Part 1

In [102…
```python
import numpy as np
import matplotlib.pyplot as plt
import random
import numpy.linalg as la
from typing import List, Tuple, Callable
from tqdm import tqdm
from scipy.stats import multivariate_normal, norm
from scipy.fft import fft, fftfreq
```

Experimental Constants

In [102…
```python
BOLT_NUM = 17
FRAME_RATE = 240
```

Loading vibration Data

In [102…
```python
data_dict = dict(np.load(r'data\\AutonomousTrajectoriesBig.npz'))
data_all = [it[1] for it in data_dict.items()]
```

In [102…
```python
data_dict.keys()
```

Out[102…
```
dict_keys(['T10', 'T11', 'T13', 'T20', 'T4', 'T18', 'T3', 'T14', 'T5', 'T17',
'T9', 'T23', 'T24', 'T21', 'T7', 'T6', 'T22', 'T12', 'T19', 'T15', 'T1', 'T2',
'T8', 'T16'])
```

In [102…
```python
print(len(data_all))
data_all[0].shape
```

```
24
```

Out[102…
```
(17, 7453)
```

1.1

Reversing columns of the data matrices, so that data is in increasing time

In [102…
```python
raw_data = []
for data_mat in data_all:
    raw_data.append(data_mat[:, ::-1])

# ensuring that the data is reversed correctly
for mat_reverse, mat_forward in zip(data_all, raw_data):
    assert np.allclose(mat_reverse[:, ::-1], mat_forward)
```

In [102…
```python
print(raw_data[0][0, :])
print(data_all[0][0, :])
raw_data[0].shape
```

```
[ 0.04094743  0.02806029  0.00584782 ...  0.0129485  -0.00066601
  0.00832467]
[ 0.00832467 -0.00066601  0.0129485  ...  0.00584782  0.02806029
  0.04094743]
```

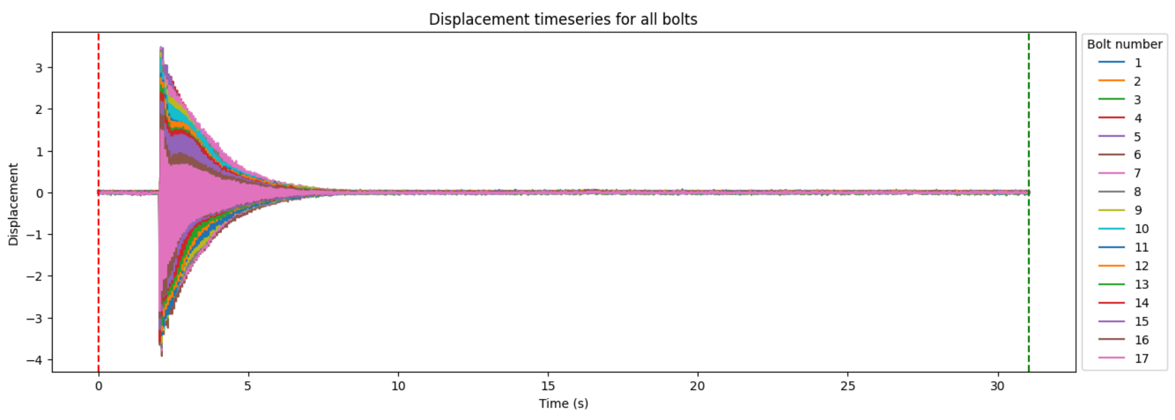
Out[102…

```
(17, 7453)
```

At this point it's probably a good idea to visualise the data to get a clearer picture of the signal and any steady states

In [102…

```python
def get_time_from_frames(frames):
    t_final = frames / FRAME_RATE
    t = np.linspace(0, t_final, frames)
    return t

def plot_timeseries(data, start=0, stop=-1, fig_size=(15, 5), show_legend=True):
    fig, ax = plt.subplots(1, 1, figsize=fig_size)

    if stop == -1:
        stop = data.shape[1]

    # time of each measurement is determined based on the camera frame rate and
    frame_count = data.shape[1]
    t = get_time_from_frames(frame_count)

    # t = t[start:stop+1]
    # plotting the timeseries data for each bolt
    for i in range(BOLT_NUM):
        ax.plot(t, data[i, :], label=f'{i+1}')

    ax.axvline(x=t[start], color='r', linestyle='--')
    ax.axvline(x=t[stop-1], color='g', linestyle='--')
    ax.set_xlabel('Time (s)')
    ax.set_ylabel('Displacement')

    if show_legend:
        ax.legend(title='Bolt number', loc='center left', bbox_to_anchor=(1, 0.5

    plt.title('Displacement timeseries for all bolts')
    plt.show()

plot_timeseries(raw_data[0], start=0, stop=-1)
```

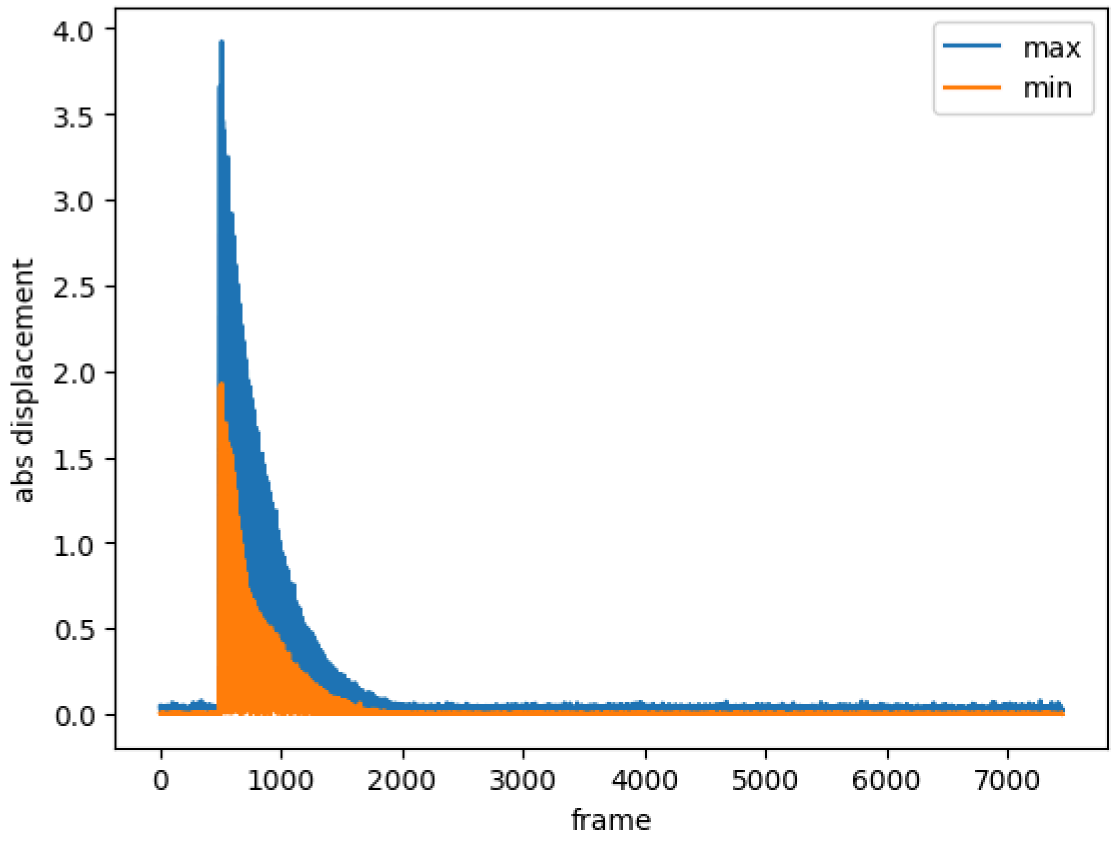


1.2

We can clearly see the initial steady state and the long time steady state from the above plot. We can also observe some variation in the initial amplitude of oscillation for the different bolts, as well as in the steady state noise. Let's visualise the minimum and maximum absolute displacement for one of the time series in the dataset, to determine a simple approach for removing steady state data.

In [103…

```
max_series = np.max(np.abs(raw_data[0]), axis=0)
min_series = np.min(np.abs(raw_data[0]), axis=0)
plt.plot(max_series, label="max")
plt.plot(min_series, label="min")
plt.legend()
plt.xlabel('frame')
plt.ylabel('abs displacement')
plt.show()
```

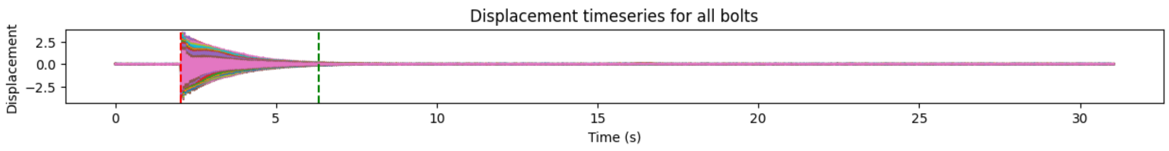


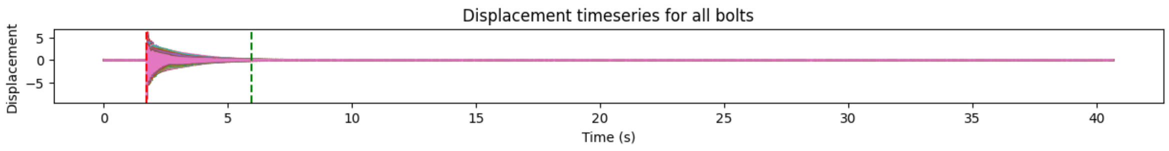
We can see that there is a clear spike in both the maximum and minimum absolute displacement (distance I suppose) and that the steady state noise for the maximum timeseries is consistently higher than the steady state noise for the minimum. We can use this to our advantage to seperate the transient behaviour from steady state, by selecting the maximum of the maximum absolute displacement timeseries for t>20, then using this as a threshold on the minimum displacement.

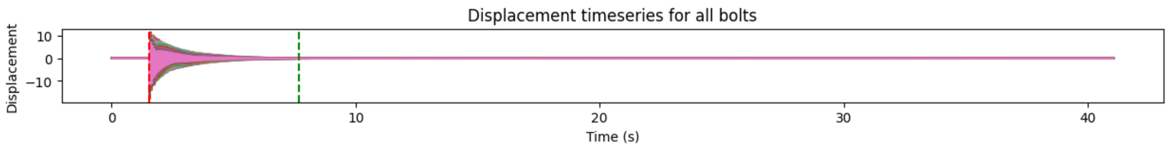
In [103…

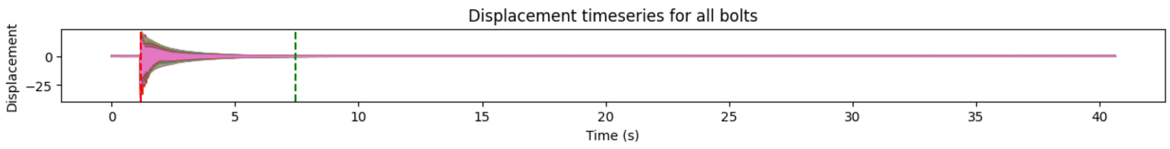
```
def find_steady_state(data):
    # timeseries of min and max distance from 0 for each bolt
    max_series = np.max(np.abs(data), axis=0)
    min_series = np.min(np.abs(data), axis=0)

    # getting the largest distance from the steady state max_series after t=20s
    t = get_time_from_frames(data.shape[1])
    steady_start = np.where(t > 20)[0][0]
```

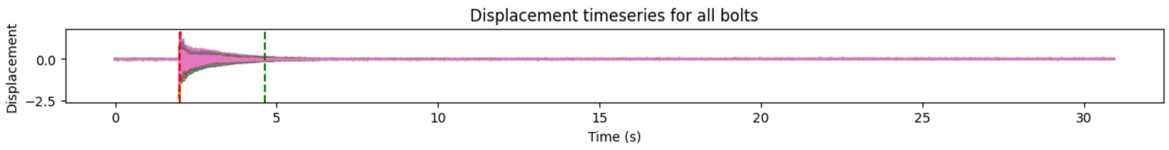
```python
    threshold = np.max(max_series[steady_start:])

    # getting indices of the transient state using max steady state threshold
    transient_indices = np.where(min_series > threshold)[0]

    # stop at the last transient index
    stop = transient_indices[-1]
    start = transient_indices[0]

    return start, stop

# visually validating the start and stop positions
processed_data = []
for i, data in enumerate(raw_data):
    start, stop = find_steady_state(data)
    plot_timeseries(data, start=start, stop=stop, fig_size=(15, 1), show_legend=
    # trimming the data
    processed_data.append(data[:, start:stop])
```

