2143062-DDPM-Part1

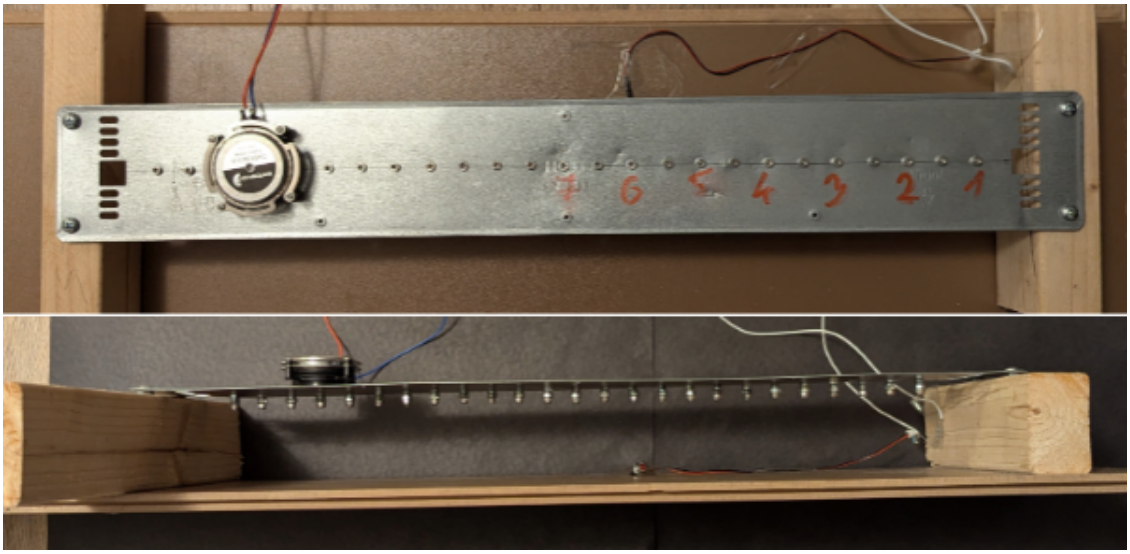
November 28, 2024

1 Data-Driven Physical Modelling (SEMTM0007) Coursework - Part 1

1.0.1 Wishawin Lertnawapan 2143062

1.1 Table of Contents

1.2 0. Problem Description



>(Figure 1.) Experimental rig. A soft steel plate with bolts attached. The vibration is measured using >a video camera with 720p resolution running at 240 frames per second. Vibration data was >extracted using digital image correlation (DIC).

System Identification:
System identification was performed to orient the nature of the data collected, as shown in Figure 1. [1]. The experimental data is derived from a series of impact vibration tests. This indicates that the nature of the system is non-autonomous, due to the time-dependent nature of the trajectories caused by the external inputs from the hammer impacts.

It is also noted to be discrete, limited by the temporal resolution of the camera. With a capture rate of 240 frames-per-second, this equates to a time interval of $k = \mathbf{4.17}$ milliseconds. Assuming a deterministic process, the dynamic system can be generalised as the discrete-time solution of a differential equation:

$$x_{k+1} = F(x_k)$$

for any trajectory x representing the vector space of (observable) states at time k, evolved by the function F.

For this experiment, x_k captures the vertical position of each of the 17 bolts such that for a single instance in time;

$$x_k = (x_1, x_2, \dots x_n)^T$$

where $n = 17$ corresponds to the number of "trajectories" or bolts tracked.

Additionally, it is given that the experiment was repeated 24 times to account for variations and reliability. Thus, the output of each can be represented in a single matrix X:

$$X = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1m} \\ x_{21} & x_{22} & \dots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{nm} \end{bmatrix}, \; X \in \mathbb{R}^{n \times m}$$

where $m = \frac{T}{k}$, the number of data points over a period of time T

To accurately perform any of the following steps of data-driven modelling, the data must first be preprocessed appropriately and normalized to ensure the model captures the dominant vibrations and features - while managing computational complexity. This leads to the following section.

Importantly, the presence of nonlinearities is apparent in the oscillatory system. Sources of damping can be inferred in the experimental setup: - The steel is stated to be soft, creating intrinsic material damping - The presence of the shorted transducer, inducing electrical damping By the nature of damping, the oscillation is expected to dissipate energy until the system reaches a steady state. in linear systems, an equation-based model of the time-domain response can be expressed as a sum of exponential sinusoids [2]:

$$x(t) = \sum_{n=1}^{N} A_n e^{-\zeta_n \omega_n t} \cos(\omega_d t + \phi_n)$$

where n is the number of modes, ω, ζ and ϕ are angular frequency, damping ratio, and phase respectively. However, in a nonlinear system with multiple sources of damping, this becomes harder to analytically predict.

This implies the governing function F is unlikely to be fully linear. The purpose of data-driven physical modelling is to approximate one through examining an appropriate approximation, which is then progressively validated and optimised. The following report approaches this system from two perspectives: Delay embedding and Dynamic Mode Modelling.

[1593]:
```
#Importing packages and libraries
import matplotlib.pyplot as plt
import seaborn as sns
```

```
import numpy as np
import numpy.linalg as la

import scipy.stats as stats
from scipy.stats import wilcoxon, norm
from scipy.stats import multivariate_normal
from scipy.integrate import solve_ivp
from scipy.linalg import expm
from scipy.signal import butter, filtfilt, hilbert, welch
from scipy.ndimage import gaussian_filter1d

from itertools import combinations_with_replacement

from sklearn.linear_model import Ridge, Lasso
from sklearn.metrics import r2_score
from sklearn.preprocessing import StandardScaler

import time
import random
```

