MATH 494-A: Final Assignment
Identifying Dynamics in Origami Cranes

Myriam Boucher-Pinard (40109509) and Alexander Vakkas (40098027)

April 24, 2023

Abstract

This study has the aim of tracking and analyzing the motion of origami as the four corners of the paper fold into the traditional crane, using Principal Component Analysis (PCA) and Sparse Identification of Nonlinear Dynamics (SINDy) on three videos at the x-, y-, and z-axes. This is accomplished by approximating the motion of the folds in space and time, through the SINDy algorithm. Particularly, it was found that one principal component is typically required to do so up to some error, most likely due to tracking method error. Furthermore, model comparison is explored organically due to the symmetry of the origami crane.

1 Introduction and Overview

Although paper originated in China, it was in Japan's Edo Era (1603 - 1868) that the popularization of origami occurred [3]. From a purely ceremonial activity due to the high cost of paper to a more leisurely activity, origami now permeates both the arts and sciences. In 1797, the *Sembazuru orikata*, or "The Secret of Folding Cranes" by Akisato Rito, was first published, and is considered to be the first known written description of paper folding. Within it, historically the first model published, is the origami crane [3].

In this study, we shall be analyzing the motion of the origami crane as it folds itself from a flat simulation-generated piece of paper to its final form and approximating its trajectory using the Sparse Identification of Nonlinear Dynamics (SINDy) algorithm. We will be analyzing the motion from the x-, y-, and z-axes. The data first needs to be pruned to remove redundancy, which can be achieved using the Singular Value Decomposition (SVD) method. This method will identify the dimensions that contain the most relevant information and allow the algorithm to draw the majority of its data from those dimensions [2]. The data shall then be processed using Principal Component Analysis (PCA), translating the high-dimensional information to linear functions, identifying their orthogonal directions. These linear functions are computed through the covariance matrix, by the SVD, and preserve as much variability as possible by ordering the orthogonal functions by variance [6]. The study is completed through the use of the SINDy algorithm, implementing the principal components to the model as an approximation of the motion of the origami folding. Although the models shall vary in appearance, we will discuss the reason for mainly maintaining one dimension for the models. Finally, in order to reduce the repetitiveness of the explanations, we will further be focusing on the beak data.

2 Theoretical Background

2.1 Understanding Origami in Mathematics

While origami originated as a form of art, it now has its place in the sciences. When learning to fold origami, we are typically introduced to the concept through the standard folding instructions; that is, how to fold a flat piece of paper into the final form of the model. However, the more compelling way of analyzing origami from a mathematical perspective might be through the crease pattern, or the pattern formed by the folds when the model is unfolded, as observed in figure 1.

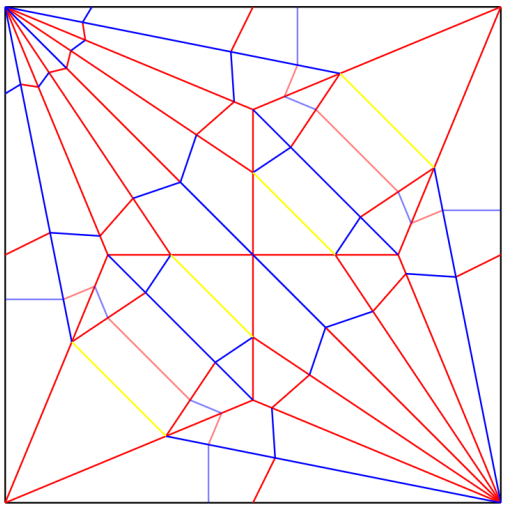


Figure 1: Crease pattern of the origami crane.

The crease pattern allows for a more holistic observation of the model, as is necessary for circle packing. The method of circle packing is the method of delineating areas within the paper that shall be dedicated to certain parts of the model [4]. For the crane, this means determining where the beak (and neck), wings, and tail shall be, as well as defining the back of the crane, which is the central vertex of the model. This optimization of the use of the paper allows for the us to define key vertices, as well as determine whether the fold shall be valley or mountain (stated differently, a $\vee$ or $\wedge$ fold, respectively). Although most crease patterns are now created through the use of an algorithm with polynomial complexity, the optimization problem posed by origami models remains [4]. So much so, in fact, that the crane is so efficiently optimized that it is in fact a solution to the generalization of the Margulis napkin problem [1]. Indeed, taking the projection of the crane (standing upright as it is typically presented), its perimeter is larger than the original sheet of paper the crane was folded with. Hence, due to the underlying mathematical implications of the origami crane, it seemed appropriate to further explore that which the model had to offer, as we do so below.

2.2 Principal Component Analysis (PCA)

$$\sigma_x^2 = \frac{1}{n-1}xx^T,\ \sigma_y^2 = \frac{1}{n-1}yy^T,\ \sigma_{xy}^2 = \frac{1}{n-1}xy^T \tag{1}$$

Principal component analysis draws from its use in determining the dominant spatial directions of motion, or, the way in which the four corners of the origami model move 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 [2, 6]. This is accomplished through variance first, and then covariance (1).

Due to the size of the data set, being the tracking of the motion of four corners in three different axes, and considering 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, their variance may be computed using σ_x^2 and σ_y^2 (1); in our case, the x- and y-directions of the motion tracker as it moves in space. Computing the covariance conversely depends on the way in which the variables vary from one another, and is thus computed using σ_{xy}^2 (1) [2]. 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, the PCA prunes the redundant information so that the most important information is defined in our later model.

As mentioned above, the aim 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).

$$X = U\Sigma V^* \tag{2}$$

Matrix X (2), composed of orthogonal matrices U and V, respectively represent space and time [2]. The matrices are not unitary due to the nature of the model; indeed, there are no imaginary numbers.

$$Y = U^T X \tag{3}$$

Returning to PCA, we are essentially looking for a change in basis. If we let X (3) 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.

$$\sigma_Y = \Sigma^2 \tag{4}$$

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 Σ^2 (4) 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 (4).

2.3 Sparse Identification of Nonlinear Dynamics (SINDy)

The diagonal matrix of principal components computed, the Sparse Identification of Nonlinear Dynamics (SINDy) algorithm is now necessary to approximate the motion of each of the corners as they fold into the model.

$$\begin{aligned} \dot{X} &= F(X) \\ \dot{X} &= \Xi\Theta(X) \\ \Xi &= Y\Theta(X)^\dagger \end{aligned} \tag{5}$$

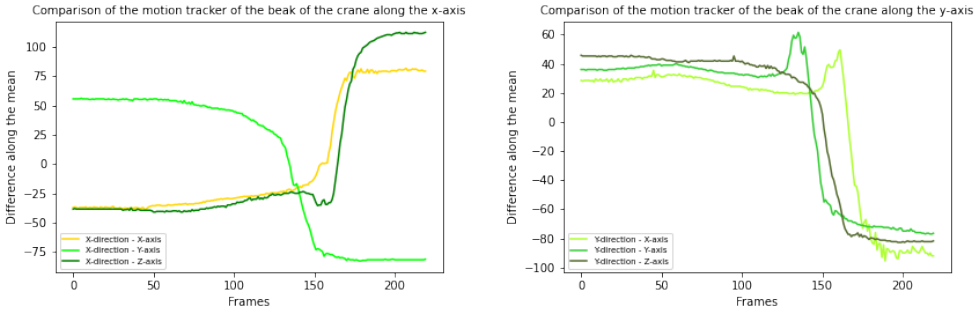
The SINDy method is a sparse identification approach for nonlinear dynamical systems that combines the dynamical system discovery problem with a statistical regression problem [2]. The SINDy method attempts to find the best dynamical system F for the data $\dot{X}$. This function approximation problem is expressed as the linear regression for $\dot{X}$, with coefficients Ξ and a regression term library $\Theta(X)$ (5). To begin, we generate the data from X, a dynamical system, and then compute its derivative $\dot{X}$. $\Theta(X)$ is then defined as a matrix of columns each representing a moment in time, the following column the next moment in time (represented by Y). Finally, the coefficients Ξ may be computed using the next moment in time (5).

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 motion; in our case, polynomial terms up to the second degree. 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 high-dimensional motion in space and time, solving at least one differential equation to describe the motion becomes necessary [2].

$$X(t) = X(0) + \int_0^t F(X(s))ds \tag{6}$$

Applying the Riemann sum approximation (6), the function becomes discrete.

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 [2]. Thus, this limits the complexity of the model, by ensuring that the most important elements have the most effect. However, we may note that the sparsity parameter used for this study is quite small, as the SINDy model would have otherwise been a linear model, which shall be further explored below.



Figures 2 and 3: Tracking the motion of the beak along the x-, y-, z-axes for the x- and y-directions of the coordinates.

3 Algorithm Implementation and Development

3.1 Creation of the Videos

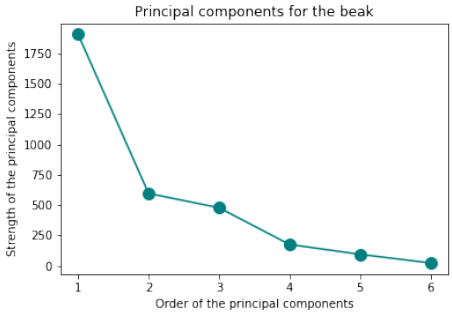
The first component of this study was to record the simulation of the folding of our origami model, along the x-, y-, and z-axes. To record this model, we used an open-source origami simulator created by several MIT students [5]. The recordings were then saved, and using an online video tool to shorten and resize each recording, these videos were imported into Python.

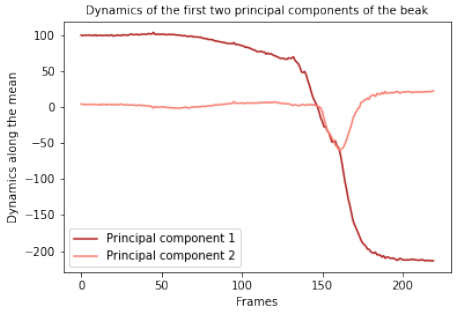
3.2 Tracking the Corners

When declaring the library of trackers that were available to us, the MIL (Multiple Instance Learning) tracker was the most pertinent to our study, as one of its cons is actually beneficial to the tracking of the model. Indeed, as the origami simulator folds the crane up from the crease pattern rather than following the folding instructions a model normally would, the simulated paper does not maintain the physical properties of a sheet of paper [5]. Rather, it blends into itself in some instances, and near the end of the model simulation, the origami crane abruptly *jumps* to action, leading to a very fast-paced change of state. Thus, as one of the disadvantages of the MIL tracker is that it has to slow down the video footage to track the motion, it becomes beneficial to use due to the footage of the simulation having an abrupt change at the end that causes the other trackers to lose sight of the corners that we are tracking [7].