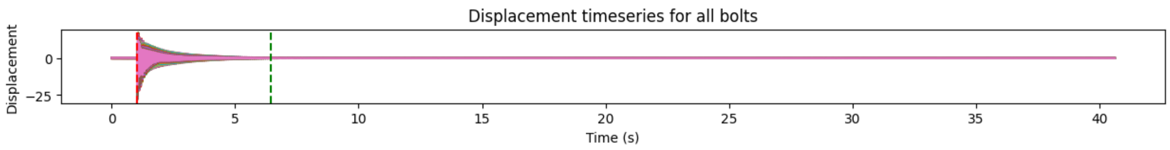


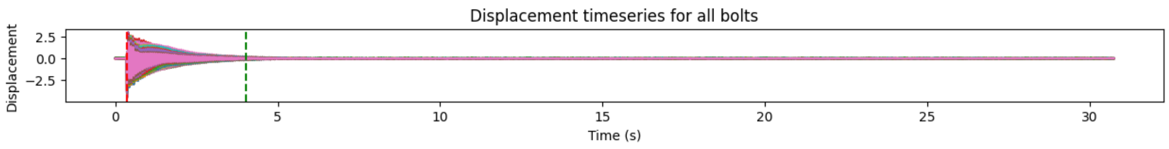


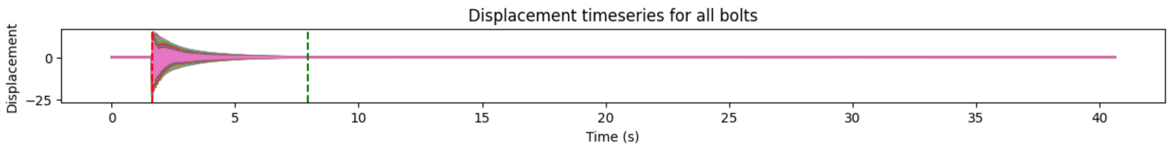


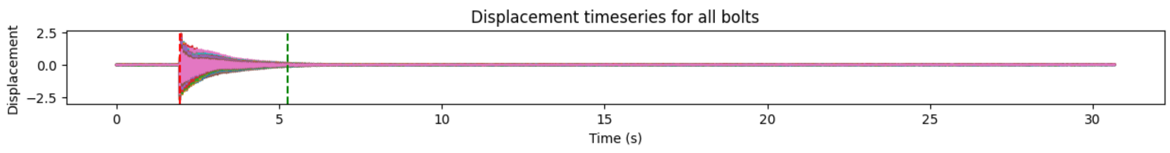


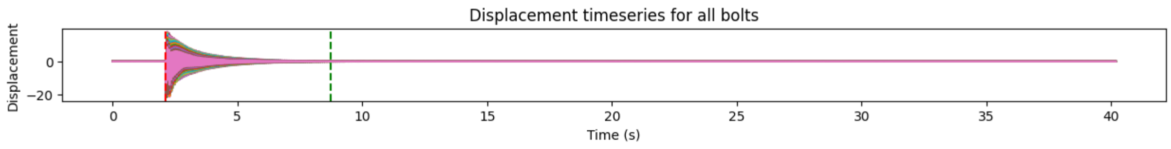


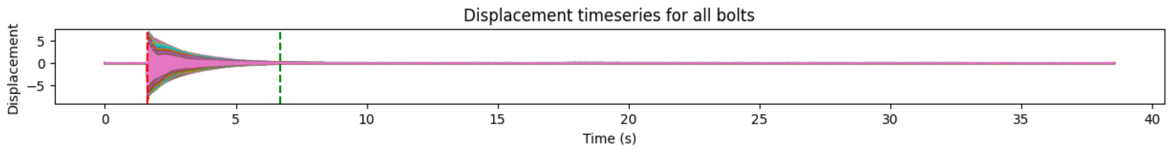


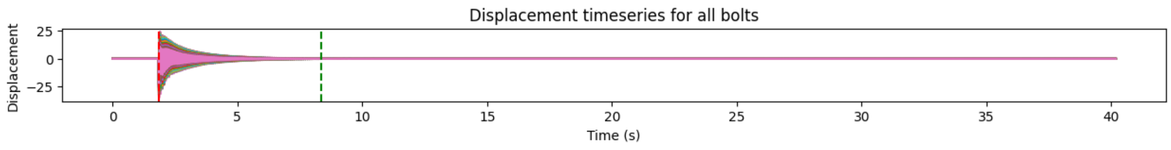


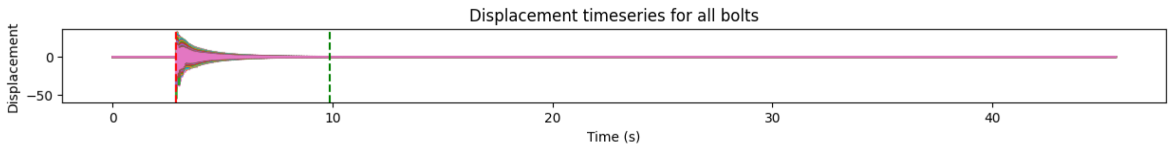


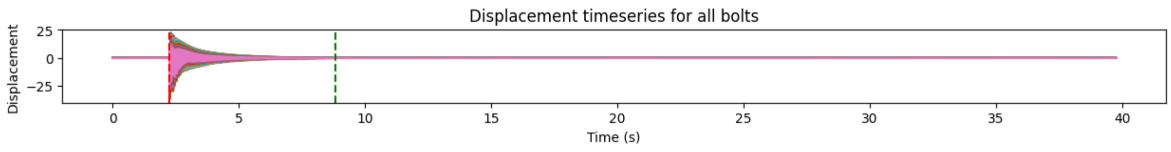

Displacement timeseries for all bolts
Displacement
2.5
0.0
−2.5
0
5
10
15
20
25
30
Time (s)

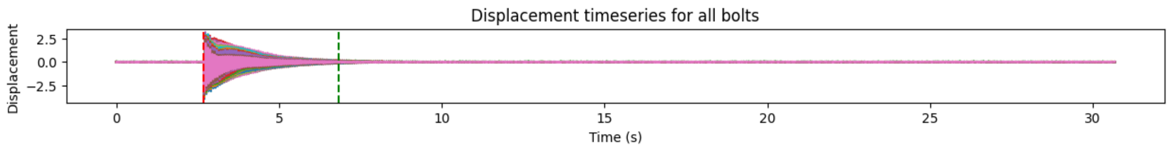

Displacement timeseries for all bolts
Displacement
0
−20
0
5
10
15
20
25
30
35
40
Time (s)

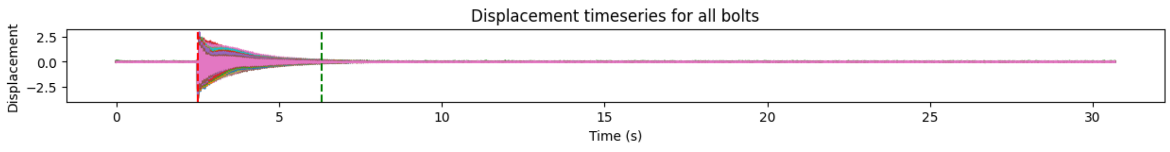

Displacement timeseries for all bolts
Displacement
5
0
−5
0
5
10
15
20
25
30
35
40
Time (s)

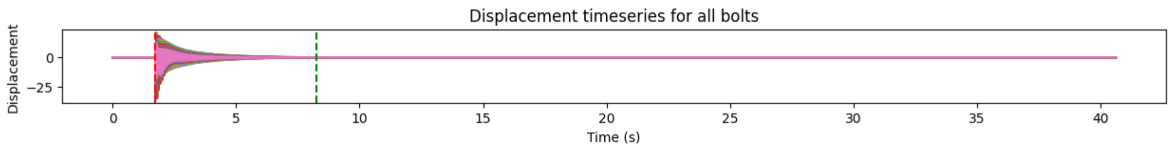

Displacement timeseries for all bolts
Displacement
25
0
−25
0
5
10
15
20
25
30
35
40
Time (s)

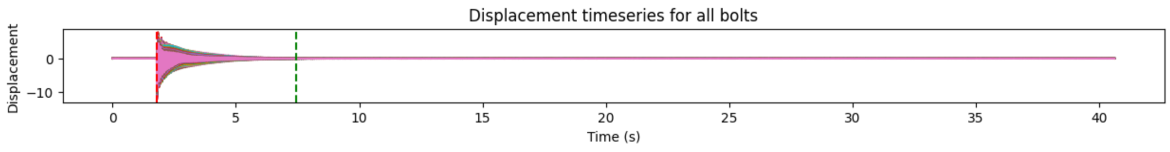

Displacement timeseries for all bolts
Displacement
0
−50
0
10
20
30
40
Time (s)

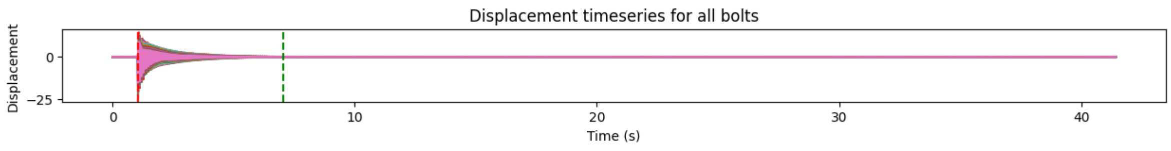

Displacement timeseries for all bolts
Displacement
25
0
−25
0
5
10
15
20
25
30
35
40
Time (s)

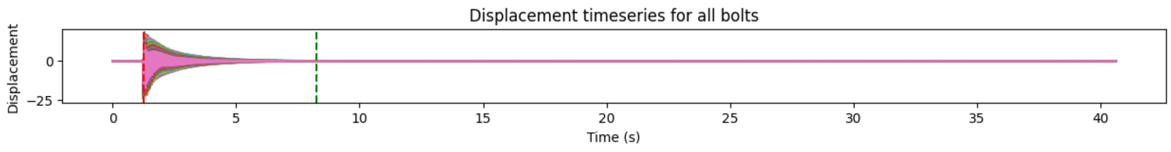

Displacement timeseries for all bolts
Displacement
2.5
0.0
−2.5
0
5
10
15
20
25
30
Time (s)

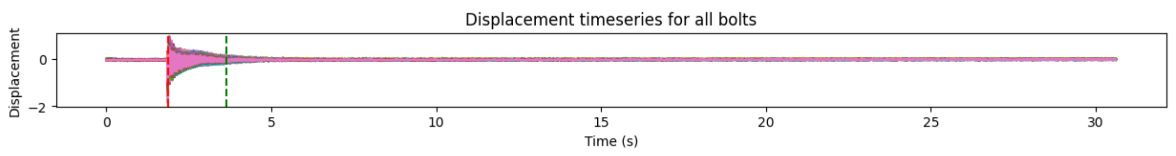

Displacement timeseries for all bolts
Displacement
2.5
0.0
−2.5
0
5
10
15
20
25
30
Time (s)

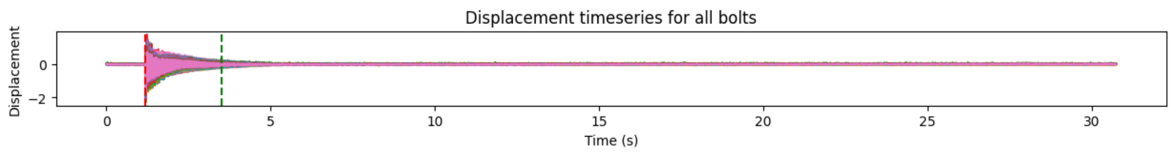

Displacement timeseries for all bolts
Displacement
0
−25
0
5
10
15
20
25
30
35
40
Time (s)

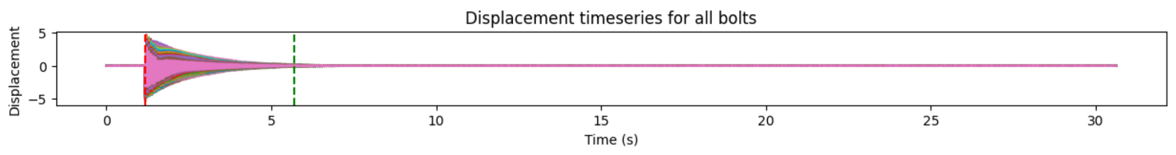

Displacement timeseries for all bolts
Displacement
0
−10
0
5
10
15
20
25
30
35
40
Time (s)

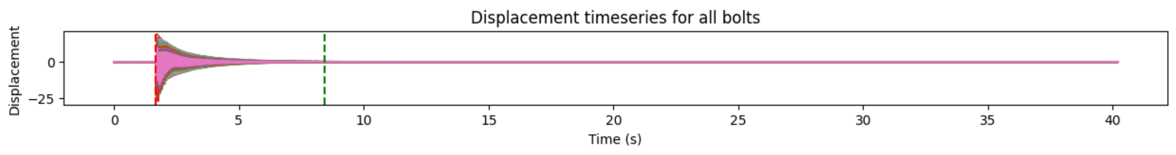

Displacement timeseries for all bolts
Displacement
0
−25
0
10
20
30
40
Time (s)


Displacement timeseries for all bolts
Displacement
0
−25
0
5
10
15
20
25
30
35
40
Time (s)









In this case, since we have a relatively small number of trials, I can visually verify that this simple filtering approach has worked as intended. For a larger data set, perhaps a slightly more robust approach would be required.

1.3

Now let's ensure that each trajectory has zero mean (oscillation about 0 displacement).

In [103…
```
def zero_mean(data):
    return data - np.mean(data, axis=1, keepdims=True)

processed_data = [zero_mean(data) for data in processed_data]

# validating that zeroing the mean has worked
for data in processed_data:
    assert np.allclose(np.mean(data, axis=1), 0, rtol=1e-15), 'Mean is not zeroe
```

We cannot trust that the steady state is at the same numerical value for all trajectories for three possible reasons. One is that there may be some uncertainty associated with the tracking of each bolt using computer vision. Two, there may be some camera distortions such as fish eye that have not been properly accounted for, leading to increasing measurement error for bolts away from the centre of the camera frame. Finally, the experimental setup doesn't appear to be rigorously leveled, so the physical location of the bolts may not actually lie on a perfectly level straight line.

1.4

We do not know that the damping in the system is linear, therefore we cannot be certain that it is independent of the magnitude of displacement, so the rescaling should take place accross all matrices aggregated instead of individually. scaling values between 1 and -1 can be benificial for numerical stability, since it reduces the scale difference between small and large losses. If we collected more data, we would need to rescale it the same amount as the original data, so it's generally a good idea to keep track of the scaling factor. Here we are not going to be dealing with further data, so I will not.

