right', prop = {'size':10})
lib.savefig("1.png")
lib.show()

lib.plot(xymat[2,:])
lib.xlabel('Frames')
lib.ylabel('Difference along the mean')
lib.title('Difference along the mean through time, camera 2-X')
lib.savefig("2-X.png")
lib.show()

lib.plot(xymat[3,:])
lib.xlabel('Frames')
lib.ylabel('Difference along the mean')
lib.title('Difference along the mean through time, camera 2-Y')
lib.savefig("2-Y.png")
lib.show()

lib.plot(xymat[2,:], c = 'orange', label = 'X-direction')
```

```
lib.plot(xymat[3,:], c = 'teal', label = 'Y-direction')
lib.xlabel('Frames')
lib.ylabel('Difference along the mean')
lib.title('Comparison of the motion of the can on camera 2')
lib.legend(loc = 'lower right', prop = {'size':10})
lib.savefig("2.png")
lib.show()

lib.plot(xymat[4,:])
lib.xlabel('Frames')
lib.ylabel('Difference along the mean')
lib.title('Difference along the mean through time, camera 3-X')
lib.savefig("3-X.png")
lib.show()

lib.plot(xymat[5,:])
lib.xlabel('Frames')
lib.ylabel('Difference along the mean')
lib.title('Difference along the mean through time, camera 3-Y')
lib.savefig("3-Y.png")
lib.show()

lib.plot(xymat[4,:], c = 'orange', label = 'X-direction')
lib.plot(xymat[5,:], c = 'teal', label = 'Y-direction')
lib.xlabel('Frames')
lib.ylabel('Difference along the mean')
lib.title('Comparison of the motion of the can on camera 3')
lib.legend(loc = 'lower right', prop = {'size':10})
lib.savefig("3.png")
lib.show()

#it makes sense that the major difference would appear in the y-values of cameras 1 and 2 and the x-values of
↪  camera 3
#it further makes sense that the difference appears periodic
#this means that the approximation of the motion using SINDy will require trigonometric functions
#(or linear pendulum equation / polynomial functions that approximate the differentials of the positional
↪  information)
#PCA shall keep two degrees of information, as according to Newton's second law, position and velocity are
↪  required

##RUNNING PCA: STEP 1 - COVARIANCE MATRIX
#this allows to determine the dominant spatial directions of motion (how one moves around in the projected
↪  plane)

covmat = np.cov(xymat)

##DIAGONALIZING THROUGH SVD: REMOVING REDUNDANCIES

U, S, VT = np.linalg.svd(xymat)
UT = np.transpose(U)

##RUNNING PCA: STEP 2 - COVARIANCE MATRIX WITHOUT TIME

Y = np.matmul(UT, xymat) #these allow for a change in basis, effectively removing time
covmatY = np.cov(Y) #determining the dominant spatial directions of motion without time

print(covmatY.shape)
print(covmatY) #it is a diagonal matrix

print(S.shape)
print(S)
#the first two principal components maintain the majority of the information

lib.plot(S, marker = 'o', markersize = 5)
lib.xlabel('Order of the principal components')
lib.ylabel('Strength of the principal components')
lib.title('Principal Components')
lib.savefig("Principal_Components.png")
```

```
lib.show()

lib.plot(S[0]*VT[0,:], c = 'orange', label = 'Position') #this is velocity (we need the time component here)
lib.plot(S[1]*VT[1,:], c = 'teal', label = 'Velocity') #this is acceleration (we need the time component here)
lib.xlabel('Frames')
lib.ylabel('Dynamics along the mean')
lib.title('Dynamics of the two first principal components')
lib.legend(loc = 'lower right', prop = {'size':10})
lib.savefig("PCA.png")
lib.show()
#as velocity is the derivative of position, the first component shall represent position, and the second,
↪  velocity

! pip install pysindy
import pysindy as py

model = py.SINDy()
print(model)

xymat_reduced = np.transpose(U[:,0:2])@xymat

print(xymat_reduced.shape)
print(xymat_reduced)

xymat_reducedT = np.transpose(xymat_reduced)
print(xymat_reducedT.shape)
print(xymat_reducedT)

X1 = xymat_reducedT[:,0]
Y1 = xymat_reducedT[:,1]

featurenames = ['X1', 'Y1']
optim = py.STLSQ(threshold = 0.1)
t = np.arange(0,11,11/226)

model = py.SINDy(feature_names = featurenames, optimizer = optim)

model.fit(xymat_reducedT, t = 11/226) #frames by the duration of the video
print(model)

from scipy.integrate import odeint

def function(z,t):
    a=-2.454
    b=-3.004
    c=4.509
    d=4.143

    dxdt = a + b*z[1]
    dydt = c + d*z[0]
    dzdt = [dxdt,dydt]

    return dzdt

##INITIAL CONDITION:

z0 = [xymat_reducedT[0,0], xymat_reducedT[0,1]]

#solving the ODE:
z = odeint(function, z0, t)

lib.plot(t,z[:,0], label = 'Position') #x(t) = position
lib.plot(t,z[:,1], label = 'Velocity') #y(t) = velocity
lib.xlabel('Time')
lib.ylabel('Dynamics along the mean')
lib.title('Approximation of the dynamics of the two first principal components')
lib.legend(loc = 'lower right', prop = {'size':10})
lib.savefig("SINDy.png")
```

```
lib.show()

lib.plot(t[0:225], S[0]*VT[0,:], c = 'orange', label = 'Exact Position') #exact
lib.plot(t, z[:,0], c = 'teal', label = 'Approximated Position') #approximation
lib.xlabel('Time')
lib.ylabel('Dynamics along the mean')
lib.title('Comparison of the dynamics of the first principal component')
lib.legend(loc = 'lower right', prop = {'size':10})
lib.savefig("Compare1.png")
lib.show()

lib.plot(t[0:225], S[1]*VT[1,:], c = 'orange', label = 'Exact Velocity') #exact
lib.plot(t, z[:,1], c = 'teal', label = 'Approximated Velocity') #approximation
lib.xlabel('Time')
lib.ylabel('Dynamics along the mean')
lib.title('Comparison of the dynamics of the second principal component')
lib.legend(loc = 'lower right', prop = {'size':10})
lib.savefig("Compare2.png")
lib.show()
```