MATH 494 Homework 2

Nada El Arabi

March 7, 2023

Abstract

In this report, we discuss taking a data-driven approach to "rediscover" the laws of physics underlying two simple spring-mass systems, one where no noise is present and another where some disruption of the motion is introduced, through the use of principal component analysis and sparse identification of nonlinear dynamics and examine their respective limitations. We begin by analyzing three videos each representing a coordinate system capturing the motion of a bucket of paint springing up and down to extract a number of principal components from the positional data produced by tracking the centre of the bucket frame to frame. We then use the principal components to apply the SINDy algorithm to uncover the second order ordinary differential equation representing a simple harmonic oscillator whose solution is the vertical displacement of the bucket through time.

1 Introduction and Overview

Discovering the governing equations underlying dynamical systems has the potential to provide incredible new insights in fields such as meteorology, genetics, epidemiology, finance and many more[2]. One challenge that is ubiquitously faced when attempting to learn from data is the size and high-dimensionality of the data sets. Principal component analysis is a technique which allows us to take high-dimensional data and use the redundancy found amongst the variables to represent the data in a lower-dimensional basis made of principal components which represent directions of highest variance. These directions of highest variance can then be reasonably used as potential candidate variables of governing equations of dynamical systems as they capture the largest "variations" in our data.

In this paper, we attempt to uncover the equations which describe the motion of a mass attached to a single spring. Of course, the governing laws of such a system are well known as Newton's second law and Hooke's first law [1]. We use this example from classical mechanics to demonstrate the effectiveness of PCA allied with the Sparse Identification of Nonlinear Dynamics algorithm. Our data is in the form of three videos which capture the same movement of a bucket attached to as spring. From these videos, we extract the displacement of the bucket in the x, y directions respective to each camera's position and posit these 6 coordinate variables as our high-dimensional redundant data. We determine two principal components and use them as a new basis for our data and attempt to recover the second order differential equation which characterizes such a motion. We do these manoeuvres twice: once with data containing minimal noise and a second time with deliberate addition of noise.

2 Theoretical Background

Suppose we have a data matrix: $\mathbf{X} = \begin{bmatrix} \mathbf{x_1} \\ \mathbf{x_2} \\ \vdots \\ \mathbf{x_n} \end{bmatrix}$

where $\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_n}$ each represent a k-length row vector of real-valued measurements from a single experiment. Suppose further that each row has a mean of zero. We know from statistical theory that the variance of $\mathbf{x_1}$ and the covariance between $\mathbf{x_1}, \mathbf{x_2}$ are respectively:

$$\sigma^2_{\mathbf{x_1}} = \mathbf{x_1}\mathbf{x_1}^T \qquad \sigma^2_{\mathbf{x_1}\mathbf{x_2}} = \mathbf{x_1}\mathbf{x_2}^T$$

Additionally, we note that variance is a measure of the spread of the data and that covariance tells us how variables relate to one another: a covariance score of zero indicating statistical independence while a covariance score of one indicates a that the variables are identical.

Computing the covariance matrix of X would therefore give us much information on how our variables correlate to each other. We may compute it in the following way:

$$C_{\mathbf{X}} = \frac{1}{n-1}\mathbf{X}\mathbf{X}^T = \begin{bmatrix} \sigma^2_{\mathbf{x_1 x_1}} & \sigma^2_{\mathbf{x_1 x_2}} & \cdots & \sigma^2_{\mathbf{x_1 x_n}} \\ \sigma^2_{\mathbf{x_2 x_1}} & \sigma^2_{\mathbf{x_2 x_2}} & \cdots & \sigma^2_{\mathbf{x_2 x_n}} \\ \vdots & \ddots & \ddots & \vdots \\ \sigma^2_{\mathbf{x_n x_1}} & \sigma^2_{\mathbf{x_n x_2}} & \cdots & \sigma^2_{\mathbf{x_n x_n}} \end{bmatrix}$$