In [103…

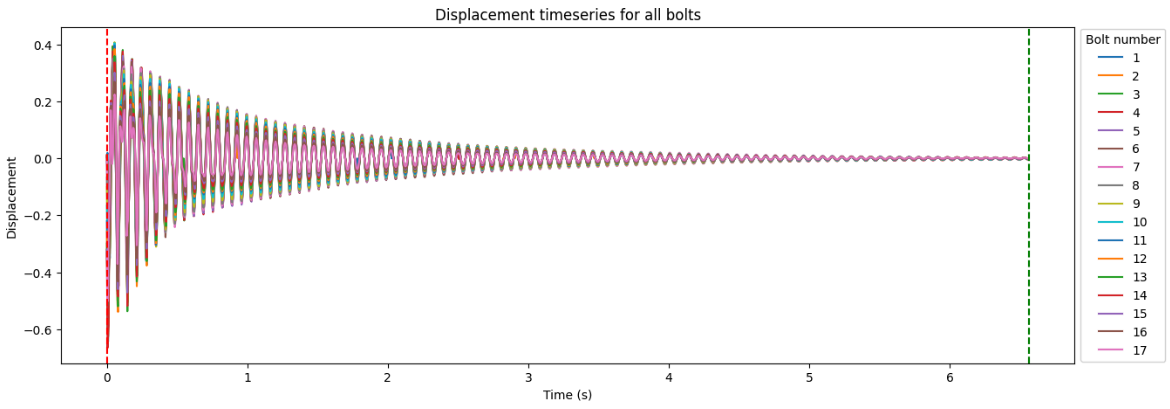
```python
print("shape pre scaling", processed_data[0].shape, "\n")

scale = -np.inf
for data in processed_data:
    abs_max = np.max(np.abs(data))
    if abs_max > scale:
        scale = abs_max

print("max absolute displacement in dataset =", scale)
print("---------------------------------------")

for i, data in enumerate(processed_data):
    processed_data[i] = data / scale

# validating that the data has been scaled correctly. There should be a 1 presen
for i, data in enumerate(processed_data):
    max_displacement = np.max(np.abs(data))
    assert (max_displacement <= 1), 'Data is not scaled correctly'
    print("test {}, max displacement = {}".format(i, max_displacement))

# checking the shape of the data remains unchanged
print("\nshape post scaling", processed_data[0].shape)
```

```
shape pre scaling (17, 1029)

max absolute displacement in dataset = 55.46600266290267
---------------------------------------
test 0, max displacement = 0.06985011878146498
test 1, max displacement = 0.15675990755397168
test 2, max displacement = 0.3231608920870714
test 3, max displacement = 0.6553487638960437
test 4, max displacement = 0.04281049391076513
test 5, max displacement = 0.5073179110590731
test 6, max displacement = 0.08105039330085374
test 7, max displacement = 0.4435932784550382
test 8, max displacement = 0.04901883259086211
test 9, max displacement = 0.39190776225625984
test 10, max displacement = 0.14821380801316614
test 11, max displacement = 0.6315495385753634
test 12, max displacement = 1.0
test 13, max displacement = 0.6657811659736601
test 14, max displacement = 0.0722840014285362
test 15, max displacement = 0.06501025213200393
test 16, max displacement = 0.6533942719476019
test 17, max displacement = 0.21652699079428922
test 18, max displacement = 0.4454424858929594
test 19, max displacement = 0.44100896055539723
test 20, max displacement = 0.03332850150358394
test 21, max displacement = 0.04083807873508804
test 22, max displacement = 0.09914561390220225
test 23, max displacement = 0.49128395646331824

shape post scaling (17, 1029)
```

plotting the final dataset as a sanity check

In [103…

```
plot_timeseries(processed_data[random.randint(0, len(processed_data)-1)], start=
```



1.5

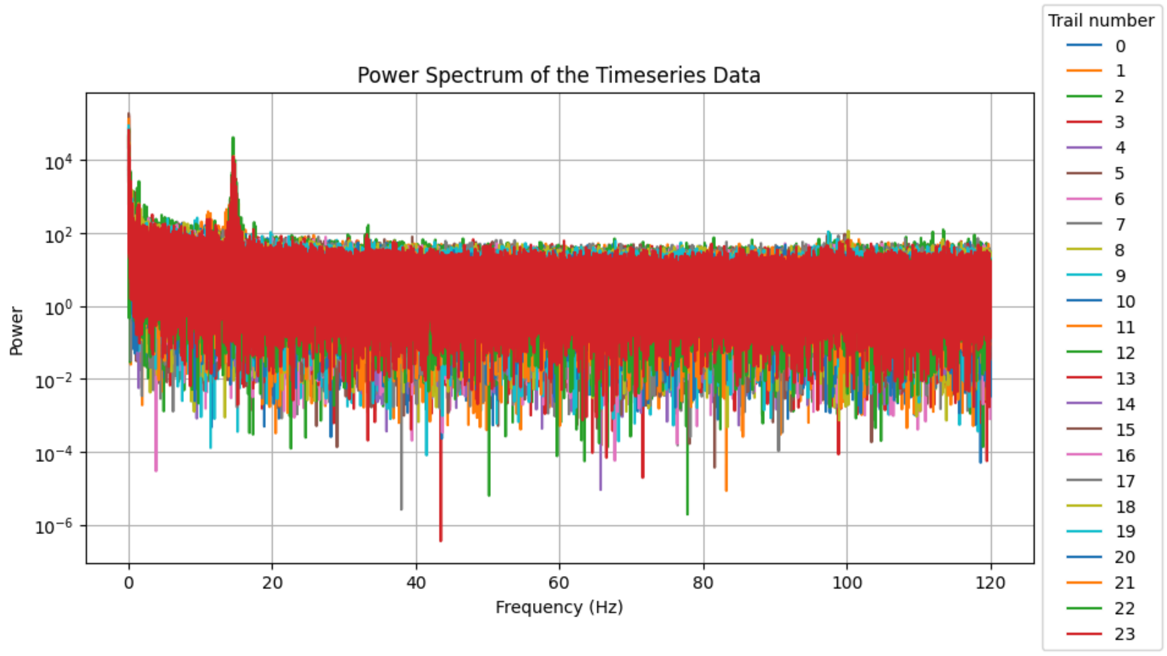
We can visualise the power spectrum of the timeseries data using the FFT. Spikes in the power spectrum tell us the frequencies of the dominant sinusoids that approximate the signal.

In [103…

```
def visualise_power_spectra(data, start, stop=None):
    power_spectra = []
    for i in range(len(raw_data)):
        # Extract the timeseries data
        timeseries_data = raw_data[i][:, start:stop]
        timeseries_data = timeseries_data.flatten() # flattening to combine stea

        # Number of samples
        N = len(timeseries_data)

        # Compute the FFT
        fft_values = fft(timeseries_data)

        # Compute the corresponding frequencies
        frequencies = fftfreq(N, 1 / FRAME_RATE)

        # Compute the power spectrum
        power_spectrum = np.abs(fft_values)**2

        power_spectra.append((frequencies, power_spectrum))

    # Plot the power spectrum
    plt.figure(figsize=(10, 5))
    for trail_num, (frequencies, power_spectrum) in enumerate(power_spectra):
        pos_inds = np.where(frequencies > 0) # Plot only the positive frequencie
        plt.plot(frequencies[pos_inds], power_spectrum[pos_inds], label=str(trai
    plt.yscale('log')
    # plt.xscale('log')
    plt.xlabel('Frequency (Hz)')
    plt.ylabel('Power')
    plt.title('Power Spectrum of the Timeseries Data')
    plt.grid(True)
    plt.legend(title='Trail number', loc='center left', bbox_to_anchor=(1, 0.5))
    plt.show()
```

In [103…

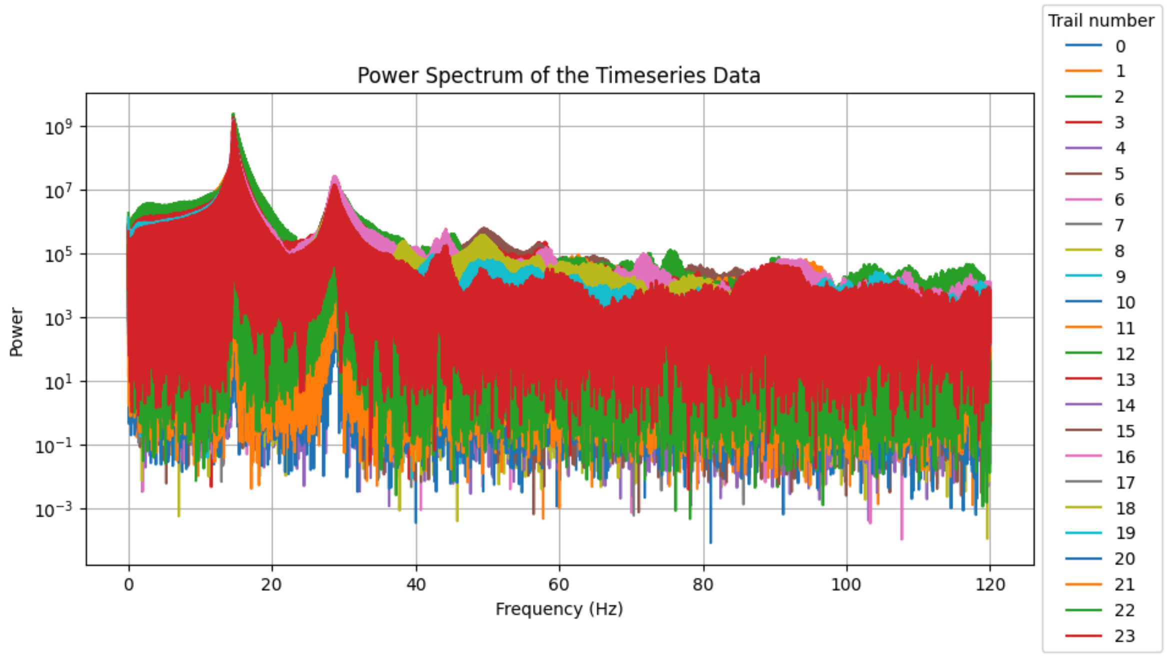
```
# getting the steady state as before
t = np.linspace(0, raw_data[0].shape[1] / FRAME_RATE, raw_data[0].shape[1])
steady_start = np.where(t > 20)[0][0]

visualise_power_spectra(raw_data, start=steady_start)
```



We see that there is a notable spike in the power spectrum around 17 Hz for all of the trials. If we then look at the power spectrum for the transient data in processed_data, we see that the dominant frequency in the power series again occurs at around 17Hz. This suggests that even in the apparently steady state behaviour of the system the metal plate is still vibrating at 17Hz with very low amplitude.

In [103…

```
visualise_power_spectra(processed_data, start=0)
```



2.1

It is necissary to seperate the data into a training and testing set to ensure that our model is not overfit to the training data. If it performs well on the training data, but poorly on the test data, the model is likely overfit.

Since we don't have a lot of data, let's do a 70-30 split

In [103…

```
train_ratio = 0.7
shuffled_data = random.sample(processed_data, len(processed_data))
train_data = shuffled_data[:int(len(shuffled_data)*train_ratio)]
test_data = shuffled_data[int(len(shuffled_data)*train_ratio):]

print(len(train_data), len(test_data))
```

```
16 8
```

2.2

We are using delay embedding to determine the states and linear model which generate the observed data. Assuming that the system is observable. In this case If we assume the system is observable, we have the equations

$$x_{k+1} = Ax_k$$
$$y_k = Cx_k$$

where $y_k \in \R^{17}, \ x_k \in \R^{d^*}, \forall k$ where d^* is the unkown dimensionality of the state space of the data.

if the system is observable then a surrogate of the unkown state space is

$$\hat{x}_k = \begin{pmatrix} y+k \\ \vdots \\ y_{k+d-1} \end{pmatrix}$$

where d is some sufficiently large delay embedding. If we consider multiple observations of the state surrogate taking unit steps in time, we end up with the hankel matrix

$$X = \begin{pmatrix} y_1 & y_2 & \cdots & y_{N-d+1} \\ y_2 & y_3 & \cdots & y_{N-d+2} \\ \vdots & \vdots & \ddots & \vdots \\ y_d & y_{d+1} & \cdots & y_N \end{pmatrix}$$

We can decorrelate X row wise to get $\hat{X}$ so that that the rows of the delay embedding matrix are as close to independent as possible. Since the rows correspond to timeseries of each of the delay dimensions, this amounts to ensuring that the timeseries encode unique information and helps with numerical stability. To fit a linear model to this decorrelated delay embedding matrix, we can consider

$$X_{\text{prev}} = \hat{X}[:, : -1]$$
$$X_{\text{next}} = \hat{X}[:, 1 :]$$

and train a linear model to predict the next state surrogate from the previous

$$\hat{x}_{k+1} = W\hat{x}_k$$

If we want to extract the predicted next output state from $\hat{x}_{k+1}$ (unseen y_{d+1} in above example) then we need to compute

$$y_k = \hat{U}[-17 :, :]\hat{x}_k$$

which corresponds to the last delay embedding entry in $\hat{x}_{k+1}$

In [103…

```
def hankel_matrix(data: np.array, delay: int):
    """
    Generates a hankel matrix by vertically stacking the delayed versions of the
    args:
    ------
    data: np.array, timeseries of shape (n,m) where m is the number of timesteps
    delay: int, delay window size

    returns:
    --------
    hankel: np.array, hankel matrix of shape (delay*n, m-delay+1)
    """
    n = data.shape[0]
    m = data.shape[1]

    hankel = np.zeros((delay*n, m-delay+1))
    for i in range(delay):
        hankel[i*n:(i+1)*n, :] = data[:, i:m-delay+(i+1)]

    return hankel

# let's test that this implementation is correct with a small test vector
simple_hankle_test = np.array([[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]])
hankle_test_mat = hankel_matrix(simple_hankle_test, 4)
print("1D test:")
print(hankle_test_mat)
# and with multidimensional data
simple_hankle_test = np.array([[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]])
simple_hankle_test = np.vstack((simple_hankle_test, simple_hankle_test))
hankle_test_mat = hankel_matrix(simple_hankle_test, 4)
print("2D test:")
print(hankle_test_mat)
```

```
1D test:
[[ 1.   2.   3.   4.   5.   6.   7.]
 [ 2.   3.   4.   5.   6.   7.   8.]
 [ 3.   4.   5.   6.   7.   8.   9.]
 [ 4.   5.   6.   7.   8.   9.  10.]]
2D test:
[[ 1.   2.   3.   4.   5.   6.   7.]
 [ 1.   2.   3.   4.   5.   6.   7.]
 [ 2.   3.   4.   5.   6.   7.   8.]
 [ 2.   3.   4.   5.   6.   7.   8.]
 [ 3.   4.   5.   6.   7.   8.   9.]
 [ 3.   4.   5.   6.   7.   8.   9.]
 [ 4.   5.   6.   7.   8.   9.  10.]
 [ 4.   5.   6.   7.   8.   9.  10.]]
```

This deccorelation function can be used in our delay embedding procedure to create the state surrogate from the hankel matrix. The process of decorrelation removes any strongly correlated information between rows of the state surrogate, which will improve the model performance.

In [104…

```python
def decorrelate(data: np.array, kappa: float = 1e6, verbose: bool = True):
    """
    args:
    ------
    data: np.array, data matrix to decorrelate, row wise, shape (n, m)
    kappa: float, stability condition for the singular values

    returns:
    --------
    decorrelated: np.array, decorrelated data matrix, row wise, shape (n*, m), w
    corresponds to the number of stable singular values
    U: np.array, left singular vectors of the data matrix that satisfy the stabi

    """
    # performing SVD on the correlation matrix
    cor_mat = data @ data.T
    U, s_vec, _ = np.linalg.svd(cor_mat, full_matrices=False, hermitian=True) #
    # selecting columns of u that satisfy stability condition
    significant_cols = np.where(s_vec[0] / s_vec < kappa)[0]
    U = U[:, significant_cols]
    decorrelated = U.T @ data

    if verbose:
        print("Decorrelating data: \n")
        print(" data shape: ", data.shape, "\n")
        print(" first 5 singular values: ", s_vec[:5], "\n")
        print(" U shape after filtering: ", U.shape, "\n")
        print(" decorrelated data shape: ", decorrelated.shape, "\n")
        print("-----------------------------\n")
    if decorrelated.shape != data.shape:
        print("decorrelated matrix has less rows that input data for numerical s

    return decorrelated, U

# simple decorrelation test
simple_decor, simple_U = decorrelate(hankle_test_mat, kappa=1e6)
print("\ndecorrelated test:")
print(simple_decor)
# and returning to correlated space
```

```
simple_correlated = simple_U @ simple_decor
print("\nrecovered state surrogate:")
print(simple_correlated)
```

```
Decorrelating data:

 data shape:  (8, 7)

 first 5 singular values:  [1.98008113e+03 7.91886744e+00 2.04821440e-13 1.038874
81e-13
 4.65642867e-14]

 U shape after filtering:  (8, 2)

 decorrelated data shape:  (2, 7)

----------------------------

decorrelated matrix has less rows that input data for numerical stability

decorrelated test:
[[ -7.51994279 -10.30349632 -13.08704986 -15.87060339 -18.65415693
  -21.43771046 -24.221264  ]
 [  1.85754151   1.35571514   0.85388876   0.35206238  -0.149764
   -0.65159038  -1.15341676]]

recovered state surrogate:
[[ 1.  2.  3.  4.  5.  6.  7.]
 [ 1.  2.  3.  4.  5.  6.  7.]
 [ 2.  3.  4.  5.  6.  7.  8.]
 [ 2.  3.  4.  5.  6.  7.  8.]
 [ 3.  4.  5.  6.  7.  8.  9.]
 [ 3.  4.  5.  6.  7.  8.  9.]
 [ 4.  5.  6.  7.  8.  9. 10.]
 [ 4.  5.  6.  7.  8.  9. 10.]]
```

We see that since rows of this test matrix are highly correlated, the decorrelation process actually reduced the number of rows for stability reasons.