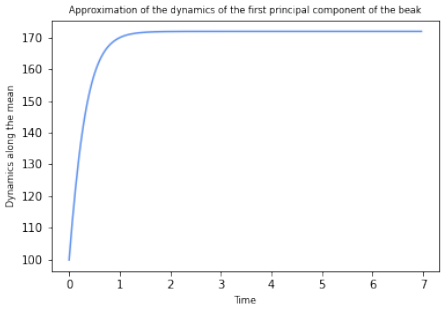
The MIL tracking was initiated through the calling forward of the very first frame, and a rectangle could be selected around each of the corners of the paper. The tracking was completed systematically in this order: the beak, the right wing, the left wing, and the tail, each having three different instances of tracking for each of the axes. Then, the program would track the given corner in every frame and transmit the coordinates of the rectangle, thus effectively creating a matrix of coordinates. However, due to the whole model moving and not simply one of the corners, it was observed that the tracking would miss its target if the rectangle was drawn too large. This most likely introduced some noise into the data. Once the tracking is complete, all of the x- and y-coordinates were appended into a single matrix for each corner, and only the fewest number of coordinates were retained (as each video slightly differed in frame count).

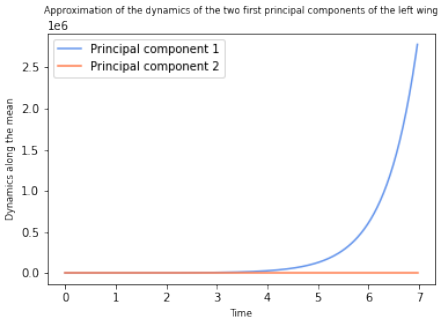
Due to the complexity of the motion of the origami crane as it was folded, the beak data observed in figure 2 clearly demonstrates the difference in the motion depending on the axis in the x-direction. Meanwhile, figure 3 depicts the similarity in the motion of the beak in all three axes in the y-direction. Nonetheless, as the difference in the motion and not the direction of the motion determines the result of the PCA, this should not have any incidence.





Figures 4 and 5: Principal components of the beak (left) and PCA dynamics of the beak (right).





Figures 6 and 7: SINDy models of the dynamics of the beak (left) and left wing (right).

3.3 Running PCA and SVD

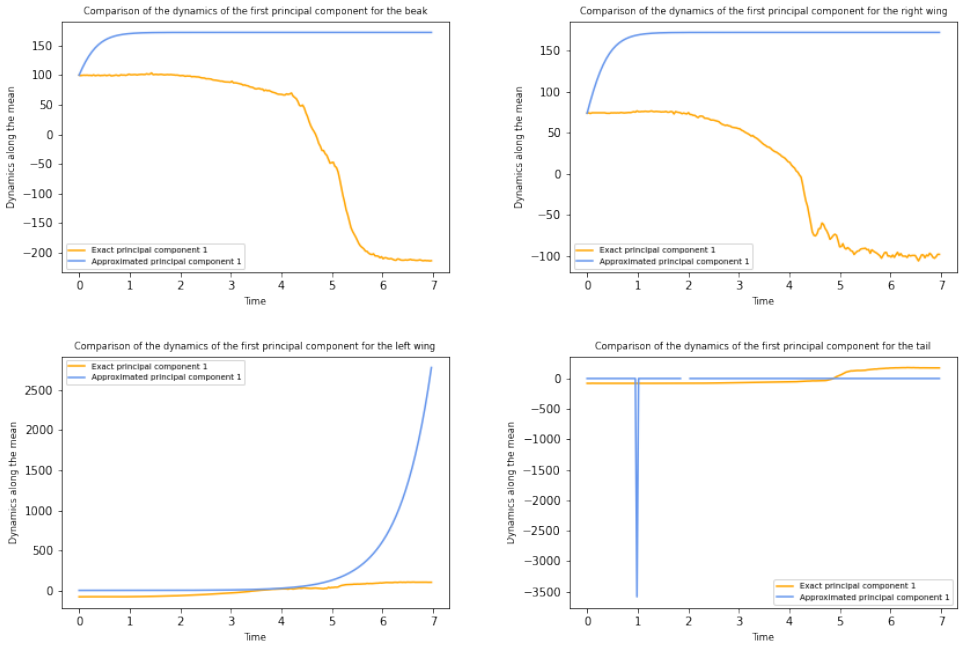
Prior to running the PCA, the mean was taken of every row of the x- and y-coordinates matrix, in order to subtract the mean from the coordinates. This allows for the PCA to be along the x-axis rather than in a much larger range. Then, the preexisting SVD and covariance functions from Numpy were run (3).

Figure 4 demonstrates the principal components of the beak, which has a much larger first principal component than the remaining five. This thus indicates that it holds much more information to be applied to the future motion model than the remaining principal components. In figure 5, the dynamics of the first two principal components of the beak are compared and demonstrate quite a variance in trajectory. Indeed, the first principal component is much more similar in its shape to the y-direction motion tracker coordinates trajectories than the latter principal component. Meanwhile, the second principal component resembles a normalized curve over the x-direction of the motion tracker coordinates. However, the importance of the first principal component may thus be observed in its resemblance to the existing actual trajectories the motion tracker followed.

3.4 Implementing SINDy

$$\begin{aligned} \frac{dx}{dt} &= 203.544 + 1.569x - 0.016x^2 \\ \frac{dx}{dt} &= 203.544 + 1.569x - 0.016y \end{aligned} \tag{7}$$

The SINDy algorithm was finally integrated, having but one principal component guiding it for the beak and a single differential equation to describe the complex motion of the origami crane folding (7). It was implemented using a coefficient from the PCA x- and y-coordinates matrix, in order to set an initial con-



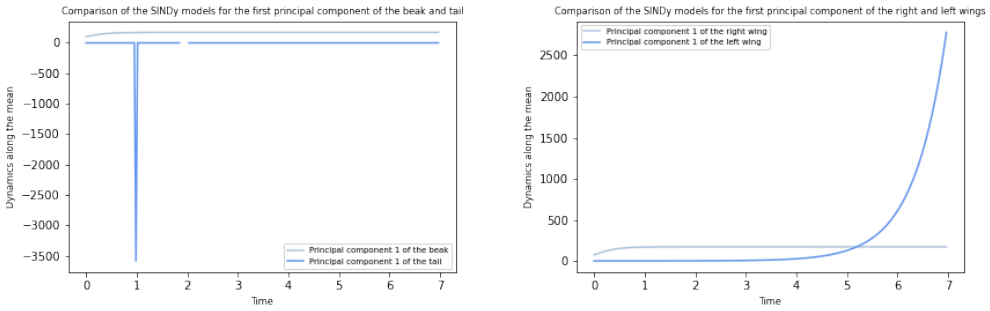
Figures 8, 9, 10, and 11: Comparison of the PCA and SINDy models for the motion of the beak, right wing, left wing, and tail, respectively.

dition. The sparsity parameter, set to 0.01, provided a threshold to limit the number of elements found in the model. The threshold was set to such a low value due to the behaviour of the model; had the threshold been set but 0.01 points higher, it would mean we would have a linear model approximating the complex dynamics of the origami crane. Thus, in order to feed the model more complexity, the added second-degree polynomial term was essential (7), thus providing the curve in figure 6. Furthermore, the use of a time interval of $\frac{7}{220}$ was selected for the implementation of the SINDy algorithm, as the videos were seven seconds long, and although a total of 225 frames were extracted, there was a steep drop in the data in the last few frames that were removed in order to normalize the data.

Although the left wing dynamics have not been discussed throughout this paper (all of its figures may be found in Appendix A), it is the only model for which two principal components were implemented. Indeed, for figure 21 in Appendix A, it may be observed that the second principal component could potentially hold valuable information. Thus, it was included in the SINDy model, but resulted in the same differential equation as the beak (7), with a minor difference, being that the final polynomial term belonged to the second principal component rather than the second degree of the first, thus, giving rise to the increasing curve of figure 7. Although a second differential equation appeared, it was equal to zero. Surprisingly, all of the coefficients were the same in all four SINDy models, for the beak, right and left wings, and the tail. This may have been due to the way with which the dynamics were not predictable, and thus the first principal component provided a model that would approximate all of the chaotic data. Nonetheless, when the data was run with four principal components, the coefficients were the same for all four SINDy models as well, thus leading to questions of whether the algorithm was properly implemented in the first place.

4 Computational Results

Overall, the models generated from our data were not as expected; indeed, they were found to be quite sub-optimal to the true dynamics of the data. Perhaps the use of more principal components could have counterbalanced this outcome, but we doubt this to be true. Doing so would mean generalizing between the different dynamics represented by the principal components, which differed greatly and would have led to a



Figures 12 and 13: Comparison of the SINDy models for the symmetrical motion of the beak and tail (left) and right and left wings (right).

much too generalized model between all the different dynamics.

Nevertheless, the models may still be analyzed for what they do show rather than what they lack. The first step to acquiring comparable data was the scaling of the data. The beak and right wing had comparable scales for both the PCA and SINDy dynamics, and thus no scaling was needed for these data sets, as observed in figures 8 and 9, respectively. However, for the curves related to the left wing and tail, the SINDy models were scaled (by 10^3 and 10^9, respectively) in the comparisons due to the discrepancy between the axis values of this model and the PCA model. This was done to ensure that the models with smaller scales would not be observed as a constant linear equation at zero, as we determined when the figures were first created. Unfortunately, this also means that the initial conditions no longer line up in figures 10 and 11. Furthermore, the large dip observed in both the left wing and tail models are problematic, outlier data points. Unfortunately, removing this data proved more difficult than expected, and thus was kept within the model. It was observed, however, when removing the initial first peak by moving forward in time approximately 50 frames, that smaller peaks of similar behaviour appeared. Thus, no matter whether the initial peaks were present, the model was not a good approximator for the left wing and tail of the origami crane. Although, a future direction would be to normalize these data points, and first locate their origin, be it in the original tracking of the data, or running the means for the PCA. Meanwhile, although the beak and left wing models were not comparatively accurate approximators, the stabilizing of the data along the second and third seconds of the origami folding may demonstrate the models attempting to approximate the consistent motion.