We remark that $C_{\mathbf{X}}$ is a symmetric matrix as the $i-jth$ entry is always equal to the $j-ith$ entry and now recall our previously discussed goal to reframe our data in terms of a new basis where variance is maximized and redundancy is minimized. To that end, we note that the off-diagonals in our covariance matrix $C_{\mathbf{X}}$ are the covariance scores of every combination of variables and that the diagonals represent the variance scores for each variable. It follows then that our goal is to diagonalize $C_{\mathbf{X}}$ and since $C_{\mathbf{X}}$ is symmetric it can be diagonalized as:

$$\mathbf{X}\mathbf{X}^T = \mathbf{P}\mathbf{D}\mathbf{P}^{-1}$$

where $\mathbf{P}$ is a unitary matrix whose rows are orthogonal eigenvectors and thus form a basis where variance is maximized and variables are statistically independent of one another [1]. It follows that the rows of $\mathbf{P}$ (or the columns of $\mathbf{P}^{-1}$, as $\mathbf{P}^{-1} = \mathbf{P}^T$) are indeed our principal components. Our data can thus be appropriately reframed by left-multiplying $\mathbf{X}$ by $\frac{1}{n-1}\mathbf{P}^T$.

It is note worthy that we can prove that the singular value decomposition can also be used in lieu of eigendecomposition to find the principal components. To illustrate, we take the SVD of $\mathbf{X}$:

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$$

$$\begin{aligned} \implies \mathbf{X}\mathbf{X}^T &= \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\left(\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\right)^T \\ &= \mathbf{U}\mathbf{\Sigma}^2\mathbf{U}^T \\ &= \mathbf{P}\mathbf{D}\mathbf{P}^{-1} \end{aligned}$$

where $\mathbf{U}$ is the orthogonal, unitary matrix of left singular vectors and $\mathbf{\Sigma}$ is the diagonal matrix of singular values [1]. The last equality gives us our correspondence between the eigendecomposition of the covariance matrix $C_{\mathbf{X}}$ and the singular value decomposition of the original matrix $\mathbf{X}$: $\mathbf{U}^T$ is the analogous matrix of basis vectors and our singular values squared are the eigenvalues' counterparts. We may then reframe our data by similarly left-multiplying $\mathbf{X}$ by $\frac{1}{n-1}\mathbf{U}^T$.

MAYBE WRITE SOME STUFF ABOUT SINDy!

3 Algorithm Implementation and Development

The first goal of our objective consists of loading the video data matrices and reconstructing the videos so that we may observe the nature of the motion. Refer to lines 0-53 in the Python code given in Appendix C (we note that only the code for the noisy data is provided as the algorithm is implemented much the same across the noisy and non-noisy cases). As discussed previously, each video captures a perspective on the motion of a bucket springing up and down. To track the motion, we chose to make use of a legacy motion tracker from openCV. A number of trackers are listed (see lines 68-76) in Appendix C and after some experimentation, we've been able to gain the best results with the "csrt" tracker. Through the use of openCV's API, we're able to display the first frame of each video and draw a rectangle around the bucket which acts as our object marker. It is then the coordinates of the centre of that rectangle that we track frame to frame. (see lines 83-128)

It is important to note that at this stage, we are introducting additional noise to our data as the motion tracker is unable to capture the motion in the single direction. This is illustrated in Figure 1 where we notice that, though there is a dominant direction, notably the y direction in subfigure 1a and subfigure 1b and the x direction in subfigure 1c, the minor axis also captures motion. In an ideal system, the curves along the minor axes would be completely flat as we know from theory that the motion is one-dimensional.

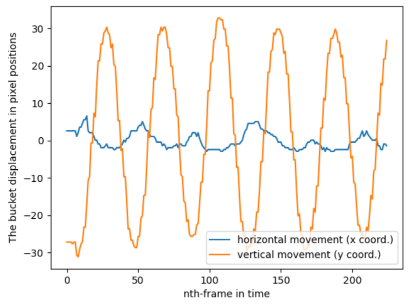
It is also in this step of the implementation that the noisy and non-noisy analyses diverge. Where we can observe in Figure 1 that a small quantity of disturbance is introduced, we can see in Figure 3 (see Appendix A) that the disturbance in the minor axis is significantly more pronounced and that the dominant axis' motion is captured a lot less smoothly.

The coordinates of the centre of the motion-tracking rectangle are saved as row variables, which form our previous discussed $\mathbf{X}$ data matrix. We then proceed to subtract the mean from each row of $\mathbf{X}$ (see lines 231-253). This is necessary for the adequate separation of the motion from the static background of the video by the singular value decomposition [1]. We compute the SVD with the help of Python's numpy library and find the principal components in the vectors composing the matrix $\mathbf{U}^T$.