1.3 1. Data Preprocessing

1.3.1 1.1 Initial Processing

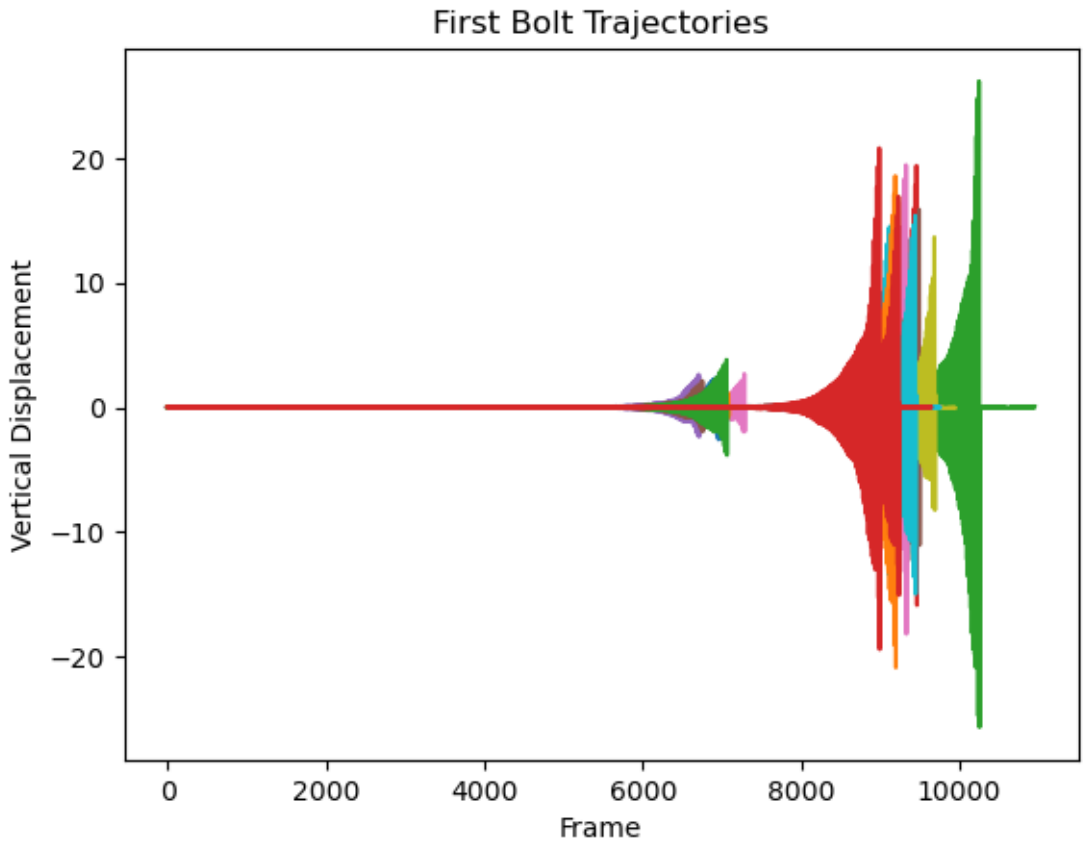
The following steps to preprocess data included:
1. Reversing to ensure chronological order 2. Removing the initial steady state 3. Eliminating decayed components 4. Normalisation and Scaling

The data was first loaded from the provided `.npz` file, and packaged into a list. Its dimensions and trajectory lengths were verified on a plot.

[875]:
```
#Loading aata
data_dict = np.load('AutonomousTrajectoriesBig.npz')
data_all = [it[1] for it in data_dict.items()]

print(f"Number of Experiments: {len(data_all)}")
for i in range(len(data_all)):
    print(f"Experiment {i+1}: {data_all[i].shape}")

# Visualise trajectories for first bolt
plt.figure()
for i in range(len(data_all)):
    plt.plot(data_all[i][0])
plt.title("First Bolt Trajectories")
plt.xlabel("Frame")
plt.ylabel("Vertical Displacement")
plt.show()
```

```
Number of Experiments: 24
```

```
Experiment 1: (17, 7453)
Experiment 2: (17, 9763)
Experiment 3: (17, 9853)
Experiment 4: (17, 9748)
Experiment 5: (17, 7423)
Experiment 6: (17, 9748)
Experiment 7: (17, 7378)
Experiment 8: (17, 9748)
Experiment 9: (17, 7363)
Experiment 10: (17, 9643)
Experiment 11: (17, 9253)
Experiment 12: (17, 9643)
Experiment 13: (17, 10948)
Experiment 14: (17, 9538)
Experiment 15: (17, 7363)
Experiment 16: (17, 7363)
Experiment 17: (17, 9748)
Experiment 18: (17, 9748)
Experiment 19: (17, 9943)
Experiment 20: (17, 9748)
Experiment 21: (17, 7348)
Experiment 22: (17, 7378)
Experiment 23: (17, 7348)
Experiment 24: (17, 9643)
```



Reversing `reverse()`

The initial data in matrices are noted to be time-reversed. Reversing was performed to ensure the sequence matches chronological progression, and that vibration states progress according to the dynamic system equation earlier. This is especially relevant for delay embedding and DMD methods which focus on the recovery of temporal relationships.

Removing Initial Steady States `truncate_intitial()`

Removing initial steady states was the process of aligning the instant of impact to the start index of the signal $t = 0$. This was done by identifying the index at which the maximum amplitude occurs. By truncating all signals to the instance of impact, the signals and especially phases of each vibration become invariant to the initial conditions.

[11]:
```python
# Reverse data function
def reverse(array):
    return array[::-1]

#Truncate initial conditions function
def truncate_initial(array):
    max_index = np.argmax(np.abs(array))
    truncated_array = array[max_index:]
```

```
    return truncated_array

#Truncate for before Initial Conditions (Noise Analysis) function
def truncate_noise(array):
    max_index = np.argmax(np.abs(array))
    pre_truncated_array = array[:max_index]
    return pre_truncated_array

#Truncate initial conditions
data_truncated_init = []
for i in range(len(data_all)):
    data_truncated_each = []
    for j in range(len(data_all[i])):
        data_truncated_each.append(truncate_initial(reverse(data_all[i][j])))
    data_truncated_init.append(data_truncated_each)
```

Eliminating Decayed Components

Eliminating the steady state refers to removing the segment of signal after it has settled into equilibrium - i.e. after the transient response has completely decayed. This step was performed to truncate the signal such that any temporal model will only be fitted to the portion of the system governed by F, and avoid artefacts in the embedding vector.