Now we need to fit a linear model to the delay embedded and decorellated data to produce the observed output data. When calculating the prediction error, we can avoid the issue of varying dimensionality, by calculating loss in the observation space instead of surrogate state space.

In [104…

```
def linear_delay_embedding(data: np.array, delay: int = 10, kappa: float = 1e6,
    """
    Fits a least squares linear model to predict the next state surrogate from t
    using a delay embedding size given by delay.

    args:
    -----
    data: np.array, timeseries of shape (n, m) where m is the number of timestep
    delay: int, delay embedding window size
    kappa: float, threshold for decorrelation
    svd_order: int, number of singular values to keep for decorrelation, if None

    returns:
    --------
    W: np.array, linear model weights for predicting the next state surrogate fr
    U: np.array, LHS Matrix from SVD, for switching between decorrelated state s
    X_prev: np.array, decorrelated previous state surrogate matrix (n*delay, m-d
    X_next: np.array, decorrelated next state surrogate matrix (n*delay, m-delay
    """
    # delay embedding the observed outputs
    hankel = hankel_matrix(data, delay)
    # previous timestep and next timestep state surrogate matrices
    X_prev = hankel[:, :-1]
    X_next = hankel[:, 1:]
    # decorrelated input state surrogate matrix
    _, U = decorrelate(hankel, kappa=kappa, verbose=False)
    # truncating the left singular vectors to svd_order
    U = U[:, :svd_rank]
    # decorrelated state surrogates
    X_prev = U.T @ X_prev # shape (n* * delay, m-delay)
    X_next = U.T @ X_next # this is the target for the linear model

    # fitting a linear model to predict the next surrogate state, i.e. X_next = 
    W = (X_next @ X_prev.T) @ la.pinv(X_prev @ X_prev.T, rcond=1e-8)

    return W, U, X_prev, X_next

def predict(W: np.array, U: np.array, data: np.array, delay: int = 10):
    """
    predicts the output state at the next timestep using the linear delay embedd
    with learned weights W and delay d, with decorrelation matrix U.

    args:
    -----
    W: np.array, linear model weights
    U: np.array, LHS Matrix from SVD, for decorrelation
    data: np.array, test trajectory of shape (n,m) where m is the number of time
    delay: int, delay embedding window size

    returns:
    --------
    Y_pred: np.array, predicted output state starting at column index i=delay in
            the final column, shape (n, m-delay)
    prediction_rmse: float, root mean squared error of the prediction vs the act
    """
    # delay embedding matrix
    hankel = hankel_matrix(data, delay)
    # decorrelated previous state surrogate matrix
    X_prev = U.T @ hankel[:, :-1]

    # predicting the next state surrogate
    X_next_pred = W @ X_prev

    # predicting the output to the unseen data (i.e one timestep ahead of the de
    Y_pred = U[-data.shape[0]:, :] @ X_next_pred
    Y_actual = data[:, delay:]

    # to avoid the delay affecting the rmse, we can go back to the output space 
    # between the predicted and observed outputs (taking into account the delay)

    # calculating the rmse for each timestep over the 17 bolts
    rmse = la.norm(Y_pred - Y_actual, axis=0) / np.sqrt(Y_actual.shape[0]) # sha

    # Original approach using the state surrogates and scaling by norm. decided 
    # Rescaled to account for the dimensionality of the data (mean term cancels
    # rmse = la.norm(X_next - X_next_pred, axis=0) / la.norm(np.ones(X_next.shap
    # standard deviation of rmse accross features

    # mean error is then taken over each timestep
    mean_error = np.mean(rmse)
    std_error = np.std(rmse)

    return Y_pred, mean_error, std_error

# training and testing the linear delay embedding model
stacked_train_data = np.hstack(train_data)
stacked_test_data = np.hstack(test_data)

delay = 5 # test delay size
kappa = 1e6
W, U, _, _ = linear_delay_embedding(stacked_train_data, delay=delay, kappa=kappa

_, train_error_mean, _ = predict(W, U, stacked_train_data, delay=delay)
test_preds, test_rmse, _ = predict(W, U, stacked_test_data, delay=delay)

print("test with delay = ", delay)
print("mean train rmse: ", train_error_mean)
print("mean test mse rmse: ", test_rmse)
```

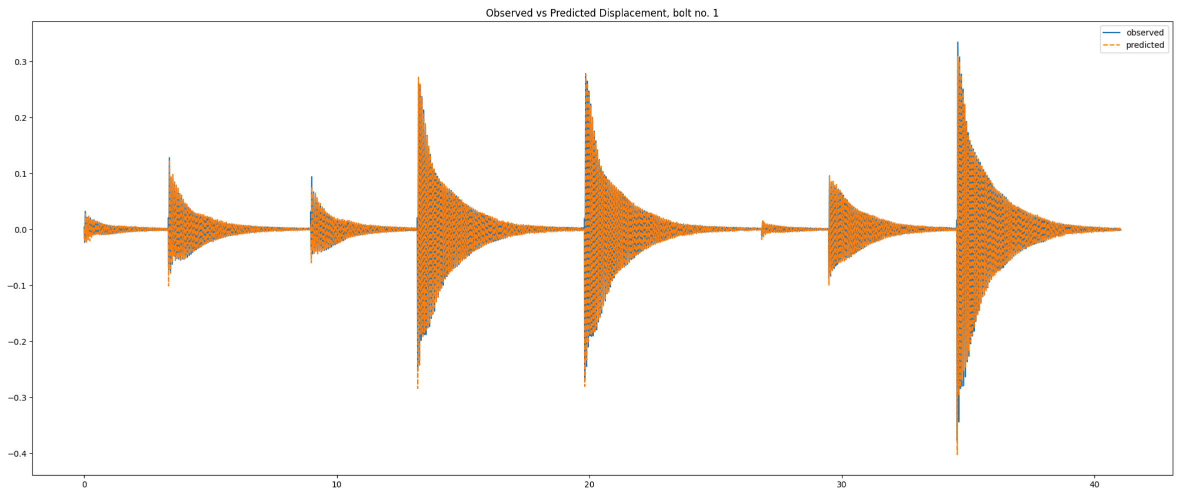
```
test with delay =  5
mean train rmse:  0.0021954229795845518
mean test mse rmse:  0.0021381703419111864
```

Now that we have a delay embedding regime that appears to work, lets just do a quick sanity check to validate that it is correctly predicting outputs

In [104…

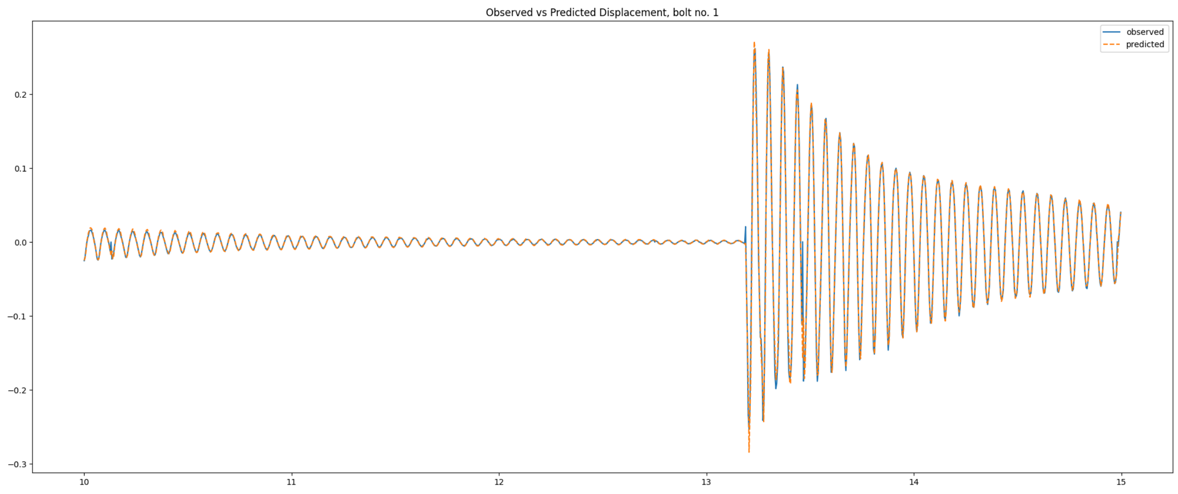
```
test_bolt_actual = stacked_test_data[0, :]
test_bolt_pred = test_preds[0, :]

test_bolt_pred = np.concatenate([np.zeros(delay), test_bolt_pred])

print("test series shape", test_bolt_actual.shape)
print("Y_pred shape", test_bolt_pred.shape)

fig, ax = plt.subplots(1, 1, figsize=(25, 10))
t = get_time_from_frames(test_bolt_actual.shape[0])
ax.plot(t, test_bolt_actual, label='observed')
ax.plot(t, test_bolt_pred, label='predicted', linestyle='--')
ax.legend()
plt.title('Observed vs Predicted Displacement, bolt no. 1')
plt.show()
```

```
test series shape (9850,)
Y_pred shape (9850,)
```



Looks to be working well. Lets zoom in to a single prediction so that we can see if there is anything funky happening at the transition between timeseries.

```
In [104…
fig, ax = plt.subplots(1, 1, figsize=(25, 10))
inds = np.where(np.logical_and(t>10, t<15))[0]
t = t[inds]
ax.plot(t, test_bolt_actual[inds], label='observed')
ax.plot(t, test_bolt_pred[inds], label='predicted', linestyle='--')
ax.legend()
plt.title('Observed vs Predicted Displacement, bolt no. 1')
plt.show()
```



We can see that there is some small error at the peaks and troughs of the oscillations and that the transition to the next timeseries does seem to impose a small amount of error, but this is not significant since it only occurs for a small fraction of the timesteps. We can also see that the model does not predict the slight anomalies in the oscillation seen between t=14 and t=15, which is a good initial sign that the model is not overfit.

2.2 and 2.3

Lets find the smallest delay such that the difference between the train and test error is smaller than 10%. We can also look at the standard deviation during this task.

```
In [104…
def train_and_compare(train, test, delay, kappa=1e6):
    W, U, _, _ = linear_delay_embedding(stacked_train_data, delay=delay, kappa=k
    _, train_error_mean, train_error_std = predict(W, U, stacked_train_data, del
    _, test_error_mean, test_error_std = predict(W, U, stacked_test_data, delay=

    error_ratio = np.abs(test_error_mean - train_error_mean)/train_error_mean

    return error_ratio, (train_error_mean, test_error_mean), (train_error_std, t

# testing the error ratio for different delay values
max_delay = 30
ratios = []
train_stds = []
test_stds = []
train_means = []
test_means = []

delays = list(range(1, max_delay+1))
for delay in delays:
    error_ratio, means, stds = train_and_compare(stacked_train_data, stacked_tes

    print(f"delay = {delay}, error ratio = {error_ratio}")

    # storing values for plotting
    ratios.append(error_ratio)
    train_stds.append(stds[0])
    test_stds.append(stds[1])
    train_means.append(means[0])
    test_means.append(means[1])
```

```
delay = 1, error ratio = 0.1116539613369719
delay = 2, error ratio = 0.04986598250995796
delay = 3, error ratio = 0.03278975951583507
delay = 4, error ratio = 0.02923458485828219
delay = 5, error ratio = 0.02607818092721224
delay = 6, error ratio = 0.02438437819808542
delay = 7, error ratio = 0.015168449349942508
delay = 8, error ratio = 0.0043265720972040485
delay = 9, error ratio = 0.0012024068800261523
delay = 10, error ratio = 0.009243522993492268
delay = 11, error ratio = 0.015141268039566938
delay = 12, error ratio = 0.02932822684905958
delay = 13, error ratio = 0.039010993854645244
delay = 14, error ratio = 0.04142515176285613
delay = 15, error ratio = 0.04932960987997138
delay = 16, error ratio = 0.057702124625483846
delay = 17, error ratio = 0.0733622271637482
delay = 18, error ratio = 0.08244200148664646
delay = 19, error ratio = 0.08957949436190984
delay = 20, error ratio = 0.10311858531428567
delay = 21, error ratio = 0.11263100848946346
delay = 22, error ratio = 0.12031859010746396
delay = 23, error ratio = 0.12563916336817765
delay = 24, error ratio = 0.13287356252219104
decorrelated matrix has less rows that input data for numerical stability
delay = 25, error ratio = 0.14424244233740457
decorrelated matrix has less rows that input data for numerical stability
delay = 26, error ratio = 0.14675429468682383
decorrelated matrix has less rows that input data for numerical stability
delay = 27, error ratio = 0.1499387298571384
decorrelated matrix has less rows that input data for numerical stability
delay = 28, error ratio = 0.15721879606833075
decorrelated matrix has less rows that input data for numerical stability
delay = 29, error ratio = 0.16146286799221407
decorrelated matrix has less rows that input data for numerical stability
delay = 30, error ratio = 0.16631679381724584
```

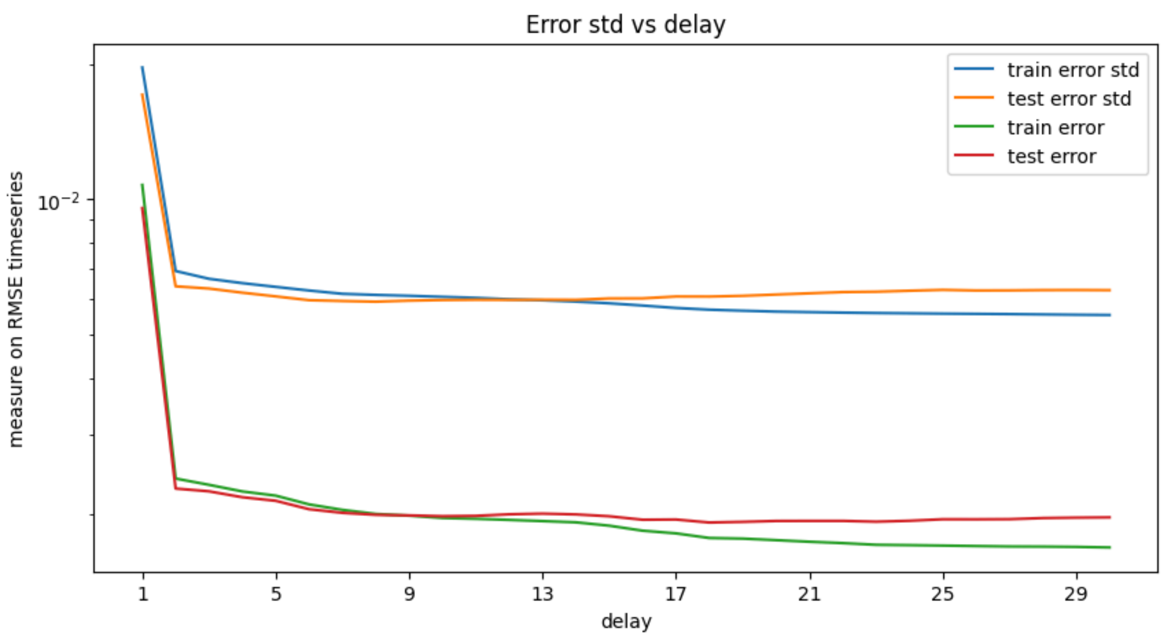
We see that the train test ratio decreases with increasing delay until a certain delay value is reached, then increases again. Past a certain point the decorrelation function produces the warning that the smallest singular values of the correlation matrix violate the numerical stability heuristic, indicating that additional delays do not introduce further information.