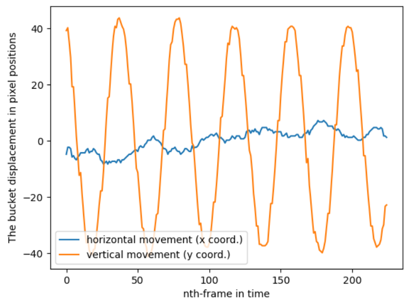
We choose to pick two principal components to form our new basis as they are sufficient to capture the two most important variables which characterize our system: acceleration and velocity. We proceed to change bases using these two principal component to use the SINDy algorithm.

4 Computational Results

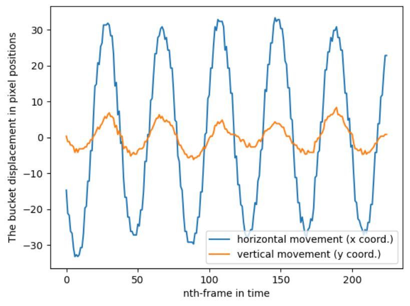
We were able to successfully track the oscillating motion of the bucket in time as showed in Figure 1. Principal component analysis also proves useful as the oscillating nature of acceleration and velocity is captured in subfigure 2c. The challenges of dealing with noisy data are illustrated in Figure 3 and Figure 4 in Appendix A.



(a) first camera

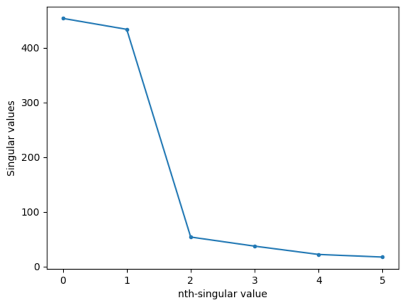


(b) second camera

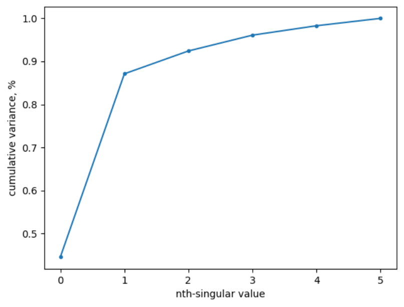


(c) third camera

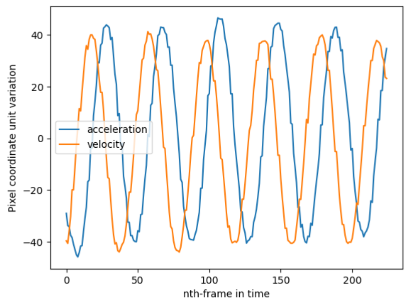
Figure 1: Displacement with respect to original coordinate position: minimal noise case



(a) Plot of singular values



(b) Cumulative sum of variance captured by first nth-singular vectors



(c) Evolution of acceleration and velocity in time

Figure 2: Graphs of study of singular values and principal components

5 Summary and Conclusions

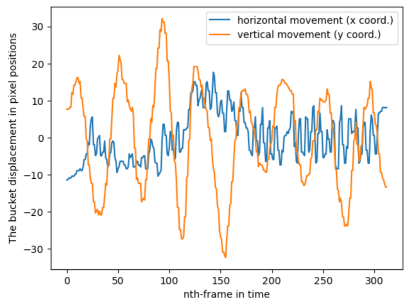
Through the use of a library of motion trackers, principal component analysis via the singular value decomposition, and finally sparse identification of non-linear dynamics, we are able to recover the governing equations of a simple spring-mass system. We made an attempt to arrive to similar ends with motion where disturbance was introduced, but noise proved to be a significant hurdle and no conclusive results could be extracted from the data.

References

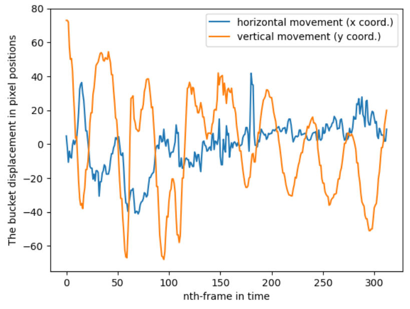
[1] Jason Bramburger. *Data-Driven Methods for Dynamic Systems: A new perspective on old problems, lectures 11-13*. 2023.