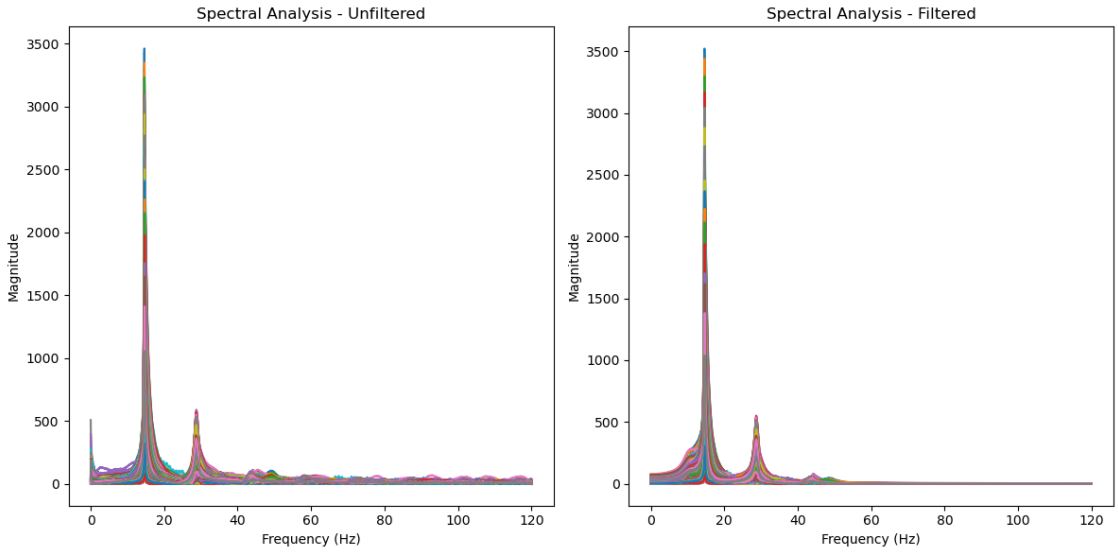
The Hilbert transform was applied to identify the position where vibration data decays into a steady state [3]. This was chosen as it consistently outperformed other truncation methods such as window gradient filtering and exponential flattening, likely as Hilbert analysis inherently tailors to frequency analysis and nonlinearity. However, the analytic signal is sensitive to high-frequency noise. To ensure no artefacts emerge in the amplitude envelope, a frequency analysis was performed to analyse the frequency spectrum for filtering.

`fft_analysis()` was used to perform a Fast Fourier Transform to identify vibration frequencies present in the signal. This was followed by using a Butterworth filter `bandpass_filter()` to isolate the dominant frequencies and remove high-frequency noise for all trajectories in the data set.

[16]:
```
#--------------------------#
high_f =50
low_f = 10
#--------------------------#

# FFT Spectral Analysis function
def fft_analysis(array,sampling_rate = 240):
    fft_values = np.fft.fft(array)
    freqs = np.fft.fftfreq(len(array), d=1/sampling_rate)
    freqs = frequencies[:len(freqs)//2]
    fft_values = np.abs(fft_values[:len(fft_values)//2])
    return fft_values, freqs

#Band pass filter function
def bandpass_filter(array, lowcut=low_f, highcut=high_f, fs=240, order=6):
```

```
    nyquist = 0.5 * fs
    low = lowcut / nyquist
    high = highcut / nyquist
    b, a = butter(order, [low, high], btype='band')
    filtered_data = filtfilt(b, a, array)
    return filtered_data
```

[2313]:
```
#Filter and truncate data
data_filtered = []
for i in range(len(data_truncated_init)):
    data_filtered_each = []
    for j in range(len(data_all[i])):
        data_filtered_each.append(bandpass_filter(data_truncated_init[i][j]))
    data_filtered.append(data_filtered_each)

fig, axes = plt.subplots(1, 2, figsize=(12, 6))

#Visualise spectrum of raw data
for i in data_truncated_init:
    for j in i:
        fft_values = np.abs(np.fft.fft(j))
        frequencies = np.fft.fftfreq(len(j), d=1/240)[:len(j)//2]
        axes[0].plot(frequencies, fft_values[:len(frequencies)])
axes[0].set_title("Spectral Analysis - Unfiltered")
axes[0].set_xlabel("Frequency (Hz)")
axes[0].set_ylabel("Magnitude")


# Visualise spectrum of truncated and filtered data
for i in data_filtered:
    for j in i:
        fft_values = np.abs(np.fft.fft(j))
        frequencies = np.fft.fftfreq(len(j), d=1/240)[:len(j)//2]
        axes[1].plot(frequencies, fft_values[:len(frequencies)])
axes[1].set_title("Spectral Analysis - Filtered")
axes[1].set_xlabel("Frequency (Hz)")
axes[1].set_ylabel("Magnitude")


plt.tight_layout()
plt.show()
```