The individual models may have failed due to the potentially high level of noise from the tracking of the origami models, considering that the tracking boxes could not only encompass the very tip of the corner but had to include some of the paper as well. Due to the ill-defined corners and the ease with which they could be lost as the simulator folded the origami crane, a future direction would be to perhaps colour the corners differently before entering the crease pattern within the simulator, perhaps off-shooting the loss of the corner by the tracker and perhaps allowing the tracking box to be made smaller to incorporate as little of the simulated paper as possible.

Meanwhile, the final portion of this study is to analyze the symmetry of the origami crane in its folding and whether the SINDy model could replicate that. Indeed, as the simulator folds the model from its crease pattern, the motion is very symmetrical from start to finish, in that the beak and tail move together, and the right and left wings move together as well. Although the models were unable to approximate the actual motion of the origami folds, the comparison between the beak and tail, and the right and left wings demonstrate similarity in their motion, in their consistency over time. Although their scales corrected as previously described, perhaps then despite the outliers for the tail model, overall these models share similarities that would be comparable for symmetrical motion. However, it had been predicted that the beak would experience a greater fluctuation at the end of its motion due to the final fold, making that the only difference from the tail fold, which is not observed here. Furthermore, the right and left wing motions diverge near the five-second mark, whereas their motion should have been nearly identical. Thus, once more, noise in the

data may be the culprit for the generated models.

Perhaps the main outcome that may be drawn from this study is that the SINDy model may not be resilient enough to noise and such drastically different dynamics to propose an accurate model for the dynamics of the data. Indeed, researchers such as Yasuda and their colleagues observe the chaotic nature of origami motion through the use of quasi-recurrent neural networks (QRNNs) in order to predict folding dynamics, which demonstrate a much more efficient model for describing such motion in origami [8]. Perhaps then a future direction would certainly be to run a similar study using the methods cited in Yasuda et al.'s research.

There are several future directions this study may take. As mentioned above, an interesting one would be to run the same data on a neural network (NN), and according to previous research, perhaps a QRNN. Furthermore, implementing a neural network to learn crease pattern folding would naturally allow for the following step, which would be to develop a neural network using training data from as many existing crease patterns as possible, such that it would expose the NN to the many different types of folds permeating the world of origami. Then, it would be compelling to have the NN predict the next fold of an origami model that has not been defined yet, or perhaps understand the model based solely on the crease pattern. Consequently, another possible future direction would be to attempt to track vertices of the model which do disappear and use a tracker more robust to object disappearance. Then, more complex models could be studied using the outlined procedure of our study. A final future direction, although perhaps more difficult to achieve, would be to track a video of origami folding by instructions and not crease patterns, which would result in object disappearance more frequently than in the crease patterns.

5 Summary and Conclusions

To conclude, the goal of the study was to model the motion of the four corners of a simulated crease pattern folding itself into the origami crane, using the PCA and SINDy algorithm, in order to learn the equations governing the dynamics of this data in space and time. A single principal component was used in three of the four models (the differing model using two principal components) in order to represent the positional status of the corners at a particular time in space. Due to the nature of the data and the noise created by the tracking of the corners, and the use of SINDy perhaps further amplifying the noisiness of the data, the algorithm was unable to properly approximate the actual motion of the folded crane. As mentioned above, perhaps a different approach altogether shall be necessary to properly map the trajectories of origami folds in space and time.

6 Github Repositories

Myriam: `https://github.com/OPUS144/Dynamical_Systems.git`

Alex: `https://github.com/Vyniril/Dynamical-Systems.git`

References

[1] Jim Propp (MIT) and sci.math.research. *The Margulis Napkin Problem*. 1996.

[2] Jason Bramburger. *Data-Driven Methods for Dynamical Systems: A new perspective on old problems*. Concordia University, 2023.

[3] Encyclopædia Britannica. *History of Origami,* *https://www.britannica.com/art/origami/History-of-origami*.

[4] Erik D. Demaine, Sandor P. Fekete, and Robert J. Lang. *Circle Packing for Origami Design Is Hard*.

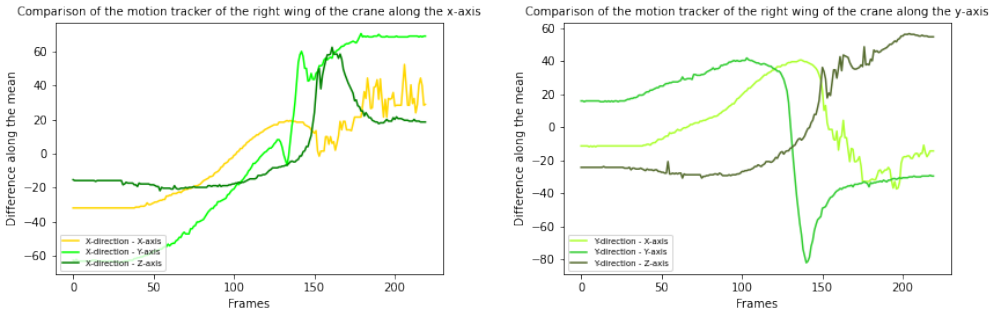
[5] Amanda Ghassaei, Erik Demaine, and Neil Gershenfeld. *Origami Simulator,* *https://origamisimulator.org/*.

[6] Ian T. Jollife and Jorge Cadima. *Principal component analysis: A review and recent developments*. 2016.

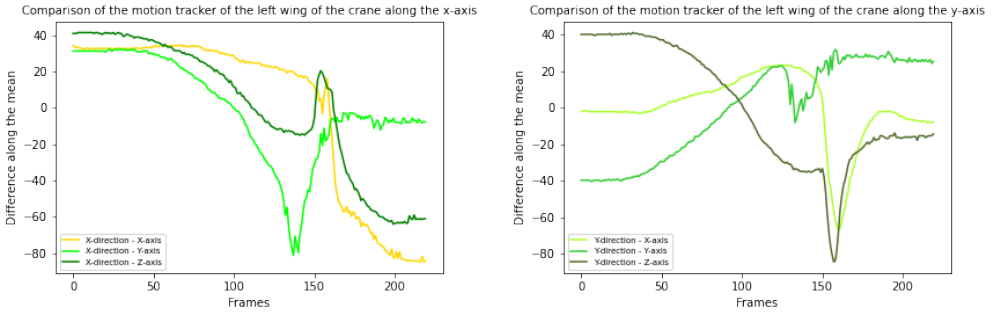
[7] Brouton Lab. *A Complete Review of the OpenCV Object Tracking Algorithms,* `https://broutonlab.com/blog/opencv-object-tracking`.

[8] Hiromi Yasuda et al. *Data-driven prediction and analysis of chaotic origami dynamics*. 2020.

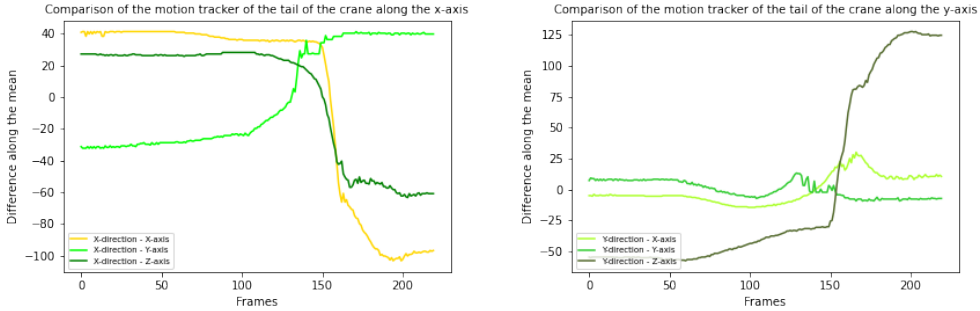
Appendix A Additional Graph



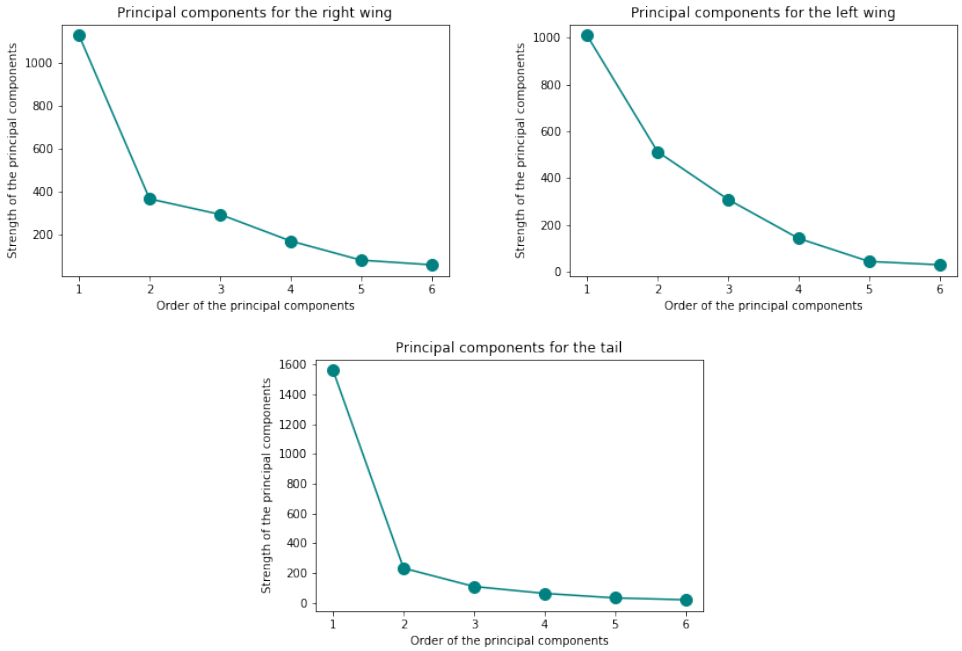
Figures 14 and 15: Tracking the motion of the right wing along the x-, y-, z-axes for the x- and y-directions of the coordinates.