3.4

Visualising how the standard deviation and mean of the error vary with delay

In [104…

```python
fig, ax = plt.subplots(1, 1, figsize=(10, 5))
ax.plot(delays, train_stds, label='train error std')
ax.plot(delays, test_stds, label='test error std')
ax.plot(delays, train_means, label='train error')
ax.plot(delays, test_means, label='test error')
ax.legend()
plt.title('Error std vs delay')
ax.set_yscale('log')
ax.set_xlabel('delay')
ax.set_ylabel('measure on RMSE timeseries')
ax.set_xticks(delays[::4])
plt.show()
```



We can see that there is a sharp drop associated with increasing the delay to 2 (delay of 1 corresponds to simply fitting a linear model to the current output, to predict the next output). There is then a very slight improvement of the mean train and test RMSE until the optimum delay is reached, at which point the training accuracy continues to decrease whilst the test accuracy remains the same. This train test mismatch indicates the point at which slight overfitting occurs.

High values of standard deviation in the train and test RMSE would indicate that the distribution of error is wide, meaning many points in the prediction actually have an error greater than the mean error. here we see that the standard deviation is $\approx \pm 0.006$ RMSE. Since the optimum mean is ≈ 0.002, this indicates that the error probably lies within the same magnitude order of magnitude everywhere (we can't be more specific without looking at the distribution of error).

Problem 3

3.1

To alter the rank of the SVD used to decorrelate the delay embedding matrix, we can take a slice of the left hand side SVD vector up to the rank we want.

$$\hat{X} = U[:, : rank]^T X$$

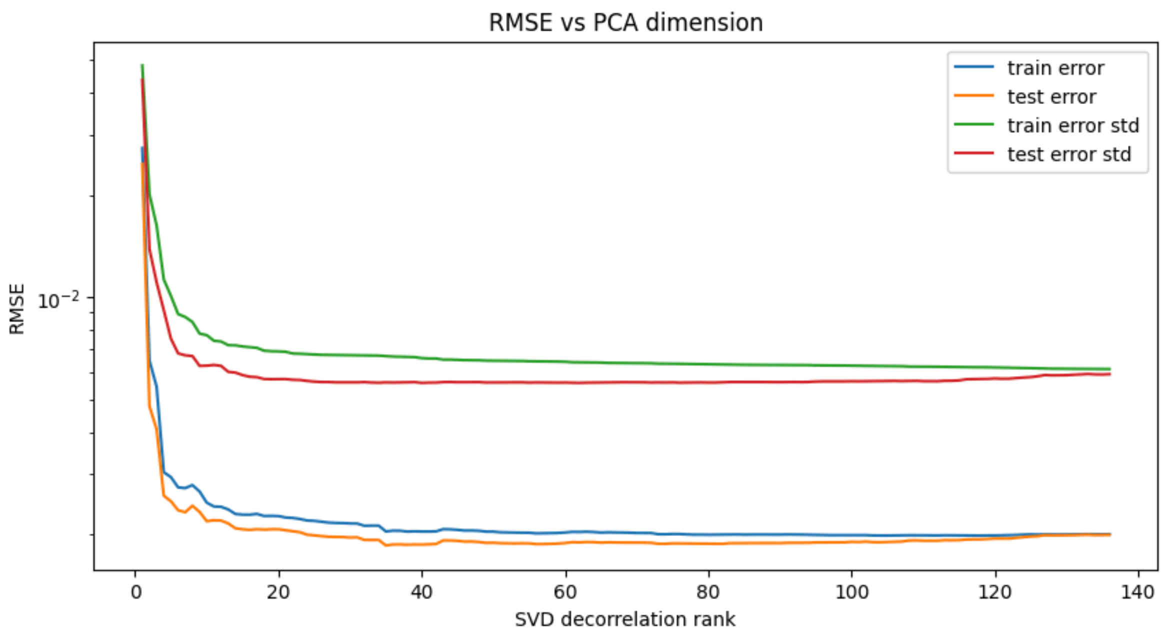
This has the effect of keeping the decorrelated state surrogate rows with the largest singular values, corresponding to those which carry the most information.

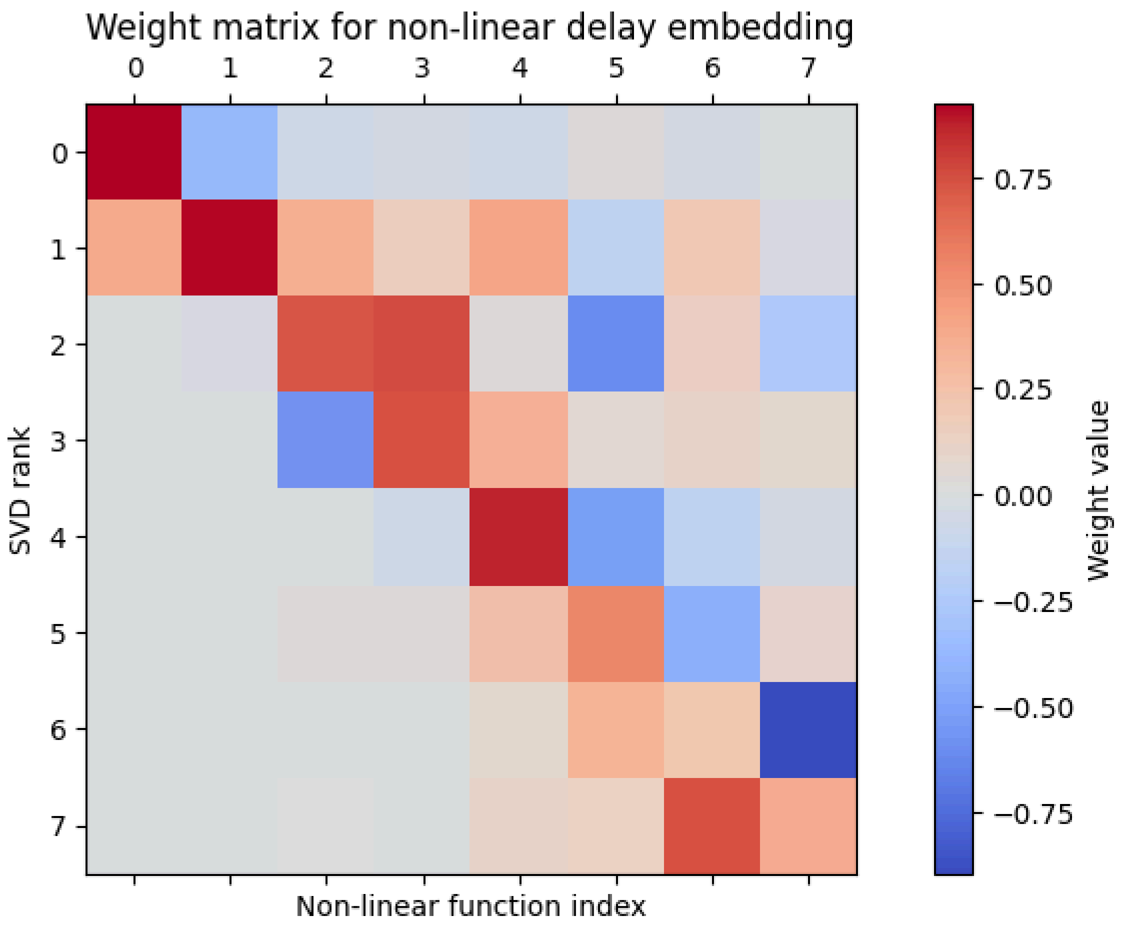
In [104…

```
def SVD_linear_test(delay: int):
    dimensions = list(range(1, train_data[0].shape[0]*delay+1))
    train_rmses = []
    test_rmses = []
    train_stds = []
    test_stds = []

    for dim in dimensions:
        W, U, _, _ = linear_delay_embedding(stacked_train_data, delay=delay, kap

        _, train_rmse, train_std = predict(W, U, stacked_train_data, delay=delay
        _, test_rmse, test_std = predict(W, U, stacked_test_data, delay=delay)

        train_rmses.append(train_rmse)
        test_rmses.append(test_rmse)
        train_stds.append(train_std)
        test_stds.append(test_std)

        if dim == delay:
            W_view = W

    # plotting errors vs PCA dimension
    fig, ax = plt.subplots(1, 1, figsize=(10, 5))
    ax.plot(dimensions, train_rmses, label='train error')
    ax.plot(dimensions, test_rmses, label='test error')
    ax.plot(dimensions, train_stds, label='train error std')
    ax.plot(dimensions, test_stds, label='test error std')
    ax.legend()
    plt.title('RMSE vs PCA dimension')
    ax.set_xlabel('SVD decorrelation rank')
    ax.set_ylabel('RMSE')
    ax.set_yscale('log')
    plt.show()

    # visualising weight matrix
    fig, ax = plt.subplots(1, 1, figsize=(10, 5))
    mat = ax.matshow(W_view, cmap='coolwarm')
    plt.colorbar(mat, label='Weight value')
    plt.title('Weight matrix for non-linear delay embedding')
    plt.ylabel('SVD rank')
    plt.xlabel('Non-linear function index')
    plt.show()
```

In [104…

```
SVD_linear_test(delay=8)
```





For the linear model we see that the train and test performance stay roughly constant until the SVD rank is reduced below 20. This implies that rows of the full decorrelated matrix beyond rank 20 carry very little information about the system. Since the data is already normalised and centered we are effectively conducting a PCA dimensionality reduction of the original delay embedding matrix, which preserves the variance of the original matrix as much as possible in the first principle components.

We can see the learned weight matrix corresponding to the rank 8 SVD trail, which we will return to for the nonlinear models, to validate that the weights corresponding to nonlinear terms are being learned.

3.2

To Fit a quadratic and polynomial model we will have to use a library of nonlinear functions for training, so we must implement this first. If we define a library of nonlinear functions, Φ then the fitting precedure is loosely

$$\tilde{X}_{\text{prev}} = \Phi(\hat{X}_{\text{prev}})$$

$$W = \hat{X}_{\text{next}} \tilde{X}^T_{\text{prev}} \left(\tilde{X}_{\text{prev}} \tilde{X}^T_{\text{prev}} \right)^+$$

$$\hat{x}_{k+1} = W \Phi(\hat{x}_k)$$

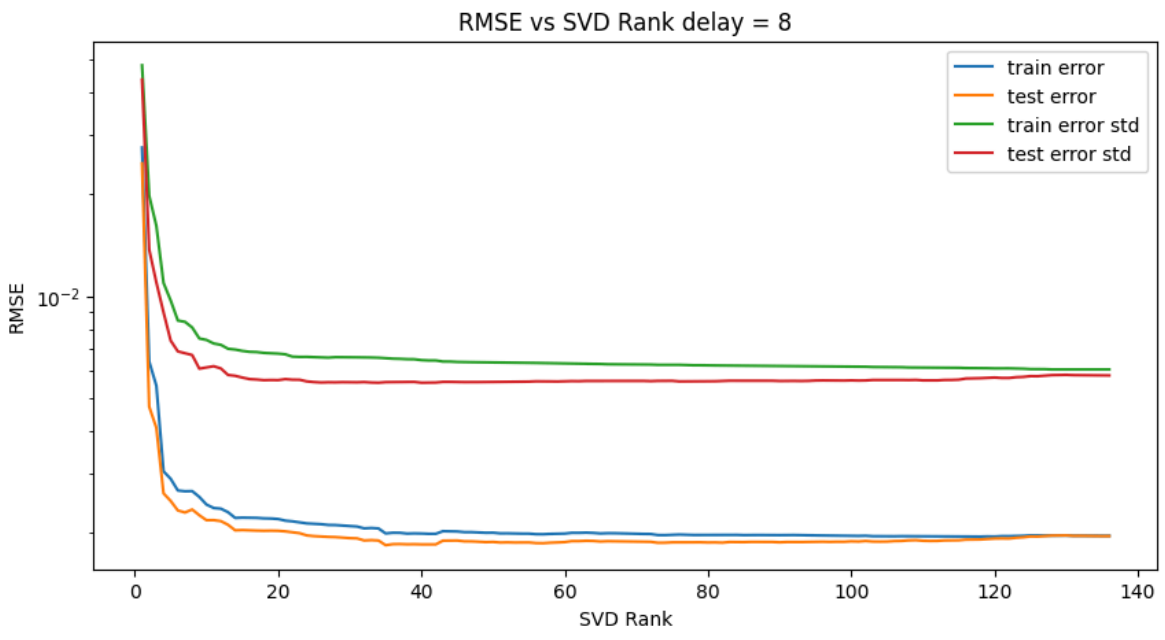
where hats denote decorrelated data and capital X terms are in hankel matrix form