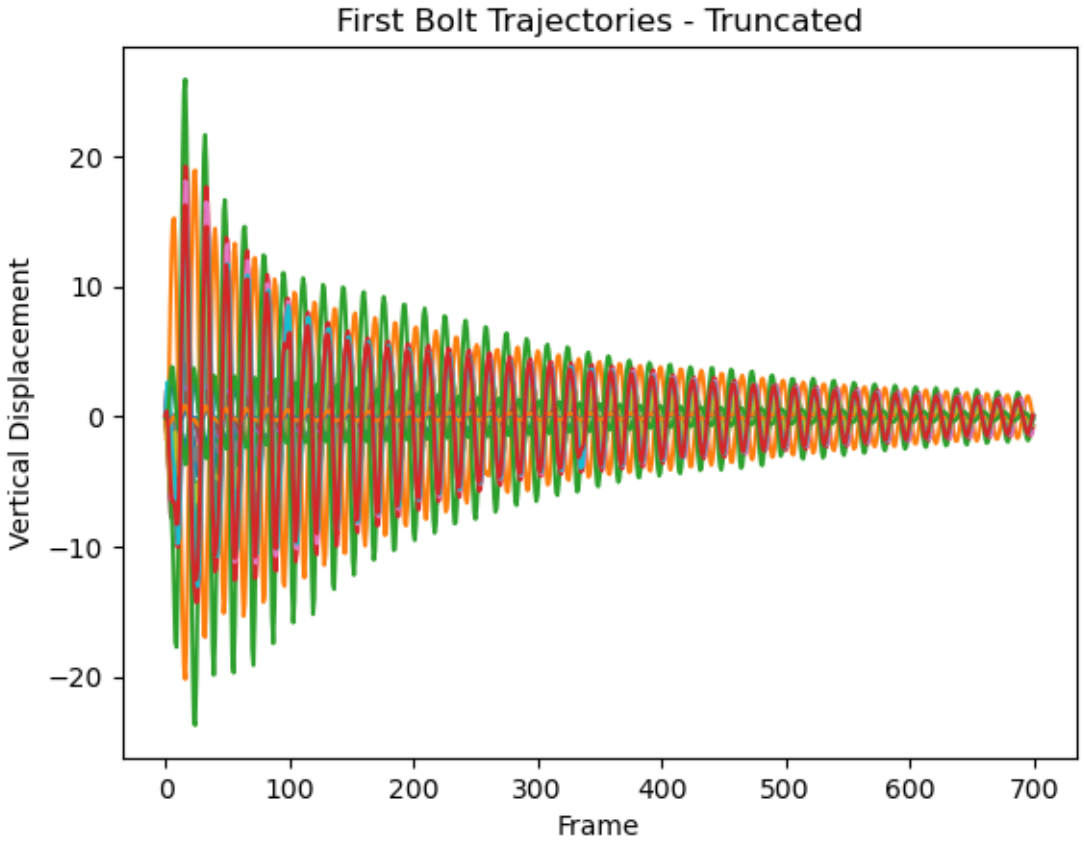


This signal was then passed into the `truncate_end()` function, which compares a sliding window of the experimental trajectory amplitude against the analytic signal envelope generated by the Hilbert transform. The following parameters were adjusted to ensure dynamic components were isolated. An arbitrary manual truncation of `max_length` was performed if the condition is not met to ensure truncation is performed regardless.

[21]:
```
#--------------------------#
max_length = 700
tolerance = 0.002
window_size = 100
#--------------------------#

# Truncates steady state signal function
def truncate_end(array, tolerance=tolerance, window_size = window_size):
    # Hilbert to calculate analytic signal
    envelope = np.abs(hilbert(array))

    # Sliding window vs envelope for steady-state
    for i in range(len(envelope) - window_size):
        window = envelope[i:i + window_size]
        if np.all(np.abs(np.diff(window)) < tolerance * np.max(envelope)):
            # Truncate the signal from this point onward
            truncated_array = array[:i]
            return truncated_array[:max_length]

    # No steady-state found
    return array[:max_length]

data_truncated_end = []
```

```
for i in range(len(data_filtered)):
    data_truncated_each = []
    for j in range(len(data_filtered[i])):
        data_truncated_each.append(truncate_end(data_filtered[i][j]))
    data_truncated_end.append(data_truncated_each)
```

[22]:
```
#Visualise first bolts after truncation and alignment
plt.figure()
for i in range(len(data_truncated_end)):
    plt.plot(data_truncated_end[i][0])
plt.title("First Bolt Trajectories - Truncated")
plt.xlabel("Frame")
plt.ylabel("Vertical Displacement")
plt.show()
```



As shown above, using the first bolt trajectories, the magnitude varies significantly across experiments due to differences in excitation force. This further validates the Hilbert approach over gradient-based methods.

Normalisation

Normalising the signal is an essential preprocessing step prior to model training to ensure relative

patterns across experimental conditions are comparable regardless of scale and contribute to the model equally. Normalisation was completed using two steps: centring and scaling. The former was performed to adjust the "offset" of the signal to achieve a zero mean value, while the latter rescales the magnitudes of oscillations to ensure all signals remain within the range (-1,1).

Centering `zero_mean()`

Conventionally, min-max scaling is used to subtract a signal by its mean value, demonstrated in methods such as covariance centring in PCA. However, this report opted for the use of z-score normalisation [4]. Z-score normalisation standardises the signal's mean and variance using

$$x_z = \frac{x - \bar{x}}{\sigma_x}$$

where σ is the signal standard deviation, and $\bar{x}$ is the mean of the signal. Z-score was chosen as it normalises between the large ranges of amplitude while also standardising the variations for comparative analysis.

Although centring may appear redundant in the context of a decaying oscillation, the steady state may vary between experiments due to the nature of the experiment. Notably, the camera's position is a source of misalignment, for which the field of view can affect the steady state amplitude within a single experiment as beams towards the field edges may appear misaligned. Calibration variations between the camera and the neutral axis of the beam while oscillating in its plane of flexion are also a source of error and may amplify if not properly reset between experiments.

Scaling `normalize()`

RMS normalisation scales a signal using its root mean square value, calculated as:

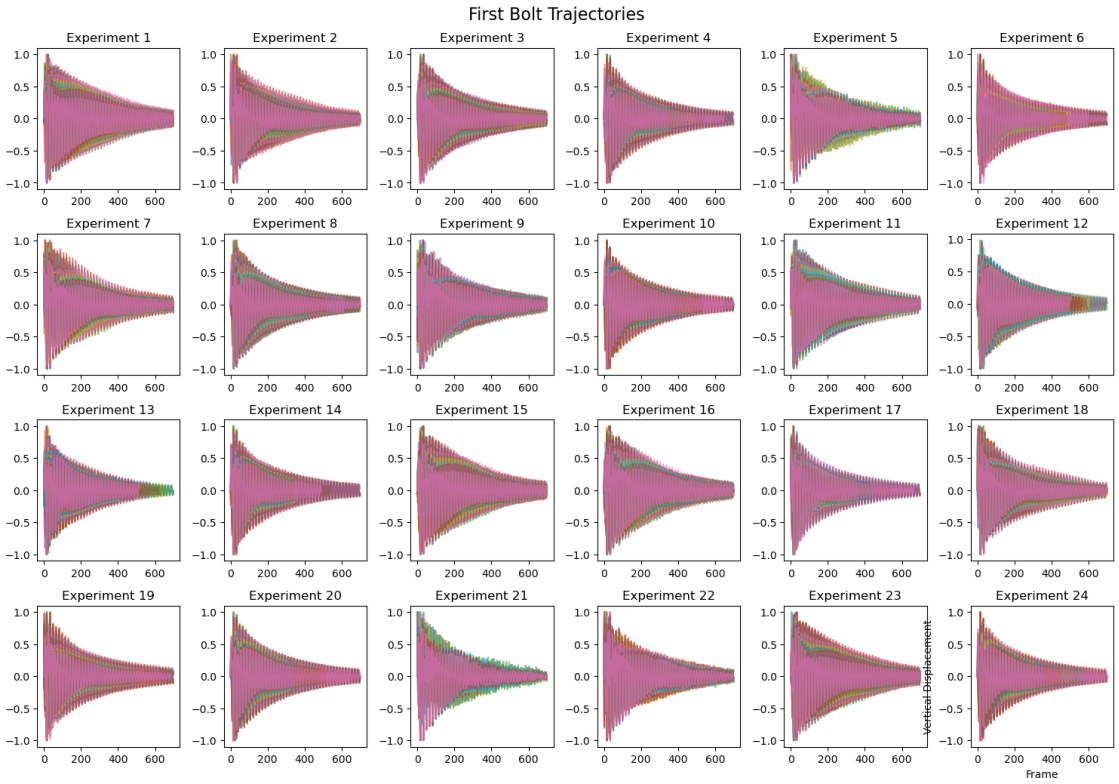
$$\text{RMS} = \sqrt{\frac{1}{N}\sum_{i}^{N} s_i^2}$$

As cited, RMS is particularly effective for signals exhibiting oscillatory behaviour, as it captures a signal's absolute magnitude - i.e. energy content, preserving the relative amplitude variations within a signal. This method also accounts for the variation in signal duration, which differs by the previous truncation step.
Scaling, as with the previous centering step ensures comparatibility of features and uniformity in the dataset.
The significance of maintaining maximum values of ≤ 1 ensures data remains within a stable range and promotes convergence and numerical stability. A relevant example is within the Singular Value Decomposition steps, which can lose precision from large differences in entries. Quantitatively this can be expressed with the condition number [5], which demonstrates the matrix's robustness in decomposition - this is elaborated in Section 2.

[27]:
```
#Z-score centering function
def center(array):
    mean = np.mean(array)
    std = np.std(array)

    z_signal = (array - mean) / std
    return z_signal
```

```
# RMS scaling function
def scale(array):
    rms_value = np.sqrt(np.mean(array**2))

    if rms_value > 0:
        scale_factor = np.max(np.abs(array)) / rms_value
        rms_array = array / (rms_value * scale_factor)
    else:
        rms_array = array

    return rms_array
```

[29]:
```
# Normalisation of data
data_norm = []
for i in range(len(data_truncated_end)):
    data_norm_each = []
    for j in range(len(data_truncated_end[i])):
        data_norm_each.append(scale(center(data_truncated_end[i][j])))
    data_norm.append(data_norm_each)

# Visualise first bolts after all preprocessing
fig, axes = plt.subplots(4, 6, figsize=(15, 10))
axes = axes.flatten()
for i in range(len(data_norm)):
    for j in range(len(data_norm[i])):
        axes[i].plot(data_norm[i][j], label=f"Trajectory {j+1}",alpha = 0.6)
    axes[i].set_title(f"Experiment {i+1}")

plt.tight_layout()
plt.suptitle("First Bolt Trajectories ", y=1.02, fontsize=16)
plt.xlabel("Frame")
plt.ylabel("Vertical Displacement")
plt.show()
```

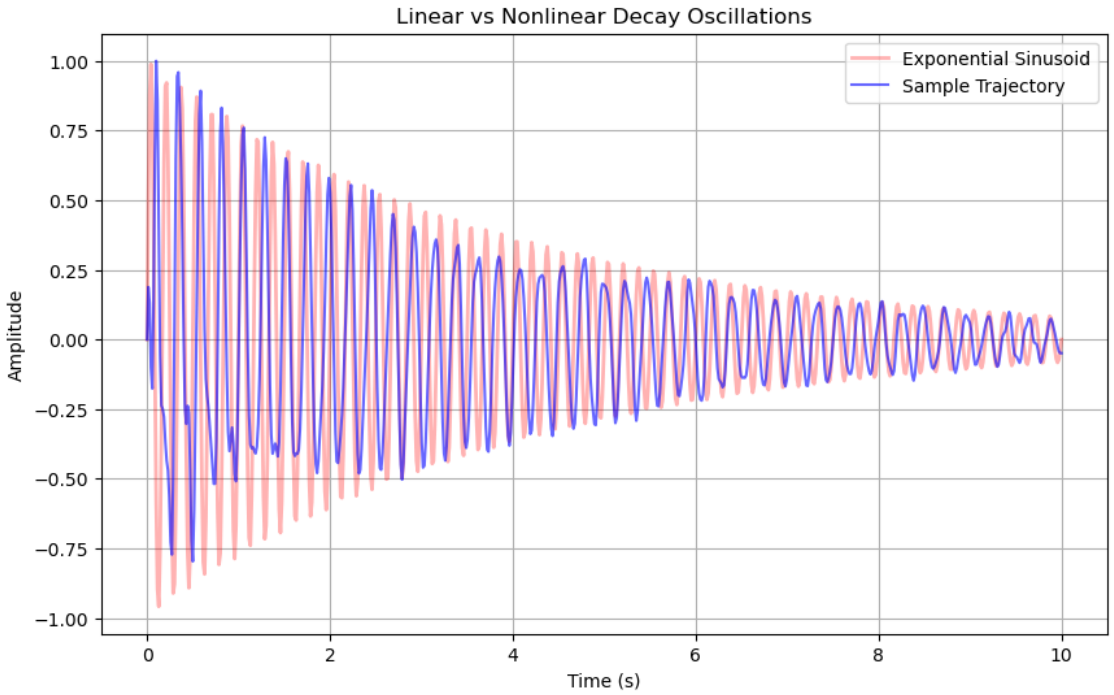


The grid of plots shows the comprehensive trajectory data for each experiment after preprocessing, aggregated into the list `data_norm`.