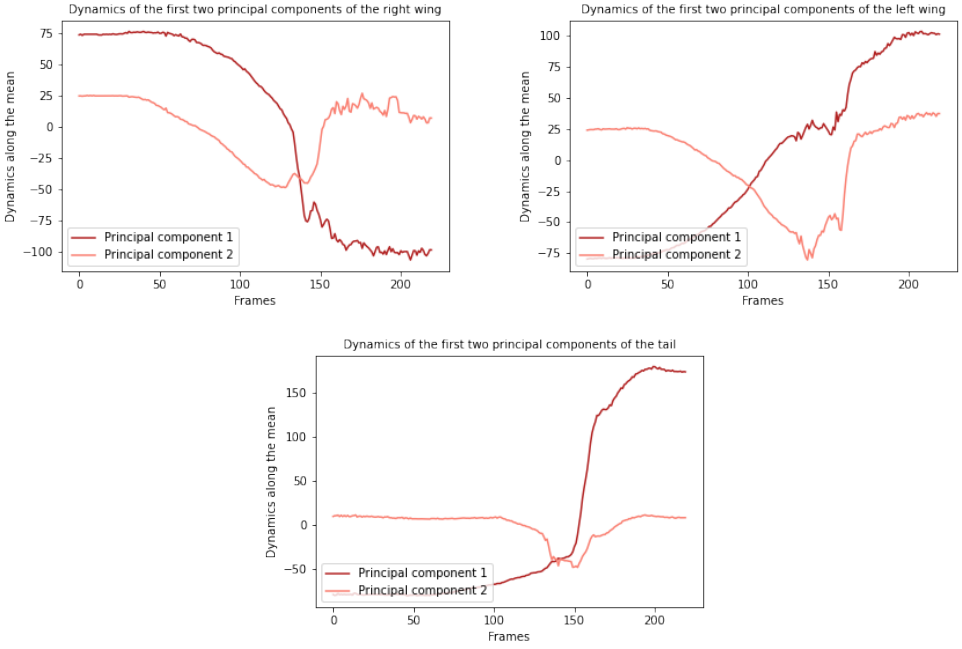
Figures 16 and 17: Tracking the motion of the left wing along the x-, y-, z-axes for the x- and y-directions of the coordinates.



Figures 18 and 19: Tracking the motion of the tail along the x-, y-, z-axes for the x- and y-directions of the coordinates.



Figures 20, 21, and 22: Principal components of the right wing (top left), left wing (top right) and the tail (bottom).



Figures 23, 24, and 25: PCA dynamics of the right wing (top left), left wing (top right) and the tail (bottom).