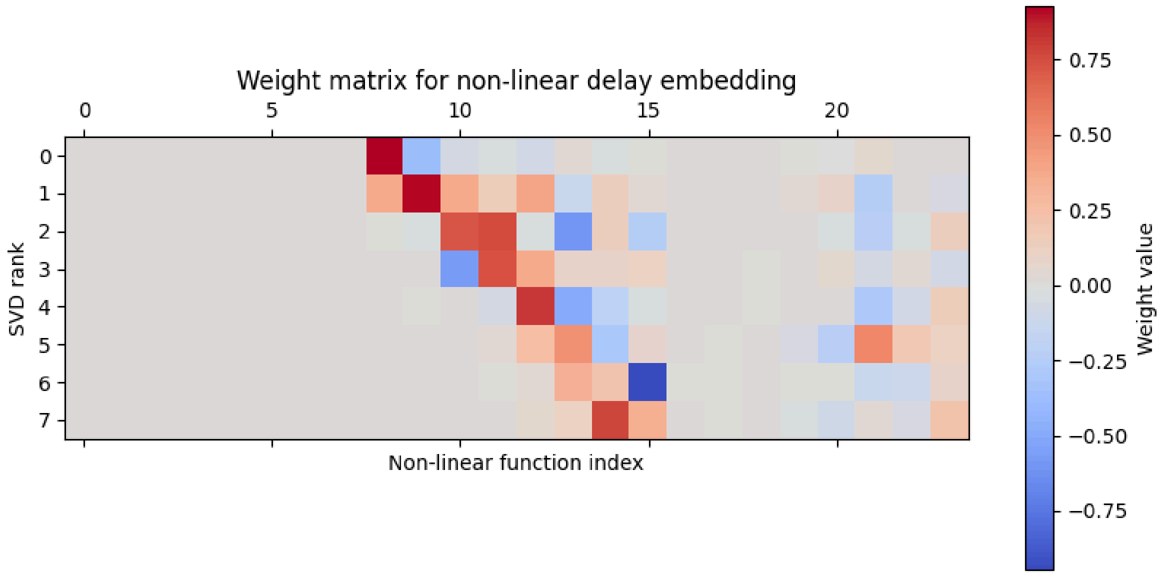
In [106…

```
def nonlinear_delay_embedding(data: np.array, func_library: List[Callable], dela
    """
    Fits a non-linear delay embedding model to predict the next state surrogate
    of the previous state surrogate, where the list of applied nonlinear functio

    args:
    -----
    data: np.array, timeseries of shape (n, m) where m is the number of timestep
    func_library: List[Callable], list of functions to apply to the previous sta
    delay: int, delay embedding window size
    kappa: float, threshold for decorrelation
    svd_rank: int, number of singular values to keep for decorrelation, if None,

    returns:
    --------
    W: np.array, non-linear model weights of shape (n, funcs*n)
    U: np.array, LHS Matrix from SVD, for switching between decorrelated state s
    """
    # delay embedding the observed outputs
    hankel = hankel_matrix(data, delay)
    # previous timestep and next timestep state surrogate matrices
    X_prev = hankel[:, :-1]
    X_next = hankel[:, 1:]
    # decorrelated input state surrogate matrix
    _, U = decorrelate(hankel, kappa=kappa, verbose=False)
    # truncating the left singular vectors to svd_order
    U = U[:, :svd_rank]
    # decorrelated state surrogates
    X_prev = U.T @ X_prev
    X_next = U.T @ X_next

    # applying the library of functions to the previous state surrogates
    func_evals = []
    for func in func_library:
        func_eval = func(X_prev)
        func_evals.append(func_eval)

    nl_X_prev = np.concatenate(func_evals, axis=0) # result is (funcs*n, obs-del

    # fitting a linear model to the new nonlinear state space
    W = (X_next @ nl_X_prev.T) @ la.pinv(nl_X_prev @ nl_X_prev.T, rcond=1e-8)

    return W, U

def predict_nonlinear(W: np.array, U: np.array, data: np.array, func_library: Li
    """
    predicts the output state at the next timestep using a non-linear delay embe
    with learned weights W, delay d, decorrelation matrix U and a library of non
    given by func_library.

    args:
    -----
    W: np.array, linear model weights of shape (n, funcs*n)
    U: np.array, LHS Matrix from SVD, for decorrelation
    data: np.array, test trajectory of shape (n,m) where m is the number of time
    d: int, delay embedding window size

    returns:
    --------
    Y_pred: np.array, predicted output trajectory
    prediction_rmse: float, root mean squared error of the prediction vs the act
    """
    # delay embedding matrix
    hankel = hankel_matrix(data, delay)
    # decorrelated previous state surrogate matrix
    X_prev = U.T @ hankel[:, :-1]

    # evaluating data through the library of functions
    func_evals = []
    for func in func_library:
        func_eval = func(X_prev)
        func_evals.append(func_eval)

    nl_X_prev = np.concatenate(func_evals, axis=0) # result is (funcs*n, obs-del

    # predicting the next state surrogate
    X_next_pred = W @ nl_X_prev

    # predicting the output at unseen timesteps
    Y_pred = U[-data.shape[0]:, :] @ X_next_pred
    Y_actual = data[:, delay:]

    # calculating the rmse for each timestep over the 17 bolts
    rmse = la.norm(Y_pred - Y_actual, axis=0) / np.sqrt(Y_actual.shape[0]) # sha

    # mean error is then taken over all timesteps
    mean_error = np.mean(rmse)
    std_error = np.std(rmse)

    return Y_pred, mean_error, std_error

def SVD_non_linear_test(func_library: List[Callable], delay: int):
    dimensions = list(range(1, train_data[0].shape[0]*delay+1))

    train_rmses = []
    test_rmses = []
    train_stds = []
    test_stds = []

    for dim in dimensions:
        W, U = nonlinear_delay_embedding(stacked_train_data, func_library, delay

        _, train_rmse, train_std = predict_nonlinear(W, U, stacked_train_data, f
        _, test_rmse, test_std = predict_nonlinear(W, U, stacked_test_data, func

        train_rmses.append(train_rmse)
        test_rmses.append(test_rmse)
        train_stds.append(train_std)
        test_stds.append(test_std)
        # saving weight matrix for delay == rank
        if dim == delay:
            W_view = W

    # plotting errors vs SVD rank
    fig, ax = plt.subplots(1, 1, figsize=(10, 5))
    ax.plot(dimensions, train_rmses, label='train error')
    ax.plot(dimensions, test_rmses, label='test error')
    ax.plot(dimensions, train_stds, label='train error std')
    ax.plot(dimensions, test_stds, label='test error std')
    ax.legend()
    ax.set_title('RMSE vs SVD Rank delay = {}'.format(delay))
    ax.set_xlabel('SVD Rank')
    ax.set_ylabel('RMSE')
    ax.set_yscale('log')
    plt.show()

    # visualising weight matrix
    fig, ax = plt.subplots(1, 1, figsize=(10, 5))
    mat = ax.matshow(W_view, cmap='coolwarm')
    plt.colorbar(mat, label='Weight value')
    plt.title('Weight matrix for non-linear delay embedding')
    plt.ylabel('SVD rank')
    plt.xlabel('Non-linear function index')
    plt.show()
```

Now let's train and test a quadratic polynomial model with function library [lambda x: np.ones_like(x), lambda x: x, lambda x: x**2]

In [106…
```
SVD_non_linear_test([lambda x: np.ones_like(x), lambda x: x, lambda x: x**2], de
```



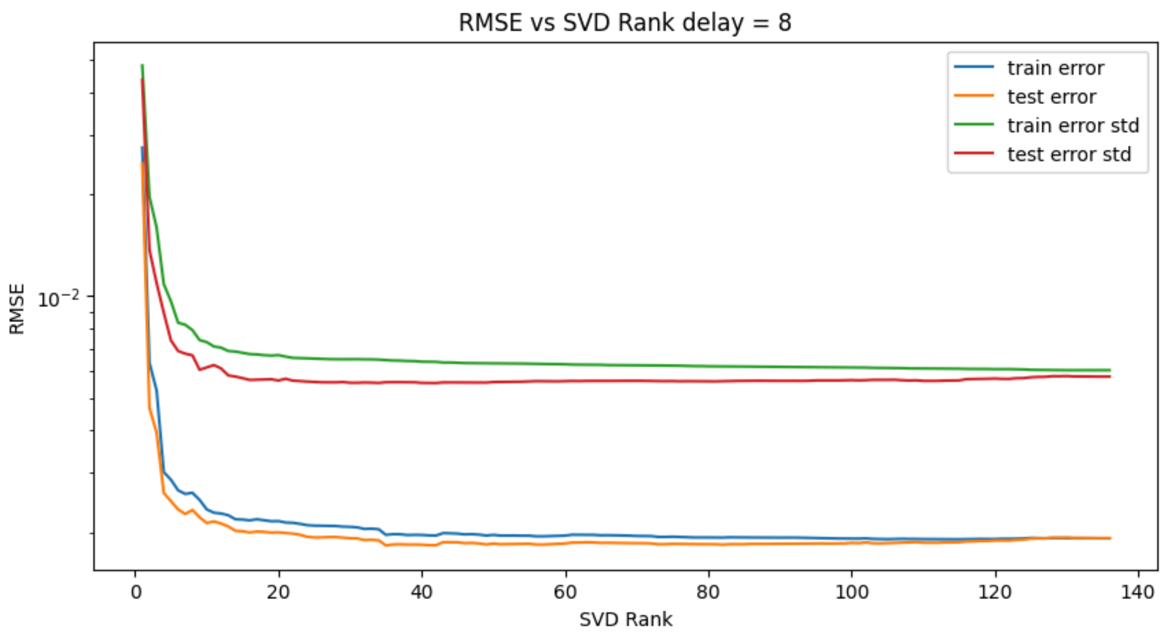


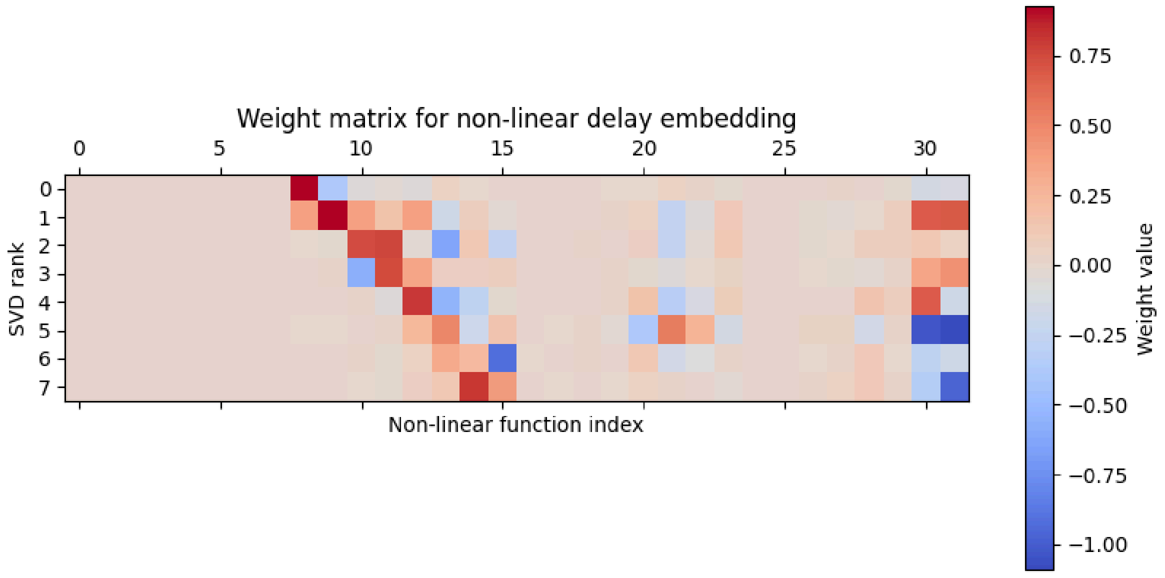
Fitting the data to the quadratic model, we see roughly equivalent performance on the test and training data. We can visually verify from the function library weight matrix that the central columns corresponding to linear mappings is dominant, although some quadratic weights have been learned.

3.3

training and testing a cubic polynomial model

```
In [106…  SVD_non_linear_test([lambda x: np.ones_like(x), lambda x: x, lambda x: x**2, lam
```





Again, training the cubic polynomial model we see very similar train and test accuracy and again we can verify that the linear weights are dominant. Although again, some cubic and quadratic weights have been learned. The correspondence in accuracy between the linear, quadratic and cubic models, suggests that a linear model is all that is necissary for this dataset.

3.4

We saw in all three accuracy plots that above a certain SVD rank, the test loss actually starts to increase. This suggests that keeping the decorrelated matrix rows corresponding to low SVD singular values, actually makes the model more overfit to the training data. This makes sense, since the small singular values indicate that the corresponding decorrelated delay embedding acounts for a small amount of the observed variance and could be simply an anomoly in the training data.

4.1

We can calculate the DMD of a linear system as a special case given

$$\hat{x}_{k+1} = W\hat{x}_k$$

where W is our learned linear model and $\hat{x}_k$ is the state surrogate at timestep k. Then koopman modes are

$$h(\hat{x}_{k+1}) = h(W\hat{x}_k) = \lambda h(\hat{x}_k)$$

and if we assume that the koopman operator is given by

$$h(\hat{x}) = v^T\hat{x}$$

where $v^T\hat{x}$ is the eigenfunction of a particular koopman mode, we then have

$$W^T v = \lambda v^T$$

Koopman modes include the eigenvalue, eigenfunction pairs of W (corresponding to left eigenvectors of W).

Because the koopman operator is linear, the rest of the koopman modes can be determined from the unique products of these solutions. For a linear model there are a finite number of koopman modes provided none of the eigenvalues are equal to one.