To validate the earlier assumption that the dynamic equation F is nonlinear, a comparative plot can be shown for a sample trajectory against a matching envelope of an exponential sine wave described in the system identification. As shown, the envelope does not align identically, however appears possible for a linear model to approximate.

[32]:
```
#Damped sine envelope
time = np.linspace(0, 10, len(data_norm[20][1]))
exp_decay = np.exp(-0.25 * time)
sin = exp_decay * np.sin(2 * np.pi * 6 * time)

# Visualise linear model vs data
plt.figure(figsize=(10, 6))
plt.plot(time, sin, label="Exponential Sinusoid", color="red", linewidth=2,␣
↪alpha=0.3)
plt.plot(time, data_norm[20][1], label="Sample Trajectory", color="blue",alpha =␣
↪0.6)
plt.title("Linear vs Nonlinear Decay Oscillations")
plt.xlabel("Time (s)")
plt.ylabel("Amplitude")
plt.legend()
```

```
plt.grid(True)
plt.show()
```



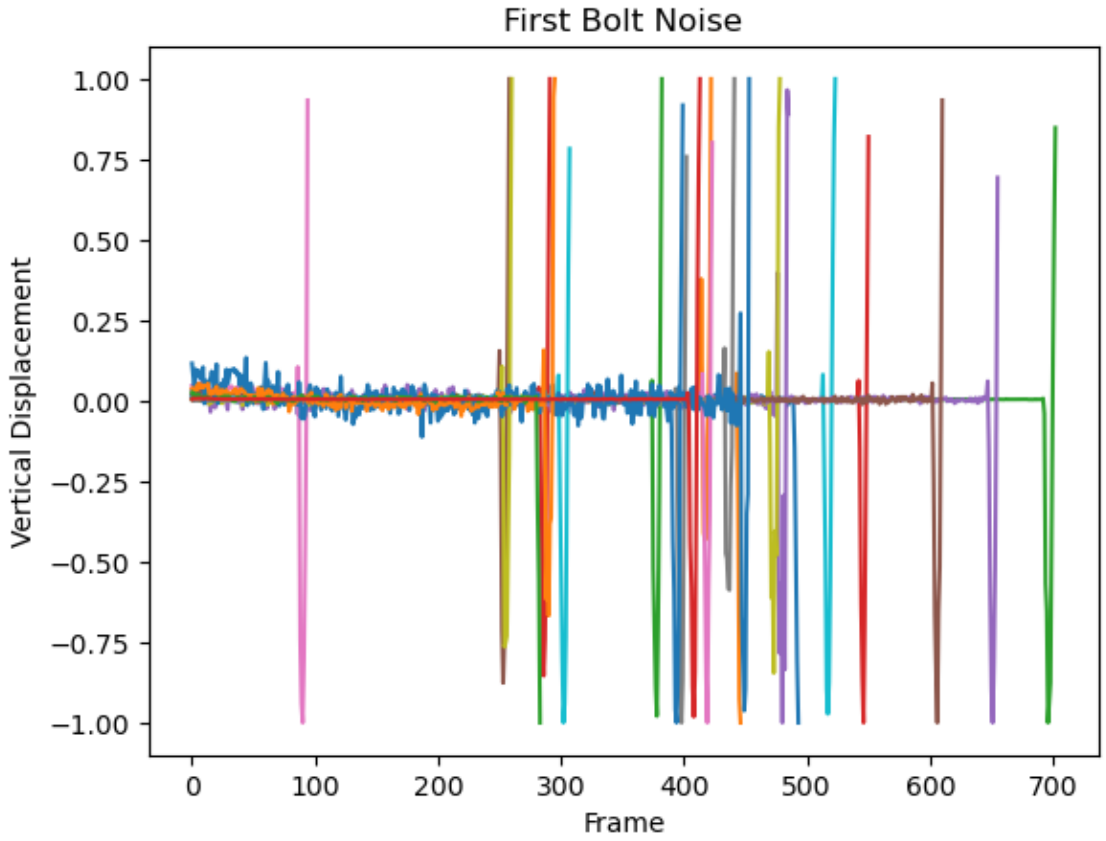
1.3.2 1.2 Noise Power Spectrum

Noise Isolation `truncate_noise()`

The apparent steady-state noise was identified for the initial signals before impact. The function `truncate_noise()` performs the same extraction of the instance of impact, however, returns the indexes which have been truncated. The sources of noise from this experiment can occur from various areas, such as external disturbances and internal camera quantisation noise.

[798]:
```
#Power Spectrum of noise
data_noise = []
for i in range(len(data_all)):
    data_noise_each = []
    for j in range(len(data_all[i])):
        data_noise_each.
 ↪append(scale(center(truncate_noise(reverse(data_all[i][j])))))
    data_noise.append(data_noise_each)

# Visualisation of noise for first bolt
plt.figure()
for i in range(len(data_noise)):
    plt.plot(data_noise[i][0])
```

```
plt.title("First Bolt Noise")
plt.xlabel("Frame")
plt.ylabel("Vertical Displacement")
plt.show()
```



Power Spectral Density (PSD)

The power spectrum of noise was characterised using Power Spectral Density [6]. PSD is an extension to FFT commonly used for noise characterisation in the frequency domain to extract frequency components while accounting for signal length. It is defined as