Appendix B Python Premade Functions

- `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 C Python Code

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

```
##PROCEDURE:
# 1. Take three screen recordings of different angles of the paper (x, y, and z) being folded from 0% to 100%.
#     - Origami simulator: https://origamisimulator.org/.
# 2. Import videos and use motion tracker on corners - try different tracker resistant to disappearance of
↪  object.
#     - Run tracker 12 (3x4) times as tracking corners separately (wings tips, beak, and tail).
# 3. Create 4 different xy-matrices (for each axis) - should have 6 rows each.
#     - We want to compare motion according to the corner!
#     - Otherwise, PCA shall be comparing different corners along the same axis.
# 4. Run PCA and SINDy - each corner SINDy should have either 2 or 4 elements (consider physics behind that).
# 5. Run a comparison between SINDY models and see how similar the motion is for each corner.
#     - PCA and SINDy models: Compare the two motion models for the wing tips and the two motion models for the
↪  beak and tail in order to see how well they overlap.
```

```
#     - Overlap the PCA and SINDy model graphs and perhaps run an error calculation or simply visualize the
↪  difference between the models.
# 6. Future directions: motion tracker for four corners at the same time.
#     - A single motion tracker would lead to a very complex SINDy model.
#     - Disappearance of the corners means the tracker may get confused.

#   o Basically, an x, y, and z point of view, making sure to get the crane's head at the front corner for the
↪  x and y angles.
#   o The colour is 66CC00 for both sides of the paper, as it kind of lags with two separate colours.
#   o We are using the crane as it is the oldest model of origami.
#   o The corners end up at the tips of the model due to the symmetry of the model.
#     - The center of the paper ends up at the middle point of the back once complete.

! 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

## IMPORTING VIDEOS AS ARRAYS:

cranex = cv.VideoCapture("CraneX.mp4")

# Defining the columns of the empty matrix:

framecount = int(cranex.get(cv.CAP_PROP_FRAME_COUNT))
width = int(cranex.get(cv.CAP_PROP_FRAME_WIDTH))
height = int(cranex.get(cv.CAP_PROP_FRAME_HEIGHT))

# Creating a buffer in order to be able to insert the information from the video into it:

bufferx = np.empty((framecount, height, width, 3), np.dtype('uint8'))
# 8-bit unsigned integer, desired output (due to the RGB values).
# The buffer stores information temporarily.

fc = 0
ret = True

while (fc < framecount and ret):
    ret, bufferx[fc] = cranex.read()
    fc += 1

cranex.release()

print(bufferx.shape) # (238 [frames], 1280 [width], 1280 [height], 3 [RGB])
print(bufferx)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪  # # # #

craney = cv.VideoCapture("CraneY.mp4")

# Defining the columns of the empty matrix:

framecount = int(craney.get(cv.CAP_PROP_FRAME_COUNT))
width = int(craney.get(cv.CAP_PROP_FRAME_WIDTH))
height = int(craney.get(cv.CAP_PROP_FRAME_HEIGHT))

# Creating a buffer in order to be able to insert the information from the video into it:

buffery = np.empty((framecount, height, width, 3), np.dtype('uint8'))
# 8-bit unsigned integer, desired output (due to the RGB values).
# The buffer stores information temporarily.

fc = 0
ret = True
```

```
while (fc < framecount and ret):
    ret, buffery[fc] = craney.read()
    fc += 1

craney.release()

print(buffery.shape) # (227 [frames], 1280 [width], 1280 [height], 3 [RGB])
print(buffery)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪  # # # #

cranez = cv.VideoCapture("CraneZ.mp4")

# Defining the columns of the empty matrix:

framecount = int(cranez.get(cv.CAP_PROP_FRAME_COUNT))
width = int(cranez.get(cv.CAP_PROP_FRAME_WIDTH))
height = int(cranez.get(cv.CAP_PROP_FRAME_HEIGHT))

# Creating a buffer in order to be able to insert the information from the video into it:

bufferz = np.empty((framecount, height, width, 3), np.dtype('uint8'))
# 8-bit unsigned integer, desired output (due to the RGB values).
# The buffer stores information temporarily.

fc = 0
ret = True

while (fc < framecount and ret):
    ret, bufferz[fc] = cranez.read()
    fc += 1

cranez.release()

print(bufferz.shape) # (225 [frames], 1280 [width], 1280 [height], 3 [RGB]).
print(bufferz)

## 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()

# Instead of guessing where the tracked object is in the next frame, an approach is used in which several
↪  potentially positive objects are selected around a positive definite object.
#    o Pros: more robust to noise, shows fairly good accuracy.
#     - There is a point in the video where everything melds together, this tracker is robust enough to handle
↪  it.
#    o Cons: relatively low speed and the impossibility of stopping tracking when the object is lost.
#     - The tracker being slow is a good thing! Our video is too fast in some places for the other trackers to
↪  follow.
```

```
## SAVING THE COORDINATES OF THE TRACKING RECTANGLE:
# Creating a matrix of zeros to then enter the coordinates of the box.

mat_beak = np.zeros((0,225))
mat_rw = np.zeros((0,225))
mat_lw = np.zeros((0,225))
mat_tail = np.zeros((0,225))
# The minimal number of frames is 225 from the z-axis crane. Thus, the matrices must account for this; the
↪  tracker shall only generate 224 data points.

## OBJECT TRACKING WITH OPENCV: X-BEAK
# This code allows us to select a box around the object:

# Get the video file and read it:
CX_beak = cv.VideoCapture("CraneX.mp4")
ret, frame = CX_beak.read()

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

## STORING THE COORDINATES: X-BEAK

X1_beak = np.zeros((238,1))
Y1_beak = np.zeros((238,1))

i = 0

while True:
    ret,frame = CX_beak.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 (RGB),
        ↪  thickness.
        XC1_beak = (x+w)/2
        YC1_beak = (y+h)/2
        print(XC1_beak, YC1_beak)
        X1_beak[i] = XC1_beak
        Y1_beak[i] = YC1_beak
    cv.imshow("Frame", frame) # We need a positional argument, matrix form.
    key = cv.waitKey(30)
    i += 1
    if key == ord("q"):
        break

CX_beak.release()
cv.destroyAllWindows()

## OBJECT TRACKING WITH OPENCV: X-RIGHT WING
# This code allows us to select a box around the object:

# Get the video file and read it:
CX_rw = cv.VideoCapture("CraneX.mp4")
ret, frame = CX_rw.read()

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

## STORING THE COORDINATES: X-RIGHT WING

X1_rw = np.zeros((238,1))
Y1_rw = np.zeros((238,1))

i = 0
```

```
while True:
    ret,frame = CX_rw.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 (RGB),
        ↪  thickness.
        XC1_rw = (x+w)/2
        YC1_rw = (y+h)/2
        print(XC1_rw, YC1_rw)
        X1_rw[i] = XC1_rw
        Y1_rw[i] = YC1_rw
    cv.imshow("Frame", frame) # We need a positional argument, matrix form.
    key = cv.waitKey(30)
    i += 1
    if key == ord("q"):
        break

CX_rw.release()
cv.destroyAllWindows()

## OBJECT TRACKING WITH OPENCV: X-LEFT WING
# This code allows us to select a box around the object:

# Get the video file and read it:
CX_lw = cv.VideoCapture("CraneX.mp4")
ret, frame = CX_lw.read()

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

## STORING THE COORDINATES: X-LEFT WING

X1_lw = np.zeros((238,1))
Y1_lw = np.zeros((238,1))

i = 0

while True:
    ret,frame = CX_lw.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 (RGB),
        ↪  thickness.
        XC1_lw = (x+w)/2
        YC1_lw = (y+h)/2
        print(XC1_lw, YC1_lw)
        X1_lw[i] = XC1_lw
        Y1_lw[i] = YC1_lw
    cv.imshow("Frame", frame) # We need a positional argument, matrix form.
    key = cv.waitKey(30)
    i += 1
    if key == ord("q"):
        break

CX_lw.release()
cv.destroyAllWindows()

## OBJECT TRACKING WITH OPENCV: X-TAIL
# This code allows us to select a box around the object:
```

```
# Get the video file and read it:
CX_tail = cv.VideoCapture("CraneX.mp4")
ret, frame = CX_tail.read()

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

## STORING THE COORDINATES: X-TAIL

X1_tail = np.zeros((238,1))
Y1_tail = np.zeros((238,1))

i = 0

while True:
    ret,frame = CX_tail.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 (RGB),
        ↪  thickness.
        XC1_tail = (x+w)/2
        YC1_tail = (y+h)/2
        print(XC1_tail, YC1_tail)
        X1_tail[i] = XC1_tail
        Y1_tail[i] = YC1_tail
    cv.imshow("Frame", frame) # We need a positional argument, matrix form.
    key = cv.waitKey(30)
    i += 1
    if key == ord("q"):
        break

CX_tail.release()
cv.destroyAllWindows()

## OBJECT TRACKING WITH OPENCV: Y-BEAK
# This code allows us to select a box around the object:

# Get the video file and read it:
CY_beak = cv.VideoCapture("CraneY.mp4")
ret, frame = CY_beak.read()

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

## STORING THE COORDINATES: Y-BEAK

X2_beak = np.zeros((227,1))
Y2_beak = np.zeros((227,1))

i = 0

while True:
    ret,frame = CY_beak.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 (RGB),
        ↪  thickness.
        XC2_beak = (x+w)/2
        YC2_beak = (y+h)/2
        print(XC2_beak, YC2_beak)
```

```
        X2_beak[i] = XC2_beak
        Y2_beak[i] = YC2_beak
    cv.imshow("Frame", frame) # We need a positional argument, matrix form.
    key = cv.waitKey(30)
    i += 1
    if key == ord("q"):
        break

CY_beak.release()
cv.destroyAllWindows()

## OBJECT TRACKING WITH OPENCV: Y-RIGHT WING
# This code allows us to select a box around the object:

# Get the video file and read it:
CY_rw = cv.VideoCapture("CraneY.mp4")
ret, frame = CY_rw.read()

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

##STORING THE COORDINATES: Y-RIGHT WING

X2_rw = np.zeros((227,1))
Y2_rw = np.zeros((227,1))

i = 0

while True:
    ret,frame = CY_rw.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 (RGB),
        ↪  thickness.
        XC2_rw = (x+w)/2
        YC2_rw = (y+h)/2
        print(XC2_rw, YC2_rw)
        X2_rw[i] = XC2_rw
        Y2_rw[i] = YC2_rw
    cv.imshow("Frame", frame) # We need a positional argument, matrix form.
    key = cv.waitKey(30)
    i += 1
    if key == ord("q"):
        break

CY_rw.release()
cv.destroyAllWindows()

## OBJECT TRACKING WITH OPENCV: Y-LEFT WING
# This code allows us to select a box around the object:

# Get the video file and read it:
CY_lw = cv.VideoCapture("CraneY.mp4")
ret, frame = CY_lw.read()

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

##STORING THE COORDINATES: Y-LEFT WING

X2_lw = np.zeros((227,1))
Y2_lw = np.zeros((227,1))
```

```
i = 0

while True:
    ret,frame = CY_lw.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 (RGB),
        ↪  thickness.
        XC2_lw = (x+w)/2
        YC2_lw = (y+h)/2
        print(XC2_lw, YC2_lw)
        X2_lw[i] = XC2_lw
        Y2_lw[i] = YC2_lw
    cv.imshow("Frame", frame) # We need a positional argument, matrix form.
    key = cv.waitKey(30)
    i += 1
    if key == ord("q"):
        break

CY_lw.release()
cv.destroyAllWindows()

## OBJECT TRACKING WITH OPENCV: Y-TAIL
# This code allows us to select a box around the object:

# Get the video file and read it:
CY_tail = cv.VideoCapture("CraneY.mp4")
ret, frame = CY_tail.read()

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

##STORING THE COORDINATES: Y-TAIL

X2_tail = np.zeros((227,1))
Y2_tail = np.zeros((227,1))

i = 0

while True:
    ret,frame = CY_tail.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 (RGB),
        ↪  thickness.
        XC2_tail = (x+w)/2
        YC2_tail = (y+h)/2
        print(XC2_tail, YC2_tail)
        X2_tail[i] = XC2_tail
        Y2_tail[i] = YC2_tail
    cv.imshow("Frame", frame) # We need a positional argument, matrix form.
    key = cv.waitKey(30)
    i += 1
    if key == ord("q"):
        break

CY_tail.release()
cv.destroyAllWindows()

##OBJECT TRACKING WITH OPENCV: Z-BEAK
# This code allows us to select a box around the object:
```

```
# Get the video file and read it:
CZ_beak = cv.VideoCapture("CraneZ.mp4")
ret, frame = CZ_beak.read()

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

##STORING THE COORDINATES: Z-BEAK

X3_beak = np.zeros((225,1))
Y3_beak = np.zeros((225,1))

i = 0

while True:
    ret,frame = CZ_beak.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 (RGB),
        ↪  thickness.
        XC3_beak = (x+w)/2
        YC3_beak = (y+h)/2
        print(XC3_beak, YC3_beak)
        X3_beak[i] = XC3_beak
        Y3_beak[i] = YC3_beak
    cv.imshow("Frame", frame) # We need a positional argument, matrix form.
    key = cv.waitKey(30)
    i += 1
    if key == ord("q"):
        break

CZ_beak.release()
cv.destroyAllWindows()

##OBJECT TRACKING WITH OPENCV: Z-RIGHT WING
# This code allows us to select a box around the object:

# Get the video file and read it:
CZ_rw = cv.VideoCapture("CraneZ.mp4")
ret, frame = CZ_rw.read()

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

##STORING THE COORDINATES: Z-RIGHT WING

X3_rw = np.zeros((225,1))
Y3_rw = np.zeros((225,1))

i = 0

while True:
    ret,frame = CZ_rw.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 (RGB),
        ↪  thickness.
        XC3_rw = (x+w)/2
        YC3_rw = (y+h)/2
```

```
        print(XC3_rw, YC3_rw)
        X3_rw[i] = XC3_rw
        Y3_rw[i] = YC3_rw
    cv.imshow("Frame", frame) # We need a positional argument, matrix form.
    key = cv.waitKey(30)
    i += 1
    if key == ord("q"):
        break

CZ_rw.release()
cv.destroyAllWindows()

##OBJECT TRACKING WITH OPENCV: Z-LEFT WING
# This code allows us to select a box around the object:

# Get the video file and read it:
CZ_lw = cv.VideoCapture("CraneZ.mp4")
ret, frame = CZ_lw.read()

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

##STORING THE COORDINATES: Z-LEFT WING

X3_lw = np.zeros((225,1))
Y3_lw = np.zeros((225,1))

i = 0

while True:
    ret,frame = CZ_lw.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 (RGB),
        ↪  thickness.
        XC3_lw = (x+w)/2
        YC3_lw = (y+h)/2
        print(XC3_lw, YC3_lw)
        X3_lw[i] = XC3_lw
        Y3_lw[i] = YC3_lw
    cv.imshow("Frame", frame) # We need a positional argument, matrix form.
    key = cv.waitKey(30)
    i += 1
    if key == ord("q"):
        break

CZ_lw.release()
cv.destroyAllWindows()

##OBJECT TRACKING WITH OPENCV: Z-TAIL
# This code allows us to select a box around the object:

# Get the video file and read it:
CZ_tail = cv.VideoCapture("CraneZ.mp4")
ret, frame = CZ_tail.read()

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

##STORING THE COORDINATES: Z-TAIL

X3_tail = np.zeros((225,1))
Y3_tail = np.zeros((225,1))
```

```
i = 0

while True:
    ret,frame = CZ_tail.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 (RGB),
        ↪  thickness.
        XC3_tail = (x+w)/2
        YC3_tail = (y+h)/2
        print(XC3_tail, YC3_tail)
        X3_tail[i] = XC3_tail
        Y3_tail[i] = YC3_tail
    cv.imshow("Frame", frame) # We need a positional argument, matrix form.
    key = cv.waitKey(30)
    i += 1
    if key == ord("q"):
        break

CZ_tail.release()
cv.destroyAllWindows()

print(mat_beak.shape)
print(mat_rw.shape)
print(mat_lw.shape)
print(mat_tail.shape)

X1_beak = np.resize(X1_beak, (225,1))
Y1_beak = np.resize(Y1_beak, (225,1))

X1_rw = np.resize(X1_rw, (225,1))
Y1_rw = np.resize(Y1_rw, (225,1))

X1_lw = np.resize(X1_lw, (225,1))
Y1_lw = np.resize(Y1_lw, (225,1))

X1_tail = np.resize(X1_tail, (225,1))
Y1_tail = np.resize(Y1_tail, (225,1))

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪  # # # #

X2_beak = np.resize(X2_beak, (225,1))
Y2_beak = np.resize(Y2_beak, (225,1))

X2_rw = np.resize(X2_rw, (225,1))
Y2_rw = np.resize(Y2_rw, (225,1))

X2_lw = np.resize(X2_lw, (225,1))
Y2_lw = np.resize(Y2_lw, (225,1))

X2_tail = np.resize(X2_tail, (225,1))
Y2_tail = np.resize(Y2_tail, (225,1))

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪  # # # #

X3_beak = np.resize(X3_beak, (225,1))
Y3beak = np.resize(Y3_beak, (225,1))

X3_rw = np.resize(X3_rw, (225,1))
Y3_rw = np.resize(Y3_rw, (225,1))

X3_lw = np.resize(X3_lw, (225,1))
```

```
Y3_lw = np.resize(Y3_lw, (225,1))

X3_tail = np.resize(X3_tail, (225,1))
Y3_tail = np.resize(Y3_tail, (225,1))

# Resizing due to the size of the xy-matrix.

X1_beak = np.reshape(X1_beak, (1,225))
Y1_beak = np.reshape(Y1_beak, (1,225))

X1_rw = np.reshape(X1_rw, (1,225))
Y1_rw = np.reshape(Y1_rw, (1,225))

X1_lw = np.reshape(X1_lw, (1,225))
Y1_lw = np.reshape(Y1_lw, (1,225))

X1_tail = np.reshape(X1_tail, (1,225))
Y1_tail = np.reshape(Y1_tail, (1,225))

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪ # # # #

X2_beak = np.reshape(X2_beak, (1,225))
Y2_beak = np.reshape(Y2_beak, (1,225))

X2_rw = np.reshape(X2_rw, (1,225))
Y2_rw = np.reshape(Y2_rw, (1,225))

X2_lw = np.reshape(X2_lw, (1,225))
Y2_lw = np.reshape(Y2_lw, (1,225))

X2_tail = np.reshape(X2_tail, (1,225))
Y2_tail = np.reshape(Y2_tail, (1,225))

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪ # # # #

X3_beak = np.reshape(X3_beak, (1,225))
Y3_beak = np.reshape(Y3_beak, (1,225))

X3_rw = np.reshape(X3_rw, (1,225))
Y3_rw = np.reshape(Y3_rw, (1,225))

X3_lw = np.reshape(X3_lw, (1,225))
Y3_lw = np.reshape(Y3_lw, (1,225))

X3_tail = np.reshape(X3_tail, (1,225))
Y3_tail = np.reshape(Y3_tail, (1,225))

# To match the matrix format (changing row vectors).

mat_beak = np.append(mat_beak, X1_beak, axis = 0)
mat_beak = np.append(mat_beak, Y1_beak, axis = 0)
mat_beak = np.append(mat_beak, X2_beak, axis = 0)
mat_beak = np.append(mat_beak, Y2_beak, axis = 0)
mat_beak = np.append(mat_beak, X3_beak, axis = 0)
mat_beak = np.append(mat_beak, Y3_beak, axis = 0)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪ # # # #

mat_rw = np.append(mat_rw, X1_rw, axis = 0)
mat_rw = np.append(mat_rw, Y1_rw, axis = 0)
mat_rw = np.append(mat_rw, X2_rw, axis = 0)
mat_rw = np.append(mat_rw, Y2_rw, axis = 0)
mat_rw = np.append(mat_rw, X3_rw, axis = 0)
mat_rw = np.append(mat_rw, Y3_rw, axis = 0)
```

```
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪  # # # #

mat_lw = np.append(mat_lw, X1_lw, axis = 0)
mat_lw = np.append(mat_lw, Y1_lw, axis = 0)
mat_lw = np.append(mat_lw, X2_lw, axis = 0)
mat_lw = np.append(mat_lw, Y2_lw, axis = 0)
mat_lw = np.append(mat_lw, X3_lw, axis = 0)
mat_lw = np.append(mat_lw, Y3_lw, axis = 0)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪  # # # #

mat_tail = np.append(mat_tail, X1_tail, axis = 0)
mat_tail = np.append(mat_tail, Y1_tail, axis = 0)
mat_tail = np.append(mat_tail, X2_tail, axis = 0)
mat_tail = np.append(mat_tail, Y2_tail, axis = 0)
mat_tail = np.append(mat_tail, X3_tail, axis = 0)
mat_tail = np.append(mat_tail, Y3_tail, axis = 0)

print(mat_beak.shape)
print(mat_beak)

print(mat_rw.shape)
print(mat_rw)

print(mat_lw.shape)
print(mat_lw)

print(mat_tail.shape)
print(mat_tail)

## 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:

row1Xb = mat_beak[0,:]
row1Yb = mat_beak[1,:]
row2Xb = mat_beak[2,:]
row2Yb = mat_beak[3,:]
row3Xb = mat_beak[4,:]
row3Yb = mat_beak[5,:]

row1Xr = mat_rw[0,:]
row1Yr = mat_rw[1,:]
row2Xr = mat_rw[2,:]
row2Yr = mat_rw[3,:]
row3Xr = mat_rw[4,:]
row3Yr = mat_rw[5,:]

row1Xl = mat_lw[0,:]
row1Yl = mat_lw[1,:]
row2Xl = mat_lw[2,:]
row2Yl = mat_lw[3,:]
row3Xl = mat_lw[4,:]
row3Yl = mat_lw[5,:]

row1Xt = mat_tail[0,:]
row1Yt = mat_tail[1,:]
row2Xt = mat_tail[2,:]
row2Yt = mat_tail[3,:]
row3Xt = mat_tail[4,:]
row3Yt = mat_tail[5,:]

# Computing the mean of every row separately:

meanX1b = np.mean(row1Xb)
```

```
meanY1b = np.mean(row1Yb)
meanX2b = np.mean(row2Xb)
meanY2b = np.mean(row2Yb)
meanX3b = np.mean(row3Xb)
meanY3b = np.mean(row3Yb)

meanX1r = np.mean(row1Xr)
meanY1r = np.mean(row1Yr)
meanX2r = np.mean(row2Xr)
meanY2r = np.mean(row2Yr)
meanX3r = np.mean(row3Xr)
meanY3r = np.mean(row3Yr)

meanX1l = np.mean(row1Xl)
meanY1l = np.mean(row1Yl)
meanX2l = np.mean(row2Xl)
meanY2l = np.mean(row2Yl)
meanX3l = np.mean(row3Xl)
meanY3l = np.mean(row3Yl)

meanX1t = np.mean(row1Xt)
meanY1t = np.mean(row1Yt)
meanX2t = np.mean(row2Xt)
meanY2t = np.mean(row2Yt)
meanX3t = np.mean(row3Xt)
meanY3t = np.mean(row3Yt)

# Computing the rows without their mean:

mat_beak[0,:] = mat_beak[0,:] - meanX1b
mat_beak[1,:] = mat_beak[1,:] - meanY1b
mat_beak[2,:] = mat_beak[2,:] - meanX2b
mat_beak[3,:] = mat_beak[3,:] - meanY2b
mat_beak[4,:] = mat_beak[4,:] - meanX3b
mat_beak[5,:] = mat_beak[5,:] - meanY3b

mat_rw[0,:] = mat_rw[0,:] - meanX1r
mat_rw[1,:] = mat_rw[1,:] - meanY1r
mat_rw[2,:] = mat_rw[2,:] - meanX2r
mat_rw[3,:] = mat_rw[3,:] - meanY2r
mat_rw[4,:] = mat_rw[4,:] - meanX3r
mat_rw[5,:] = mat_rw[5,:] - meanY3r

mat_lw[0,:] = mat_lw[0,:] - meanX1l
mat_lw[1,:] = mat_lw[1,:] - meanY1l
mat_lw[2,:] = mat_lw[2,:] - meanX2l
mat_lw[3,:] = mat_lw[3,:] - meanY2l
mat_lw[4,:] = mat_lw[4,:] - meanX3l
mat_lw[5,:] = mat_lw[5,:] - meanY3l

mat_tail[0,:] = mat_tail[0,:] - meanX1t
mat_tail[1,:] = mat_tail[1,:] - meanY1t
mat_tail[2,:] = mat_tail[2,:] - meanX2t
mat_tail[3,:] = mat_tail[3,:] - meanY2t
mat_tail[4,:] = mat_tail[4,:] - meanX3t
mat_tail[5,:] = mat_tail[5,:] - meanY3t

print(mat_beak)
print(mat_rw)
print(mat_lw)
print(mat_tail)

# The mean should be removed to run PCA as the results will look awful otherwise.

mat_beak = np.load("mat_beak.npy")
print(mat_beak.shape)

mat_rw = np.load("mat_rw.npy")
```

```python
print(mat_rw.shape)

mat_lw = np.load("mat_lw.npy")
print(mat_lw.shape)