In [105…

```
def linear_DMD(W):
    """
    return the eigenvalues and DMD matrix corresponding to the approximate koopm
    a linear model, with square weight matrix W.

    Returns:
    --------
    W_eigvals: np.array, eigenvalues of the DMD matrix, sorted by decreasing mag
    dmd_mat: np.array, DMD matrix each row of which corresponds to a koopman fun
    """

    # we want the left eigenvectors, so we need to transpose W. W is of course s
    W_eigvals, W_eigvecs = la.eig(W.T)
    # sorting the eigenvectors by the magnitude of the eigenvalues
    sort_indices = np.argsort(np.abs(W_eigvals))[::-1]
    W_eigvals = W_eigvals[sort_indices]
    W_eigvecs = W_eigvecs[:, sort_indices]

    # The first order DMD matrix is simply
    dmd_mat = W_eigvecs.T

    return W_eigvals, dmd_mat
```

In [105…

```
# let's just retrain the model to get the weights
W, U, X_prev, X_next = linear_delay_embedding(stacked_train_data, delay=8, kappa

W_eigvals, dmd_mat = linear_DMD(W)
```

4.2

The residual of a given DMD mode, measures how close we are to an actual koopman mode. It is given by

$$res(\lambda, h) = \frac{||\text{\Kappa}h - \lambda h||^2}{||h||^2}$$

where κh is the actual koopman operator and λh is the DMD approximation. To approximate this residual with data, we can use the equation

$$res(\lambda, h) \approx \frac{v^* \left(L - \lambda H - \bar{\lambda} H^T + |\lambda|^2 G\right) v}{v^* G v}$$

where $G = XX^T$, $H = YX^T$ and $L = YY^T$

and in this case, since we have a linear model, $Y = \hat{X}_{\text{prev}}$, $X = \hat{X}_{\text{prev}}$, with $Y = WX$

More precisely, the residual is the relative error between the approximate koopman mode and actual koopman mode.

In [105…

```python
## 4.2
def koopman_residue(X, Y, eigen_val, eigen_vec):
    """
    Calculates the approximate residual of the Koopman mode, corresponding to ei
    using the input and output state data X and Y.
    args:
    -----
    X: np.array, input state matrix of shape (n, m) where n is the number of fea
    Y: np.array, output state matrix of shape (n, m) where n is the number of fe
    koop_val: float, value of the koopman operator
    eigen_vec: np.array, eigenvector of the Koopman operator

    returns:
    --------
    residue: float, residue of the Koopman operator
    """

    G = X @ X.T
    H = Y @ X.T
    L = Y @ Y.T

    v = eigen_vec
    lam = eigen_val

    # shorthand for the hermitian transpose
    def HT(v: np.array) -> np.array:
        return np.conj(v).T

    res = HT(v) @ (L - lam*H - np.conj(lam)*HT(H) + np.abs(lam)**2*G) @ v
    res /= HT(v) @ G @ v

    # Because of machine precision, the residue can be a complex number with a v
    assert np.isclose(np.imag(res), 0, atol=1e-15), 'Residue is not real to mach
    res = np.real(res)

    return res
```

lets calculate the residual of each of the modes

```
In [105…  def sorted_koopman_residues(W_eigvals, dmd_mat, X_prev, X_next, return_inds=Fals
              no_modes = W_eigvals.shape[-1]

              # getting the residue of each approximate koopman mode
              residues = []
              for i in range(no_modes):
                  residue = koopman_residue(X_prev, X_next, W_eigvals[i], dmd_mat[i, :])
                  residues.append(residue)
              residues = np.array(residues)

              # sorting the residues and the corresponding eigenvalues and eigenvectors by
              sort_inds = np.argsort(residues)
              residues = residues[sort_inds]
              W_eigvals = W_eigvals[sort_inds]
              dmd_mat = dmd_mat[sort_inds, :]

              # returning the sorted residues, eigenvalues and dmd matrix (eigenvectors)
              if return_inds:
                  return residues, W_eigvals, dmd_mat, sort_inds

              return residues, W_eigvals, dmd_mat

          residues, W_eigvals, dmd_mat = sorted_koopman_residues(W_eigvals, dmd_mat, X_pre

          print("residues: ", residues)
          print("number of modes: ", len(residues))
```

```
residues:  [0.00905937 0.00905937 0.03085111 0.03085111 0.10878054 0.1144617
 0.1144617  0.11931922 0.13703426 0.13703426 0.18379061 0.18379061
 0.23132825 0.23132825 0.23778637 0.23778637 0.24264559 0.24264559
 0.2677891  0.2677891  0.28629178 0.28629178 0.29401171 0.29955253
 0.29955253 0.3035216  0.3035216  0.31397749 0.31397749 0.31522977
 0.31522977 0.3210113  0.32235862 0.32235862 0.33328547 0.33328547
 0.34322404 0.34322404 0.35404491 0.35404491 0.35431745 0.35431745
 0.35954411 0.35954411 0.36707913 0.36707913 0.37205635 0.37205635
 0.38475654 0.38475654 0.38532846 0.38532846 0.39629468 0.39835012
 0.39835012 0.40051784 0.40051784 0.40685699 0.40685699 0.40968068
 0.40968068 0.41362921 0.41362921 0.4145471  0.4145471  0.41835866
 0.41835866 0.42156455 0.42156455 0.42419559 0.42419559 0.43669519
 0.43669519 0.43953118 0.43953118 0.44431105 0.44431105 0.44737213
 0.44737213 0.45163011 0.45163011 0.45678756 0.45678756 0.4646656
 0.4646656  0.49613509 0.49613509 0.50103377 0.50103377 0.51978814
 0.51978814 0.52080184 0.52080184 0.52096519 0.52096519 0.52223278
 0.52229709 0.52229709 0.52721917 0.52721917 0.52734955 0.52734955
 0.53597776 0.53597776 0.53929586 0.53929586 0.54231856 0.54231856
 0.54244766 0.54244766 0.54337239 0.54337239 0.56115995 0.56115995
 0.57067719 0.57260547 0.57260547 0.57820308 0.58428874 0.58428874
 0.58514436 0.58514436 0.65216634 0.65216634 0.65506993 0.65506993
 0.67946238 0.67946238 0.78212164 0.86363979 0.86363979 0.87926704
 0.87926704 0.90776548 0.95473893 0.97895318]
number of modes:  136
```

4.3

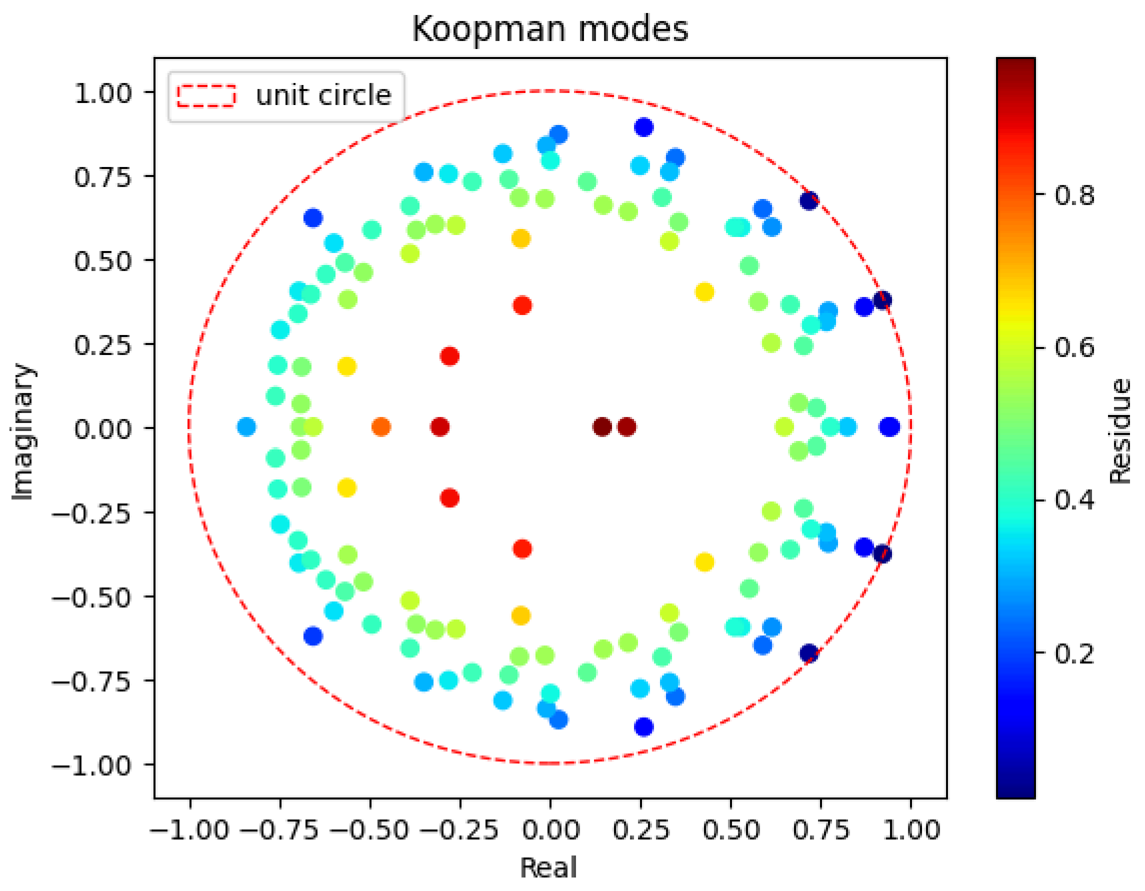
We have 136 modes because we are using a delay of 8 for our linear delay embedding model (17*8). We see that the worst residual is close to 1, which would indicate the approximate mode is off by a whole factor of the true solution. However, this might be the case if the eigenvalue of the mode is very close to zero. Let's take a look by plotting the eigenvalues in the complex plane, colorcoded by residue.

In [105…

```python
def plot_koopman_modes(eigvals, residues):
    """
    Plots the first n_modes Koopman modes, corresponding to the eigenvalues eigv
    args:
    -----
    eigvals: np.array, eigenvalues of the Koopman operator
    eigvecs: np.array, eigenvectors of the Koopman operator
    n_modes: int, number of modes to plot
    """
    fig, ax = plt.subplots()
    # seperating real and imaginary parts for plotting
    eigvals_x = np.real(eigvals)
    eigvals_y = np.imag(eigvals)
    scat = ax.scatter(eigvals_x, eigvals_y, c=residues, cmap='jet')
    # plot unit circle
    unit_circle = plt.Circle((0, 0), 1, color='r', fill=False, linestyle='--', l
    ax.add_artist(unit_circle)

    # formatting
    ax.set_xlabel('Real')
    ax.set_ylabel('Imaginary')
    ax.set_title('Koopman modes')
    plt.colorbar(scat, label='Residue')
    # setting limits so that the unit circle is central and visible
    ax.set_xlim(-1.1, 1.1)
    ax.set_ylim(-1.1, 1.1)
    ax.legend(loc='upper left')

    plt.show()
```

In [105…

```python
plot_koopman_modes(W_eigvals, residues)
```



We can see that our theory of smaller eigenvalues producing larger residuals is correct. Now when it comes to plotting the frequencies, that each koopman mode corresponds to, we must first consider what this means. We saw previously, that under the koopman operator, our linear observable system follows the update rule

$$h(\hat{x}_{k+1}) = \lambda h(\hat{x}_k)$$

meaning, given some timestep Δt and initial output, producing the surrogate state $\hat{x}_0$, we have the power rule

$$h(\hat{x}(t)) = \lambda^{t/\Delta t} h(\hat{x}_0)$$

which in exponential form is

$$h(\hat{x}(t)) = e^{\ln(\lambda)(t/\Delta t)} h(\hat{x}_0)$$

This can be rewritten as a decaying complex exponential, since $\ln(\lambda)$ is complex

$$h(\hat{x}(t)) = f(t)e^{i\theta t}$$

where

$$f(t) = h(\hat{x}_0)e^{\frac{1}{\Delta t}\text{Re}\{\ln(\lambda)\}t}$$

and

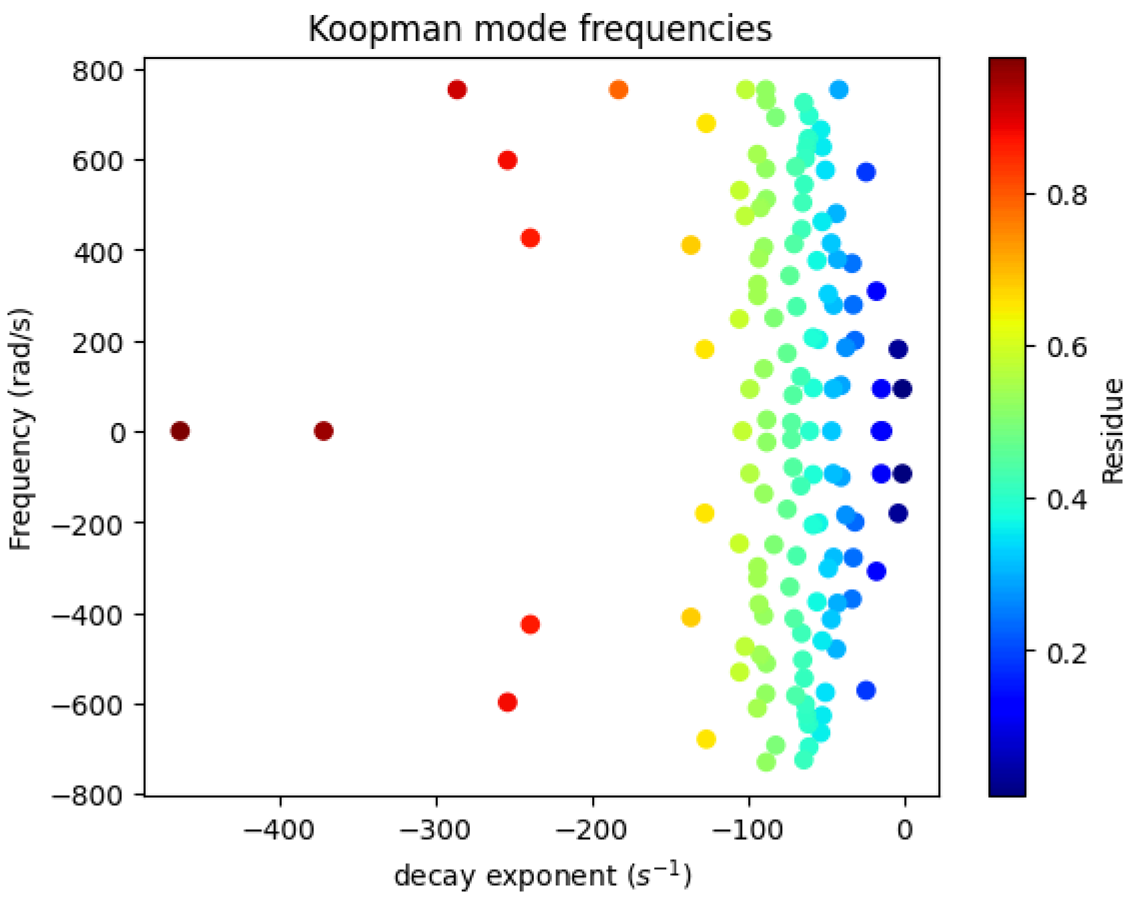
$$\theta = \frac{1}{\Delta t}\text{Im}\{\ln(\lambda)\}$$