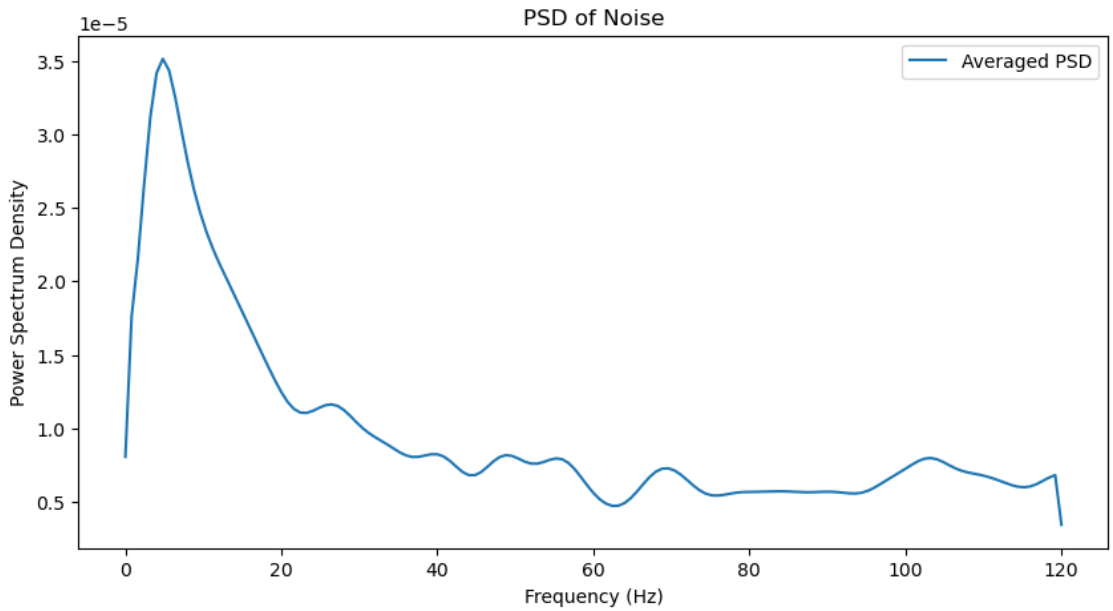
$$\text{PSD}(f) = \frac{1}{M}\sum_{i}^{M}\frac{|FFT(x)|^2}{L}$$

where L is the signal length, and $FFT(x)$ is a FFT applied to a signal.

The `welch()` function was applied to improve the PSD estimation by applying a window to M segments to be averaged. Computing multiple segments reduces variance, and is advantageous especially as the signal length is relatively short.

[809]:
```
# PSD
psd_accumulator = None
num_signals = 0
```

```
nperseg_value = len(data_noise[0][0]) // 8
nfft_value = max(nperseg_value, 300)

for i in data_noise:
    for j in i:
        frequencies, power = welch(j, fs=240, nperseg=nperseg_value,nfft =␣
 ↪nfft_value)

        if psd_accumulator is None:
            psd_accumulator = np.zeros_like(power)

        psd_accumulator += power
        num_signals += 1

average_psd = psd_accumulator / num_signals

# Visualisation of noise PSD
plt.figure(figsize=(10, 5))
plt.plot(frequencies, average_psd, label="Averaged PSD")
plt.title("PSD of Noise")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Power Spectrum Density")
plt.legend()
plt.show()
```



Spectral analysis of the noise signals appears to share a similar dominant frequency as the vibration

frequency in the original data. This is surprising, as noise is expected to be uniform or within a broadband. The significance of this observation is indicative of a relationship between noise energy and system dynamics.
Vibrational systems often amplify external noise at their resonant frequencies. This may suggest a coupling between the camera and system vibration, leading to the amplification of sensor artefacts near this frequency especially when averaged over the entire trajectory.

1.4 2. Linear Model - Delay Embedding

1.4.1 2.0 Linear Model Fitting

A linear model assumes the function governing the evolution of the state space F

$$x_{k+1} = F(x_k)$$

is linear (matrix A). This is done by minimisation of a cost function. Common optimisation methods include Linear Least Squares (LLS), gradient descent, and regularisation.

As the full data is accessible, with computationally feasible dimensionality, gradient descent and heuristic search methods were overly complex and sacrificed interpretability. Additionally, numerical stability was ensured through normalisation from Section 1. Regularisation for sparsity promoting means was deemed unnecessary and added an additional hyperparameter account for. Thus, LLS was the preferred framework.

$$LLS(A) = \frac{1}{2}\sum_{k}^{N} ||Ax_k - y_k||_F^2$$

The principle of LLS is outlined above, used to minimise the error between the prediction states y_k and the original state x_k by calculating the Frobenius norm $||\cdot||_F$.

1.4.2 2.1 Delay Embedding Model

The vibration system inherently possesses temporal characteristics, manifesting in oscillatory behaviour and quasi-periodicity. To effectively capture these dynamics through a single amplitude observable, the state space can be extended to a higher dimension by encoding these temporal dependencies as spatial states, known as delay vectors. This allows the model to explicitly train to fit to time series, rather than treat trajectories independently.

Data Restructuring

Before performing model training, the data matrix was restructured through explicit transposition. This small reorganisation was done to train individual models which share modal dynamics of the system and improve interpretability.

Recalling that there are 24 hammer tests which introduce variability, which can be treated as independent realisations of the test under different conditions. By training across experiments for the same bolt, the same expected dynamics specific to a bolt in its position, proximity to nonlinear sources or boundary conditions, or mounting conditions are isolated. The model is then conditioned to emphasise robustness, and further analysis such as PCA can be used to highlight modal differences between bolt dynamics.

[189]:
```
# Restructuring data matrix
data_bolt = []
```

```
for i in range(len(data_norm[0])):
    bolt_experiments = []
    for j in range(len(data_norm)):
        bolt_experiments.append(data_norm[j][i])
    data_bolt.append(bolt_experiments)

print(f"Original Data Structure: List of {len(data_norm)} experiments with␣
 ↪{len(data_norm[0])} bolts")
print(f"New Data Structure: List of {len(data_bolt)} bolts with␣
 ↪{len(data_bolt[0])} experiments")
```

```
Original Data Structure: List of 24 experiments with 17 bolts
New Data Structure: List of 17 bolts with 24 experiments
```

Train Test Split

A training and testing split is a fundamental practice in machine learning to ensure a model can generalise to unseen data. When a linear model is trained under specific conditions, the model may learn additional data from noise or artefacts over underlying patterns - leading to overfitting. The use of testing data enables reliable assessment of performance by evaluating the model's predictions against a simulated set of unseen data.

From the structure of `data_bolt`, three dimensions of split can be considered. Splitting between bolts was considered unsuitable due to the different underlying dynamics between bolts as mentioned above. Although separating between time is typically cited to be more appropriate when considering predictive models for time-series data, however, led to unusually large discrepancies in testing and training errors. As an alternative, a split between experiments was used, which was deemed justifiable as the approach can highlight a model's generalisability to variations across experiments under identical system dynamics.