[2] Steven Brunton et al. *Discovering governing equations from data by sparse identification of nonlinear dynamical systems*. Mar. 2016. URL: `https://www.pnas.org/doi/10.1073/pnas.1517384113`.

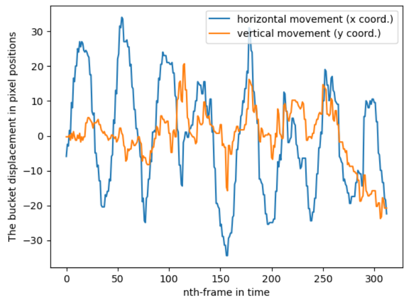
Appendix A Additional graphs



(a) first camera

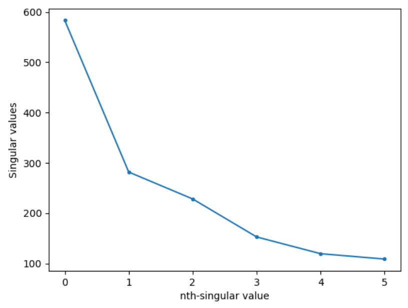


(b) second camera

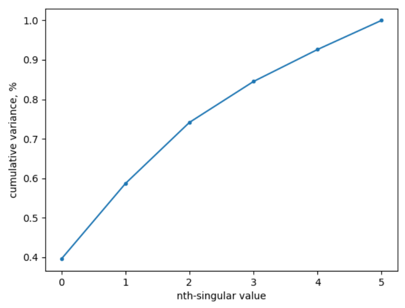


(c) third camera

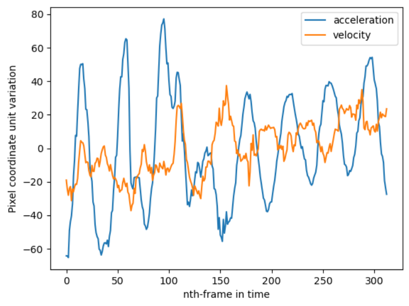
Figure 3: Displacement with respect to original coordinate position: noisy case



(a) Plot of singular values



(b) Cumulative sum of variance captured by first nth-singular vectors



(c) Evolution of acceleration and velocity in time

Figure 4: Graphs of study of singular values and principal components

Appendix B Python Functions

This is how to make an **unordered** list:

- `y = linspace(x1,x2,n)` returns a row vector of `n` evenly spaced points between `x1` and `x2`.
- `[X,Y] = meshgrid(x,y)` returns 2-D grid coordinates based on the coordinates contained in the vectors `x` and `y`. X is a matrix where each row is a copy of `x`, and `Y` is a matrix where each column is a copy of `y`. The grid represented by the coordinates `X` and `Y` has `length(y)` rows and `length(x)` columns.

Appendix C Python Code

```

import scipy.io
import numpy as np
import matplotlib.pyplot as plt
import cv2 as cv


# In[2]:


#For each camera, we extract the videos from the mat files
Mat1 = scipy.io.loadmat('/Users/nadaelarabi/MATH-494/M494H2/cam1_2.mat')
dataMat1 = Mat1["vidFrames1_2"]


# In[7]:

```

```

Mat2 = scipy.io.loadmat('/Users/nadaelarabi/MATH-494/M494H2/cam2_2.mat')
dataMat2 = Mat2["vidFrames2_2"]


# In[8]:


Mat3 = scipy.io.loadmat('/Users/nadaelarabi/MATH-494/M494H2/cam3_2.mat')
dataMat3 = Mat3["vidFrames3_2"]


# In[9]:


#We need to determine the dimensions of each of the camera to see what dimensions our data
    matrix should have
print("The dimensions of cam1's take are: ",dataMat1.shape," cam2's: ",dataMat2.shape," cam3
    's: ",dataMat3.shape)


# In[17]:


#We write the videos necessary to capture the motion from each camera
#We only need to change the dataMati and the name of the videos for each camera take, I
    wrote them out as
#camera1, camera2, camera3
#Sometimes necessary to first write the filename as mp4 inside 'filename'
height, width,_,nbFrames = dataMat1.shape
codec_id = "mp4v" # ID for a video codec.
fourcc = cv.VideoWriter_fourcc(*codec_id)
filename = "/Users/nadaelarabi/MATH-494/M494H2/camera1Noise.mp4v"
out = cv.VideoWriter(filename, fourcc=fourcc, fps=20, frameSize=(width, height))


for i in range(nbFrames):
    out.write(dataMat1[:,:,:,i])


# In[29]:


#Here we declare the matrix that will store our x and y coordinates
x_y_coordinateMatrix = np.zeros((0,313))


# In[55]:


#Now we can set up the motion tracking

#First, we begin by setting up a dictionary of legacy motion trackers in opencv
TrDict = {
    'csrt' : cv.legacy.TrackerCSRT_create,
    'mosse' : cv.legacy.TrackerMOSSE_create,
    'kcf' : cv.legacy.TrackerKCF_create,
    'medianflow': cv.legacy.TrackerMedianFlow_create,
    'mil' : cv.legacy.TrackerMIL_create,
    'tld' : cv.legacy.TrackerTLD_create,
    'boosting' : cv.legacy.TrackerBoosting_create
}


# In[56]:


#Now we initialize the tracker, csrt was chosen after testing with the other trackers and
```

```
    was found to be the best
tracker = TrDict['csrt']()


# In[57]:


#The code below allows us to manually draw a rectangle that will track the motion of the
    bucket
#Recommended to draw a larger box as it makes it less susceptible to be thown off

v = cv.VideoCapture('camera3Noise.mp4')
ret, frame = v.read()
cv.imshow('Frame',frame)
bb = cv.selectROI('Frame',frame)
tracker.init(frame,bb)



# In[58]:


#Displaying and storing the coordinates into arrays
x_coordinates = np.zeros((356,1))
y_coordinates = np.zeros((356,1))
index = 0
while True:
    ret,frame = v.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)
        x_centerCoordinate = (x+w)/2
        y_centerCoordinate = (y+h)/2
        print(x_centerCoordinate,y_centerCoordinate)
        x_coordinates[index] = x_centerCoordinate
        y_coordinates[index] = y_centerCoordinate
    cv.imshow('Frame',frame)
    key = cv.waitKey(30)
    index = index+1
    if key == ord('q'):
        break

v.release()
cv.destroyAllWindows()


# In[59]:


#Here we will resize our coordinate arrays so that they are an appropriate shape
#Since the camera take with the fewest number of frames generates 225 data points,
x_coordinates = np.resize(x_coordinates,(313,1))
y_coordinates = np.resize(y_coordinates,(313,1))


# In[60]:


#Now we can reshape the coordinates to make them one long row
x_coordinates = np.reshape(x_coordinates,(1,313))
y_coordinates = np.reshape(y_coordinates,(1,313))


# In[61]:

```

```

np.shape(x_y_coordinateMatrix)


# In[62]:


x_y_coordinateMatrix = np.append(x_y_coordinateMatrix,x_coordinates,axis=0)


# In[63]:


x_y_coordinateMatrix = np.append(x_y_coordinateMatrix,y_coordinates,axis=0)


# In[64]:


np.shape(x_y_coordinateMatrix)


# In[65]:


x_y_coordinateMatrix


# In[66]:


np.save('x_y_coordinateMatrixNOISE',x_y_coordinateMatrix)


# In[67]:


#Here we import the data matrix
x_y_coordinatesMat = np.load('/Users/nadaelarabi/MATH-494/M494H2/x_y_coordinateMatrixNOISE.
    npy')


# In[72]:


plt.plot(x_y_coordinatesMat[0,:], label = "horizontal movement (x coord.)")
plt.plot(x_y_coordinatesMat[1,:],label = "vertical movement (y coord.)")
#plt.title('Displacement with respect to original coordinate position, 1st camera, minimal
    noise')
plt.xlabel('nth-frame in time')
plt.ylabel('The bucket displacement in pixel positions')
plt.legend()

plt.savefig('Displ_camera1_NOISE.png')


# In[73]:


plt.plot(x_y_coordinatesMat[2,:],label = "horizontal movement (x coord.)")
plt.plot(x_y_coordinatesMat[3,:],label = "vertical movement (y coord.)")
#plt.title('Displacement with respect to original coordinate position, 2nd camera, minimal
    noise')
plt.xlabel('nth-frame in time')
plt.ylabel('The bucket displacement in pixel positions')
plt.legend()

plt.savefig('Displ_camera2_NOISE.png')
```

```


# In[74]:


plt.plot(x_y_coordinatesMat[4,:],label = "horizontal movement (x coord.)")
plt.plot(x_y_coordinatesMat[5,:],label = "vertical movement (y coord.)")
#plt.title('Displacement with respect to original coordinate position, 3rd camera, minimal
    noise')
plt.xlabel('nth-frame in time')
plt.ylabel('The bucket displacement in pixel positions')
plt.legend()
plt.savefig('Displ_camera3_NOISE.png')


# In[71]:


#Let's try to compute the mean a bit differently
row1_x = x_y_coordinatesMat[0,:]
row1_y = x_y_coordinatesMat[1,:]
row2_x = x_y_coordinatesMat[2,:]
row2_y = x_y_coordinatesMat[3,:]
row3_x = x_y_coordinatesMat[4,:]
row3_y = x_y_coordinatesMat[5,:]

mean_x1 = np.mean(row1_x)
mean_y1 = np.mean(row1_y)
mean_x2 = np.mean(row2_x)
mean_y2 = np.mean(row2_y)
mean_x3 = np.mean(row3_x)
mean_y3 = np.mean(row3_y)

x_y_coordinatesMat[0,:] = x_y_coordinatesMat[0,:] - mean_x1
x_y_coordinatesMat[1,:] = x_y_coordinatesMat[1,:] - mean_y1
x_y_coordinatesMat[2,:] = x_y_coordinatesMat[2,:] - mean_x2
x_y_coordinatesMat[3,:] = x_y_coordinatesMat[3,:] - mean_y2
x_y_coordinatesMat[4,:] = x_y_coordinatesMat[4,:] - mean_x3
x_y_coordinatesMat[5,:] = x_y_coordinatesMat[5,:] - mean_y3

x_y_coordinatesMat


# In[75]:


u,s,vt = np.linalg.svd(x_y_coordinatesMat)


# In[76]:


ut = u.transpose()


# In[77]:


Y = np.matmul(ut,x_y_coordinatesMat)


# In[78]:


covarianceMat_Y = np.cov(Y)


# In[79]:
```

```


covarianceMat_Y


# In[80]:


#We can plot the singular values
plt.plot(s,marker='.')
plt.xlabel('nth-singular value')
plt.ylabel('Singular values')
#plt.title('Plot of singular values of minimal noise data')
plt.savefig('svNOISE.png')


# In[81]:


plt.plot(np.cumsum(s)/np.sum(s), marker='.')
plt.xlabel('nth-singular value')
plt.ylabel('cumulative variance, %')
#plt.title('Cumulative sum of variance captured by first nth-singular vectors ')
plt.savefig('svCumulativeNOISE.png')


# In[82]:


plt.plot(s[0]*vt[0,:], label = 'acceleration')
plt.plot(s[1]*vt[1,:], label = 'velocity')
plt.xlabel('nth-frame in time')
plt.ylabel('Pixel coordinate unit variation')
#plt.title('Evolution of acceleration and velocity in time')
plt.legend()
plt.savefig('avNOISE.png')



#Note that the blue curve is acceleration and the orange curve is its derivative, velocity
#s[0]*vt[0,:], s[1]*vt[1,:]


# In[83]:


import pysindy as ps
x_y_coordinatesMat


# In[84]:


x_y_coordinates_reduced = (u[:,0:2].transpose())@x_y_coordinatesMat


# In[85]:


x_y_coordinates_reduced.shape


# In[86]:


x_y_coordinates_reduced = x_y_coordinates_reduced.transpose()


```

```python
# In[87]:


x = x_y_coordinates_reduced[:,0]
y = x_y_coordinates_reduced[:,1]


# In[88]:


featureNames = ['x','y']


# In[101]:


opt = ps.STLSQ(threshold = 0.1)


# In[102]:


model = ps.SINDy(feature_names = featureNames,optimizer = opt)


# In[103]:


model.fit(x_y_coordinates_reduced,t = 15/314)


# In[104]:


model.print()


# In[ ]:
```