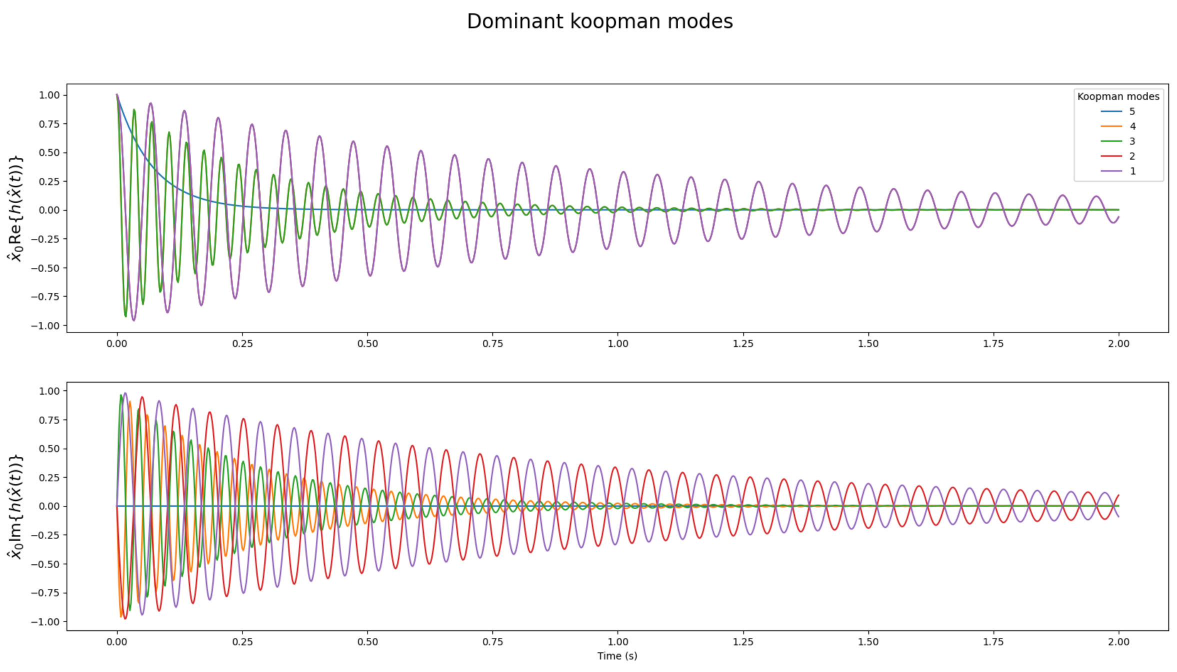
A really cool implication of this, is that the full koopman space solution is a fourier series solution to the underlying system, since each koopman mode defines a decaying (since all the koopman eigenvalues lie in the complex unit circle) sinusoidal function. In any case, the frequency of the koopman mode corresponding to h is given by θ (rad/s). The decay exponent of each koopman mode is also given by $\frac{1}{\Delta t}\mathrm{Re}\{\ln(\lambda)\}\ s^{-1}$

Let's plot this and we can go one step further and plot the sinsusoids corresponding to each mode.

In [105…

```
def koopman_space_solution(eig_val, t, delta_t):
    """
    Calculates the koopman space state solution at time t, where ther koopman op
    args:
    -----
    eig_val: float, eigenvalue of the Koopman operator
    t: float, time to evaluate the solution
    x_0: np.array, initial condition of the system

    returns:
    --------
    x_t: np.array, solution to the Koopman operator at time t
    """
    koop_xt = np.exp(1/delta_t * np.real(np.log(eig_val)) * t)*np.exp(1j/delta_t
    return koop_xt


def plot_koopman_mode_frequencies(eig_vals, residuals, delta_t):
    """
    Plots the frequency of the Koopman modes, corresponding to the eigenvalues e
    args:
    -----
    eig_vals: np.array, eigenvalues of the Koopman operator
    residuals: np.array, residues of the Koopman operator
    delta_t: float, time step between each state in the data
    """
    # calculating the frequencies of the Koopman modes
    freqs = 1/delta_t * np.imag(np.log(eig_vals))
    # calculating the decay exponents of the Koopman modes
    decay_expo = 1/delta_t * np.real(np.log(eig_vals))

    fig, ax = plt.subplots()
    scat = ax.scatter(decay_expo, freqs, c=residuals, cmap='jet')
    ax.set_xlabel('decay exponent ($s^{-1}$)')
    ax.set_ylabel('Frequency (rad/s)')
    ax.set_title('Koopman mode frequencies')
    plt.colorbar(scat, label='Residue')
    plt.show()

    slowest_decaying = np.argsort(decay_expo)[-5:]

    fig, axs = plt.subplots(2, 1, figsize=(20, 10))
    t = np.linspace(0, 2, 1000)
    for i in slowest_decaying:
        koop_sol = koopman_space_solution(eig_vals[i], t, delta_t)
        axs[0].plot(t, np.real(koop_sol), label=f'{i+1}')
        axs[1].plot(t, np.imag(koop_sol), label=f'{i+1}')
    axs[0].legend(title='Koopman modes')
    axs[0].set_ylabel(r'$\hat{x}_0\mathrm{Re}\{h(\hat{x}(t))\}$', fontsize=16)
    axs[1].set_ylabel(r'$\hat{x}_0\mathrm{Im}\{h(\hat{x}(t))\}$', fontsize=16)
    axs[1].set_xlabel('Time (s)')

    plt.suptitle('Dominant koopman modes', fontsize=20)
    plt.show()

# we can determine delta_t from the framerate of the camera
plot_koopman_mode_frequencies(W_eigvals, residues, 1/FRAME_RATE)
```





The top plot is a really powerful visualisation, as it shows us that our innacurate koopman modes correspond to system dynamics that very quickly decay to zero and the more accurate modes are more persistent in the system dynamics. We also see that there is a conjugate symmetry in the modes, corresponding to orthogonal oscillations.

The bottom plot shows the real and imaginary dimensionless response of the top 5 dominant koopman modes in koopman space. We can see that that modes 1 and 2, together with 3 and 4 are conjugate modes and that mode 5 is actually non oscillatory, instead exhibiting a real exponential decay and 0 imaginary part. when transforming from the koopman space to the surrogate state space, these modes will be linearly combined according to the eigenfunctions of each mode to produce the dynamics of each of the bolts.

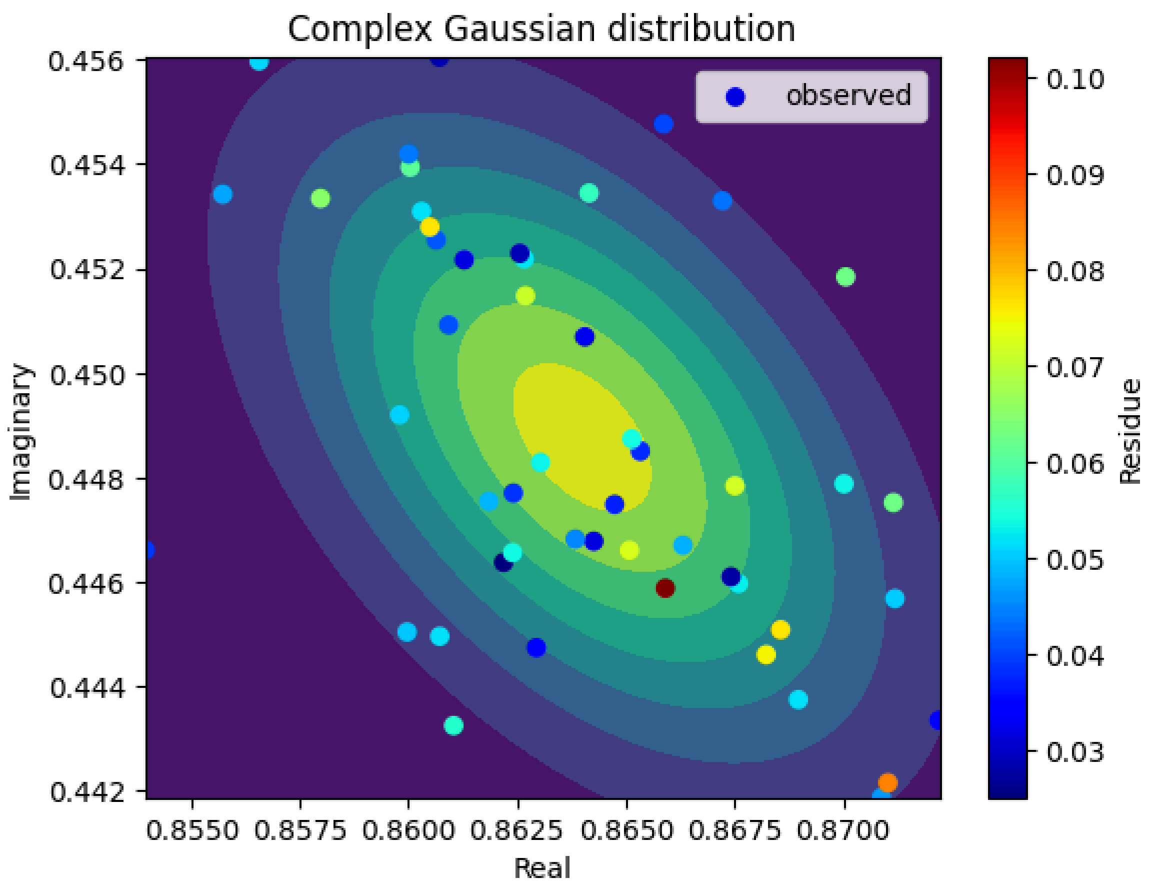
4.4

In [ ]:
```python
# to implement bagging, we want to repeatedly fit the model to different ordered

def dmd_bagging(trails=50):
    """
    Finds the eigenvalue corresponding to the koopman mode with smallest residue
    using bagging. The returned result can be used to quantify the uncertainty i

    returns:
    --------
    eigvals: np.array, eigenvalues from each trial corresponding to the smallest
    residuals: np.array, residuals of the eigenvalues from each trial
    mean_eigval: np.array, 2 entry array with the mean of the eigenvalues in the
    cov_eigval: np.array, 2x2 array, covariance matrix of the eigenvalues in the
    """

    eigvals = []
    residuals = []
    reference_eigval = None
    for trial in tqdm(range(trails), desc='Bagging trials'):
        # randomly sampling 80% of the training data and sorting, so that there
        inds = np.random.choice(stacked_train_data.shape[1], size=round(stacked_
        inds = np.sort(inds)

        # retraining the model on the bagged data
        W, U, X_prev, X_next = linear_delay_embedding(stacked_train_data[:,inds]
        W_eigvals, dmd_mat = linear_DMD(W) # solution is sorted by eigenvalue ma

        # calculating the residuals with all of the data is computationally expe
        residues, sorted_vals, sorted_dmd = sorted_koopman_residues(W_eigvals, d

        for eigen_val, residue in zip(sorted_vals, residues):
            if trial == 0: # setting the reference eigenvalue to the first eigen
                if not np.isclose(np.imag(eigen_val), 0, rtol=1e-2):
                    eigvals.append(eigen_val)
                    residuals.append(residue)
                    reference_eigval = eigen_val
                    break
            elif np.isclose(eigen_val, reference_eigval, rtol=1e-1): # ensuring
                eigvals.append(eigen_val)
                residuals.append(residue)
                break


    # eigenvals are complex so distribution is 2D, therefore we need to find the
    eigvals = np.array(eigvals)
    real_eigvals = np.real(eigvals)
    imag_eigvals = np.imag(eigvals)

    mean_eigval = [np.mean(real_eigvals), np.mean(imag_eigvals)]
    cov_eigval = np.cov(np.vstack([real_eigvals, imag_eigvals]))

    return eigvals, residuals, mean_eigval, cov_eigval


def visualise_complex_gaussian(obs, residuals, mean, cov):
    """
    Visualises a 2D complex gaussian distribution given by the mean and covarian
    args:
    -----
    obs: np.array, observed data points
    residuals: np.array, residuals of the observed data points
    mean: np.array, mean of the complex gaussian
    cov: np.array, covariance matrix of the complex gaussian
    """
    gaus_2d = multivariate_normal(mean=mean, cov=cov)
    real_range = (np.min(np.real(obs)), np.max(np.real(obs)))
    imag_range = (np.min(np.imag(obs)), np.max(np.imag(obs)))
    real, imag = np.meshgrid(np.linspace(*real_range, 100), np.linspace(*imag_ra

    # plotting
    fig, ax = plt.subplots()

    ax.contourf(real, imag, gaus_2d.pdf(np.dstack([real, imag])))
    scat = ax.scatter(np.real(obs), np.imag(obs), c=residuals, label='observed',
    ax.set_xlabel('Real')
    ax.set_ylabel('Imaginary')
    ax.set_title('Complex Gaussian distribution')
    ax.set_xlim(real_range)
    ax.set_ylim(imag_range)
    plt.colorbar(scat, label='Residue')
    plt.legend()
    plt.show()
```

In [107…
```
bagged_eigvals, residuals, mean, cov = dmd_bagging(trails=50)
print("mean eigenvalue: ", mean)
print("covariance of eigenvalues: ", cov)
```

```
Bagging trials: 100%|██████████| 50/50 [00:10<00:00,  4.91it/s]
mean eigenvalue:  [0.8639935809872348, 0.4487622246410752]
covariance of eigenvalues:  [[ 1.83693027e-05 -8.21051027e-06]
 [-8.21051027e-06  1.46027939e-05]]
```

In [107…
```
visualise_complex_gaussian(bagged_eigvals, residuals, mean, cov)
```



4.5

We know that the vibration frequency of a given koopman mode is proportional to the log of the imaginary component of the eigenvalue. More specifically

$$\theta = \frac{1}{\Delta t}\mathrm{Im}\{\ln(\lambda)\}$$

we can use the covariance matrix to get an estimate of the standard deviation in the imaginary axis, which quantifies our uncertainty. The corresponding uncertainty of the frequency θ can be calculated as follows (assuming all other variables are certain)

In [107…

```
im_sig = np.sqrt(cov[1,1])
im_mean = mean[1]
# 95% confidence interval is
interval_95 = norm.interval(0.95, loc=im_mean, scale=im_sig)
print(r"95% lambda confidence interval = ", interval_95)

theta_uncertainty = FRAME_RATE * (np.log(interval_95[1]) - np.log(mean[1]))
print(r"frequency 95% uncertainty =  +-", theta_uncertainty, "rad/s")
```

```
95% lambda confidence interval =  (0.44127249624336934, 0.456251953038781)
frequency 95% uncertainty =  +- 3.972481091769371 rad/s
```

So we can see qualitively from the diagram and quantitavely from the calculated 95% uncertainty that the uncertainty in the frequency is fairly low for these residues. Suggesting that this koopman mode is a good approximation.