mat_tail = np.load("mat_tail.npy")
print(mat_tail.shape)

lib.plot(mat_beak[0,0:220], c = "gold", label = "X-direction - X-axis")
lib.plot(mat_beak[2,0:220], c = "lime", label = "X-direction - Y-axis")
lib.plot(mat_beak[4,0:220], c = "green", label = "X-direction - Z-axis")
lib.xlabel("Frames")
lib.ylabel("Difference along the mean")
lib.title("Comparison of the motion tracker of the beak of the crane along the x-axis", fontsize = 10)
lib.legend(loc = "lower left", prop = {"size":7})
lib.savefig("beak_means_x.png")
lib.show()

lib.plot(mat_beak[1,0:220], c = "greenyellow", label = "Y-direction - X-axis")
lib.plot(mat_beak[3,0:220], c = "limegreen", label = "Y-direction - Y-axis")
lib.plot(mat_beak[5,0:220], c = "darkolivegreen", label = "Y-direction - Z-axis")
lib.xlabel("Frames")
lib.ylabel("Difference along the mean")
lib.title("Comparison of the motion tracker of the beak of the crane along the y-axis", fontsize = 10)
lib.legend(loc = "lower left", prop = {"size":7})
lib.savefig("beak_means_y.png")
lib.show()

"""

lib.plot(mat_beak[0,0:220], c = "gold", label = "X-direction - X-axis")
lib.plot(mat_beak[1,0:220], c = "greenyellow", label = "Y-direction - X-axis")
lib.plot(mat_beak[2,0:220], c = "lime", label = "X-direction - Y-axis")
lib.plot(mat_beak[3,0:220], c = "limegreen", label = "Y-direction - Y-axis")
lib.plot(mat_beak[4,0:220], c = "green", label = "X-direction - Z-axis")
lib.plot(mat_beak[5,0:220], c = "darkolivegreen", label = "Y-direction - Z-axis")
lib.xlabel("Frames")
lib.ylabel("Difference along the mean")
lib.title("Comparison of the motion tracker of the beak of the crane along all axes", fontsize = 10)
lib.legend(loc = "lower left", prop = {"size":7})
lib.savefig("beak_means.png")
lib.show()

"""

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪ # # # #

lib.plot(mat_rw[0,0:220], c = "gold", label = "X-direction - X-axis")
lib.plot(mat_rw[2,0:220], c = "lime", label = "X-direction - Y-axis")
lib.plot(mat_rw[4,0:220], c = "green", label = "X-direction - Z-axis")
lib.xlabel("Frames")
lib.ylabel("Difference along the mean")
lib.title("Comparison of the motion tracker of the right wing of the crane along the x-axis", fontsize = 10)
lib.legend(loc = "lower left", prop = {"size":7})
lib.savefig("rw_means_x.png")
lib.show()

lib.plot(mat_rw[1,0:220], c = "greenyellow", label = "Y-direction - X-axis")
lib.plot(mat_rw[3,0:220], c = "limegreen", label = "Y-direction - Y-axis")
lib.plot(mat_rw[5,0:220], c = "darkolivegreen", label = "Y-direction - Z-axis")
lib.xlabel("Frames")
lib.ylabel("Difference along the mean")
lib.title("Comparison of the motion tracker of the right wing of the crane along the y-axis", fontsize = 10)
lib.legend(loc = "lower left", prop = {"size":7})
lib.savefig("rw_means_y.png")
lib.show()
```

```python
"""

lib.plot(mat_rw[0,0:220], c = "gold", label = "X-direction - X-axis")
lib.plot(mat_rw[1,0:220], c = "greenyellow", label = "Y-direction - X-axis")
lib.plot(mat_rw[2,0:220], c = "lime", label = "X-direction - Y-axis")
lib.plot(mat_rw[3,0:220], c = "limegreen", label = "Y-direction - Y-axis")
lib.plot(mat_rw[4,0:220], c = "green", label = "X-direction - Z-axis")
lib.plot(mat_rw[5,0:220], c = "darkolivegreen", label = "Y-direction - Z-axis")
lib.xlabel("Frames")
lib.ylabel("Difference along the mean")
lib.title("Comparison of the motion tracker of the right wing of the crane along all axes", fontsize = 10)
lib.legend(loc = "lower left", prop = {"size":7})
lib.savefig("rw_means.png")
lib.show()

"""

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪ # # # #

lib.plot(mat_lw[0,0:220], c = "gold", label = "X-direction - X-axis")
lib.plot(mat_lw[2,0:220], c = "lime", label = "X-direction - Y-axis")
lib.plot(mat_lw[4,0:220], c = "green", label = "X-direction - Z-axis")
lib.xlabel("Frames")
lib.ylabel("Difference along the mean")
lib.title("Comparison of the motion tracker of the left wing of the crane along the x-axis", fontsize = 10)
lib.legend(loc = "lower left", prop = {"size":7})
lib.savefig("lw_means_x.png")
lib.show()

lib.plot(mat_lw[1,0:220], c = "greenyellow", label = "Y-direction - X-axis")
lib.plot(mat_lw[3,0:220], c = "limegreen", label = "Y-direction - Y-axis")
lib.plot(mat_lw[5,0:220], c = "darkolivegreen", label = "Y-direction - Z-axis")
lib.xlabel("Frames")
lib.ylabel("Difference along the mean")
lib.title("Comparison of the motion tracker of the left wing of the crane along the y-axis", fontsize = 10)
lib.legend(loc = "lower left", prop = {"size":7})
lib.savefig("lw_means_y.png")
lib.show()

"""

lib.plot(mat_lw[0,0:220], c = "gold", label = "X-direction - X-axis")
lib.plot(mat_lw[1,0:220], c = "greenyellow", label = "Y-direction - X-axis")
lib.plot(mat_lw[2,0:220], c = "lime", label = "X-direction - Y-axis")
lib.plot(mat_lw[3,0:220], c = "limegreen", label = "Y-direction - Y-axis")
lib.plot(mat_lw[4,0:220], c = "green", label = "X-direction - Z-axis")
lib.plot(mat_lw[5,0:220], c = "darkolivegreen", label = "Y-direction - Z-axis")
lib.xlabel("Frames")
lib.ylabel("Difference along the mean")
lib.title("Comparison of the motion tracker of the left wing of the crane along all axes", fontsize = 10)
lib.legend(loc = "lower left", prop = {"size":7})
lib.savefig("lw_means.png")
lib.show()

"""

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪ # # # #

lib.plot(mat_tail[0,0:220], c = "gold", label = "X-direction - X-axis")
lib.plot(mat_tail[2,0:220], c = "lime", label = "X-direction - Y-axis")
lib.plot(mat_tail[4,0:220], c = "green", label = "X-direction - Z-axis")
lib.xlabel("Frames")
lib.ylabel("Difference along the mean")
lib.title("Comparison of the motion tracker of the tail of the crane along the x-axis", fontsize = 10)
lib.legend(loc = "lower left", prop = {"size":7})
lib.savefig("tail_means_x.png")
```

```
lib.show()

lib.plot(mat_tail[1,0:220], c = "greenyellow", label = "Y-direction - X-axis")
lib.plot(mat_tail[3,0:220], c = "limegreen", label = "Y-direction - Y-axis")
lib.plot(mat_tail[5,0:220], c = "darkolivegreen", label = "Y-direction - Z-axis")
lib.xlabel("Frames")
lib.ylabel("Difference along the mean")
lib.title("Comparison of the motion tracker of the tail of the crane along the y-axis", fontsize = 10)
lib.legend(loc = "lower left", prop = {"size":7})
lib.savefig("tail_means_y.png")
lib.show()

"""

lib.plot(mat_tail[0,0:220], c = "gold", label = "X-direction - X-axis")
lib.plot(mat_tail[1,0:220], c = "greenyellow", label = "Y-direction - X-axis")
lib.plot(mat_tail[2,0:220], c = "lime", label = "X-direction - Y-axis")
lib.plot(mat_tail[3,0:220], c = "limegreen", label = "Y-direction - Y-axis")
lib.plot(mat_tail[4,0:220], c = "green", label = "X-direction - Z-axis")
lib.plot(mat_tail[5,0:220], c = "darkolivegreen", label = "Y-direction - Z-axis")
lib.xlabel("Frames")
lib.ylabel("Difference along the mean")
lib.title("Comparison of the motion tracker of the tail of the crane along all axes", fontsize = 10)
lib.legend(loc = "lower left", prop = {"size":7})
lib.savefig("tail_means.png")
lib.show()

"""

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

## DIAGONALIZING THROUGH SVD: REMOVING REDUNDANCIES

Ub, Sb, VTb = np.linalg.svd(mat_beak)
UTb = np.transpose(Ub)

Ur, Sr, VTr = np.linalg.svd(mat_rw)
UTr = np.transpose(Ur)

Ul, Sl, VTl = np.linalg.svd(mat_lw)
UTl = np.transpose(Ul)

Ut, St, VTt = np.linalg.svd(mat_tail)
UTt = np.transpose(Ut)

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

Yb = np.matmul(UTb, mat_beak) # These allow for a change in basis, effectively removing time.
covmatYb = np.cov(Yb) # Determining the dominant spatial directions of motion without time.

print(covmatYb.shape)
print(covmatYb) # It is a diagonal matrix.

Yr = np.matmul(UTr, mat_rw) # These allow for a change in basis, effectively removing time.
covmatYr = np.cov(Yr) # Determining the dominant spatial directions of motion without time.

print(covmatYr.shape)
print(covmatYr) # It is a diagonal matrix.

Yl = np.matmul(UTl, mat_lw) # These allow for a change in basis, effectively removing time.
covmatYl = np.cov(Yl) # Determining the dominant spatial directions of motion without time.

print(covmatYl.shape)
print(covmatYl) # It is a diagonal matrix.

Yt = np.matmul(UTt, mat_tail) # These allow for a change in basis, effectively removing time.
```

```
covmatYt = np.cov(Yt) # Determining the dominant spatial directions of motion without time.

print(covmatYt.shape)
print(covmatYt) # It is a diagonal matrix.

print(Sb.shape)
print(Sb)

t = (1,2,3,4,5,6)

lib.plot(t, Sb, marker = 'o', markersize = 10, c = 'teal')
lib.xlabel('Order of the principal components')
lib.ylabel('Strength of the principal components')
lib.title('Principal components for the beak')
lib.savefig("Principal_Components_beak.png")
lib.show()

# For the beak, use the first principal component only.

print(Sr.shape)
print(Sr)

lib.plot(t, Sr, marker = 'o', markersize = 10, c = 'teal')
lib.xlabel('Order of the principal components')
lib.ylabel('Strength of the principal components')
lib.title('Principal components for the right wing')
lib.savefig("Principal_Components_rw.png")
lib.show()

# For the right wing, use the first principal component only.

print(Sl.shape)
print(Sl)

lib.plot(t, Sl, marker = 'o', markersize = 10, c = 'teal')
lib.xlabel('Order of the principal components')
lib.ylabel('Strength of the principal components')
lib.title('Principal components for the left wing')
lib.savefig("Principal_Components_lw.png")
lib.show()

# For the left wing, use the first TWO principal components.

print(St.shape)
print(St)

lib.plot(t, St, marker = 'o', markersize = 10, c = 'teal')
lib.xlabel('Order of the principal components')
lib.ylabel('Strength of the principal components')
lib.title('Principal components for the tail')
lib.savefig("Principal_Components_tail.png")
lib.show()

# For the tail, use the first principal component only.

# The principal components in this scenario determine the directions in which the variation occurs the most.

lib.plot(Sb[0]*VTb[0,0:220], c = 'firebrick', label = 'Principal component 1')
lib.plot(Sb[1]*VTb[1,0:220], c = 'salmon', label = 'Principal component 2')
lib.xlabel('Frames')
lib.ylabel('Dynamics along the mean')
lib.title('Dynamics of the first two principal components of the beak', fontsize = 10)
lib.legend(loc = 'lower left', prop = {'size':10})
lib.savefig("PCA_beak.png")
lib.show()

lib.plot(Sr[0]*VTr[0,0:220], c = 'firebrick', label = 'Principal component 1')
lib.plot(Sr[1]*VTr[1,0:220], c = 'salmon', label = 'Principal component 2')
```

```
lib.xlabel('Frames')
lib.ylabel('Dynamics along the mean')
lib.title('Dynamics of the first two principal components of the right wing', fontsize = 10)
lib.legend(loc = 'lower left', prop = {'size':10})
lib.savefig("PCA_rw.png")
lib.show()

lib.plot(Sl[0]*VTl[0,0:220], c = 'firebrick', label = 'Principal component 1')
lib.plot(Sl[1]*VTl[1,0:220], c = 'salmon', label = 'Principal component 2')
lib.xlabel('Frames')
lib.ylabel('Dynamics along the mean')
lib.title('Dynamics of the first two principal components of the left wing', fontsize = 10)
lib.legend(loc = 'lower left', prop = {'size':10})
lib.savefig("PCA_lw.png")
lib.show()

lib.plot(St[0]*VTt[0,0:220], c = 'firebrick', label = 'Principal component 1')
lib.plot(St[1]*VTt[1,0:220], c = 'salmon', label = 'Principal component 2')
lib.xlabel('Frames')
lib.ylabel('Dynamics along the mean')
lib.title('Dynamics of the first two principal components of the tail', fontsize = 10)
lib.legend(loc = 'lower left', prop = {'size':10})
lib.savefig("PCA_tail.png")
lib.show()

! pip install pysindy
import pysindy as py

model = py.SINDy()
print(model)

mat_beak_reduced = np.transpose(Ub[0:220,0:1])@mat_beak[:,0:220]

print(mat_beak_reduced.shape)
print(mat_beak_reduced)

mat_beak_reducedT = np.transpose(mat_beak_reduced)
print(mat_beak_reducedT.shape)
print(mat_beak_reducedT)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪ # # # #

mat_rw_reduced = np.transpose(Ur[0:220,0:1])@mat_rw[:,0:220]

print(mat_rw_reduced.shape)
print(mat_rw_reduced)

mat_rw_reducedT = np.transpose(mat_rw_reduced)
print(mat_rw_reducedT.shape)
print(mat_rw_reducedT)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪ # # # #

mat_lw_reduced = np.transpose(Ul[0:220,0:2])@mat_lw[:,0:220]

print(mat_lw_reduced.shape)
print(mat_lw_reduced)

mat_lw_reducedT = np.transpose(mat_lw_reduced)
print(mat_lw_reducedT.shape)
print(mat_lw_reducedT)

mat_lw_reduced2 = np.transpose(Ul[0:220,0:1])@mat_lw[:,0:220]

print(mat_lw_reduced2.shape)
print(mat_lw_reduced2)
```

```
mat_lw_reducedT2 = np.transpose(mat_lw_reduced2)
print(mat_lw_reducedT2.shape)
print(mat_lw_reducedT2)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪ # # # #

mat_tail_reduced = np.transpose(Ut[0:220,0:1])@mat_tail[:,0:220]

print(mat_tail_reduced.shape)
print(mat_tail_reduced)

mat_tail_reducedT = np.transpose(mat_tail_reduced)
print(mat_tail_reducedT.shape)
print(mat_tail_reducedT)

# The size of the matrices was reduced slightly considering the large drop-off to zero experienced in the final
↪ columns.

X1b = mat_beak_reducedT[:,0]
print(X1b.shape)
print(X1b)

X1r = mat_rw_reducedT[:,0]
print(X1r.shape)
print(X1r)

X1l = mat_lw_reducedT[:,0]
X2l = mat_lw_reducedT[:,1]
print(X1l.shape)
print(X2l.shape)
print(X1l)
print(X2l)

X1l2 = mat_lw_reducedT2[:,0]
print(X1l2.shape)
print(X1l2)

X1t = mat_tail_reducedT[:,0]
print(X1t.shape)
print(X1t)

featurenamesb = ['X1b']

featurenamesr = ['X1r']

featurenamesl = ['X1l', 'X2l']
featurenamesl2 = ['X1l2']

featurenamest = ['X1t']

optim = py.STLSQ(threshold = 0.001)
# The threshold was made so small in order to capture more powers of polynomials (the model would be linear
↪ otherwise).

time = np.arange(0,7,7/220)
t = time[0:220]
print(t.shape)

modelb = py.SINDy(feature_names = featurenamesb, optimizer = optim)

# The models should have some kind of symmetry!

modelr = py.SINDy(feature_names = featurenamesr, optimizer = optim)

# The models should have some kind of symmetry!
```

```
modell = py.SINDy(feature_names = featurenamesl, optimizer = optim)

# The models should have some kind of symmetry!

modell2 = py.SINDy(feature_names = featurenamesl2, optimizer = optim)

# The models should have some kind of symmetry!

modelt = py.SINDy(feature_names = featurenamest, optimizer = optim)

# The models should have some kind of symmetry!

modelb.fit(mat_beak_reducedT, t) # Frames by the duration of the video.

modelr.fit(mat_rw_reducedT, t) # Frames by the duration of the video.

modell.fit(mat_lw_reducedT, t) # Frames by the duration of the video.
modell2.fit(mat_lw_reducedT2, t) # Frames by the duration of the video.

modelt.fit(mat_tail_reducedT, t) # Frames by the duration of the video.

print(modelb)
modelb.print()

print(modelr)
modelr.print()

print(modell)
modell.print()

print(modell2)
modell2.print()

print(modelt)
modelt.print()

from scipy.integrate import odeint

def functionb(zb, t):
    a=203.544
    b=1.569
    c=-0.016

    dzbdt = a + b*zb + c*zb**2

    return dzbdt

def functionr(zr, t):
    a=203.544
    b=1.569
    c=-0.016

    dzrdt = a + b*zr + c*zr**2

    return dzrdt

def functionl(zl, t):
    a=203.544
    b=1.569
    c=-0.016

    dz1ldt = a + b*zl[0] + c*zl[1]
    dz2ldt = 0
    dzldt = [dz1ldt, dz2ldt]

    return dzldt

def functionl2(zl2, t):
```

```
    a=203.544
    b=1.569
    c=-0.016

    dzl2dt = a + b*zl2 + c*zl2**2

    return dzl2dt

def functiont(zt, t):
    a=203.544
    b=1.569
    c=-0.016

    dztdt = a + b*zt + c*zt**2

    return dztdt

    ## INITIAL CONDITIONS:

X0b = mat_beak_reducedT[0,0]
print(X0b)

X0r = mat_rw_reducedT[0,0]
print(X0r)

X0l = [mat_lw_reducedT[0,0], mat_lw_reducedT[0,1]]
print(X0l)

X0l2 = mat_lw_reducedT2[0,0]
print(X0l2)

X0t = mat_tail_reducedT[0,0]
print(X0t)

# Solving the ODE:

zb = odeint(functionb, X0b, t)

lib.plot(t,zb[:,0], c = 'cornflowerblue', label = 'Principal component 1')
lib.xlabel('Time', fontsize = 8)
lib.ylabel('Dynamics along the mean', fontsize = 8)
lib.title('Approximation of the dynamics of the first principal component of the beak', fontsize = 8)
lib.savefig("SINDy_beak.png")
lib.show()

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪  # # # #

zr = odeint(functionr, X0r, t)

lib.plot(t,zr[:,0], c = 'cornflowerblue', label = 'Principal component 1')
lib.xlabel('Time', fontsize = 8)
lib.ylabel('Dynamics along the mean', fontsize = 8)
lib.title('Approximation of the dynamics of the first principal component of the right wing', fontsize = 8)
lib.savefig("SINDy_rw.png")
lib.show()

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪  # # # #

zl = odeint(functionl, X0l, t)

lib.plot(t,zl[:,0], c = 'cornflowerblue', label = 'Principal component 1')
lib.plot(t,zl[:,1], c = 'coral', label = 'Principal component 2')
lib.xlabel('Time', fontsize = 8)
lib.ylabel('Dynamics along the mean', fontsize = 8)
lib.title('Approximation of the dynamics of the two first principal components of the left wing', fontsize =
↪  8)
```

```
lib.legend(loc = 'upper left', prop = {'size':10})
lib.savefig("SINDy_lw.png")
lib.show()

zl2 = odeint(functionl2, X0l2, t)

lib.plot(t,zl2[:,0], c = 'cornflowerblue', label = 'Principal component 1')
lib.xlabel('Time', fontsize = 8)
lib.ylabel('Dynamics along the mean', fontsize = 8)
lib.title('Approximation of the dynamics of the first principal components of the left wing', fontsize = 8)
lib.savefig("SINDy_lw2.png")
lib.show()

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪  # # # #

zt = odeint(functiont, X0t, t)

lib.plot(t,zt[:,0], c = 'cornflowerblue', label = 'Principal component 1')
lib.xlabel('Time', fontsize = 8)
lib.ylabel('Dynamics along the mean', fontsize = 8)
lib.title('Approximation of the dynamics of the first principal component of the tail', fontsize = 8)
lib.savefig("SINDy_tail.png")
lib.show()

lib.plot(t, Sb[0]*VTb[0,0:220], c = 'orange', label = 'Exact principal component 1')
lib.plot(t, zb[:,0], c = 'cornflowerblue', label = 'Approximated principal component 1')
lib.xlabel('Time', fontsize = 8)
lib.ylabel('Dynamics along the mean', fontsize = 8)
lib.title('Comparison of the dynamics of the first principal component for the beak', fontsize = 8)
lib.legend(loc = 'lower left', prop = {'size':7})
lib.savefig("Compare1_beak.png")
lib.show()

lib.plot(t, Sr[0]*VTr[0,0:220], c = 'orange', label = 'Exact principal component 1')
lib.plot(t, zr[:,0], c = 'cornflowerblue', label = 'Approximated principal component 1')
lib.xlabel('Time', fontsize = 8)
lib.ylabel('Dynamics along the mean', fontsize = 8)
lib.title('Comparison of the dynamics of the first principal component for the right wing', fontsize = 8)
lib.legend(loc = 'lower left', prop = {'size':7})
lib.savefig("Compare1_rw.png")
lib.show()

lib.plot(t, Sl[0]*VTl[0,0:220], c = 'orange', label = 'Exact principal component 1')
lib.plot(t, (1/1000)*zl[:,0], c = 'cornflowerblue', label = 'Approximated principal component 1')
lib.xlabel('Time', fontsize = 8)
lib.ylabel('Dynamics along the mean', fontsize = 8)
lib.title('Comparison of the dynamics of the first principal component for the left wing', fontsize = 8)
lib.legend(loc = 'upper left', prop = {'size':7})
lib.savefig("Compare1_lw.png")
lib.show()

lib.plot(t, St[0]*VTt[0,0:220], c = 'orange', label = 'Exact principal component 1')
lib.plot(t, (1/1000000000)*zt[:,0], c = 'cornflowerblue', label = 'Approximated principal component 1')
lib.xlabel('Time', fontsize = 8)
lib.ylabel('Dynamics along the mean', fontsize = 8)
lib.title('Comparison of the dynamics of the first principal component for the tail', fontsize = 8)
lib.legend(loc = 'lower right', prop = {'size':7})
lib.savefig("Compare1_tail.png")
lib.show()

lib.plot(t, Sl[1]*VTl[1,0:220], c = 'coral', label = 'Exact principal component 2')
lib.plot(t, zl[:,1], c = 'teal', label = 'Approximated principal component 2')
lib.xlabel('Time', fontsize = 8)
lib.ylabel('Dynamics along the mean', fontsize = 8)
lib.title('Comparison of the dynamics of the second principal component for the left wing', fontsize = 8)
lib.legend(loc = 'lower left', prop = {'size':7})
lib.savefig("Compare2_lw.png")
```

```python
lib.show()

lib.plot(t,zb[:,0], c = 'lightsteelblue', label = 'Principal component 1 of the beak')
lib.plot(t,(1/1000000000)*zt[:,0], c = 'cornflowerblue', label = 'Principal component 1 of the tail')
lib.xlabel('Time', fontsize = 8)
lib.ylabel('Dynamics along the mean', fontsize = 8)
lib.title('Comparison of the SINDy models for the first principal component of the beak and tail', fontsize =
↪  8)
lib.legend(loc = 'lower right', prop = {'size':7})
lib.savefig("SINDy1_bt.png")
lib.show()

# Here, there should be some kind of symmetry, only changing at the very end when the beak folds over.

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪  # # # #

lib.plot(t,zr[:,0], c = 'lightsteelblue', label = 'Principal component 1 of the right wing')
lib.plot(t,(1/1000)*zl[:,0], c = 'cornflowerblue', label = 'Principal component 1 of the left wing')
lib.xlabel('Time', fontsize = 8)
lib.ylabel('Dynamics along the mean', fontsize = 8)
lib.title('Comparison of the SINDy models for the first principal component of the right and left wings',
↪  fontsize = 8)
lib.legend(loc = 'upper left', prop = {'size':7})
lib.savefig("SINDy1_rlw.png")
lib.show()

# Here, they should be perfectly symmetrical.

## FUTURE DIRECTION:

# Run another analysis where we compare motion along the axes rather than the corners.
```