Note that deterministic shuffling was used to ensure experimental reproducibility, especially while tuning parameters. A `0.8` split was used as a conventional split ratio.

[192]:
```
#-------------------------#
split_index = 0.8
#-------------------------#

data_train = []
data_test = []

#Deterministic shuffling split
for bolt_exp in data_bolt:
    random.seed(42)
    shuffled_exp = []
    for exp in bolt_exp:
        shuffled_exp.append((random.random(), exp))

    shuffled_exp.sort(key=lambda x: x[0])
    shuffled_exp = [item[1] for item in shuffled_exp]
```

```
idx = int(split_index * len(bolt_exp))

train_exps = shuffled_exp[:idx]
test_exps = shuffled_exp[idx:]

data_train.append(train_exps)
data_test.append(test_exps)
```

Model Training `train_DE_error()`

Training a delay model is done in four steps:

Step 1: Perform delay embedding `delay_embedding()`
For each trajectory, a delay embedding matrix is created to encode the time-series data into a higher dimensional space. These matrices are created such that given a surrogate time series array $x_1, x_2, \ldots, x_N$, the predictor matrix X is created

$$X = \begin{bmatrix} x_1 & x_2 & \cdots & x_{N-d+1} \\ x_2 & x_3 & \cdots & x_{N-d+2} \\ \vdots & \vdots & \ddots & \vdots \\ x_d & x_{d+1} & \cdots & x_N \end{bmatrix}$$

for a specified number of delay states d, and trajectory length N. Similarly, the target matrix Y is constructed;

$$Y = \begin{bmatrix} x_2 & x_3 & \cdots & x_{N-d+2} \\ x_3 & x_4 & \cdots & x_{N-d+3} \\ \vdots & \vdots & \ddots & \vdots \\ x_{d+1} & x_{d+2} & \cdots & x_{N+1} \end{bmatrix}$$

Step 2: Perform decorrelation `decorr()`
The covariance matrix of XX^T is then transformed into an orthogonal basis for linear decorrelation of features.

$$C = XX^T$$

This matrix is then decomposed using singular value decomposition to retrieve the orthogonal basis (decorrelation matrix)

$$C = U\Sigma U^T$$

Step 3: Fit the linear model `lin_model()`
Utilising the LLS methodology outlined earlier, a model is fit between the predictors $\hat{X}$ and targets $\hat{Y}$. These matrices are projected onto the orthogonal basis U calculated from earlier.

$$\begin{cases} \hat{X} = U^T X \\ \hat{Y} = U^T Y \end{cases}$$

The preprocessing step implicitly acts to regularise the dynamics, as the subspace spanned by the decorrelation matrix are inherently capturing the largest directions of variability - i.e. vibrational modes.
An optional polynomial library of functions is applied, however as the model in question is defined to

be strictly linear, this is not used for optimal delay calculations. By rearranging the LLS formulation and setting its derivative to 0, the matrix formulation can be stated as

$$W = \hat{Y}\hat{X}^T(\hat{X}\hat{X}^T)^\dagger$$

where † denotes the Moores-Penrose pseudo inverse. However, for this application, the inversion rank is limited by the parameter `svd_rank` as opposed to the maximum singular value matrix rank κ.

Step 4: Calculate error `calculate_error()`
Calculate the mean normalised error, evaluating how the model predicts Y by the model created by X. The calculation of the mean across all data points was done column-wise for each data point through the use of the Frobenius norm as with the previous step. However, this function explicitly accounts for the dimensionality of each data point by normalisation with respect to the delay value i.e. dividing by $\sqrt{d-1}$ [7] before computing the mean.

[194]:
```
# Delay Embedding Function
def delay_embedding(array, delay):
    N = len(array)
    num_col = N - delay + 1
    delay_matrix = np.zeros((delay, num_col))

    for i in range(delay):
        delay_matrix[i, :] = array[i:i + num_col]

    XX = delay_matrix[:, :-1]
    YY = delay_matrix[:, 1:]
    return XX, YY

# Decorrelation matrix function
def decorr(XX):
    CC = XX @ np.transpose(XX)              #C = XX^T
    U, S, Vt = la.svd(CC, hermitian=True)   #C = UΣU^T
    return U, S, Vt

# LLS Linear model fitting function
def lin_model(XX, YY, U, svd_rank, poly_order=1):
    # Project XX and YY onto the decorrelated subspace
    XX_proj = np.transpose(U[:, 0:svd_rank]) @ XX       #Xproj = Uhat^TX
    YY_proj = np.transpose(U[:, 0:svd_rank]) @ YY       #Yproj = Uhat^TX

    # Polynomial expansion
    XXhat = polyeval(XX_proj, 1, poly_order)        #Xhat = Φ(Xproj)

    # Fit the linear model W = Yproj Xhat^T (Xhat Xhat^T)^+
    WW = (YY_proj @ np.transpose(XXhat)) @ la.pinv(XXhat @ np.transpose(XXhat),␣
↪rcond=1e-6)
    return WW
```

```
# Error calculation function
def calculate_error(WW, XXhat, YY, delay):
    errors = np.linalg.norm(WW @ XXhat - YY, axis=0)

    # Normalize by the square root of the embedding dimension
    normalized_errors = errors / np.sqrt(delay - 1)

    mean_error = np.mean(normalized_errors)

    return mean_error
```

[922]:
```
# Polynomial library function
def polyeval(XX, min_order, max_order):
    Xplist = []
    if min_order == 0:
        Xplist.append(np.ones(XX.shape[1]))
    for n in range(max(1, min_order), max_order + 1):
        ll = list(combinations_with_replacement(range(XX.shape[0]), n))
        for k in ll:
            Xplist.append(XX[k, :].prod(0))
    XXp = np.vstack(Xplist)
    return XXp
```

For preliminary analysis, a conservative `svd_rank` of 5 was chosen. This value dictates the truncation rank of the projection matrix U and the resulting number of singular values (columns) when projection inputs, acting as the models' tradeoff parameter between complexity and generalisation. The relatively high number was to remain conservative in preserving important dynamics and act as a means to compensate for the limited complexity caused by the low polynomial order constraint.

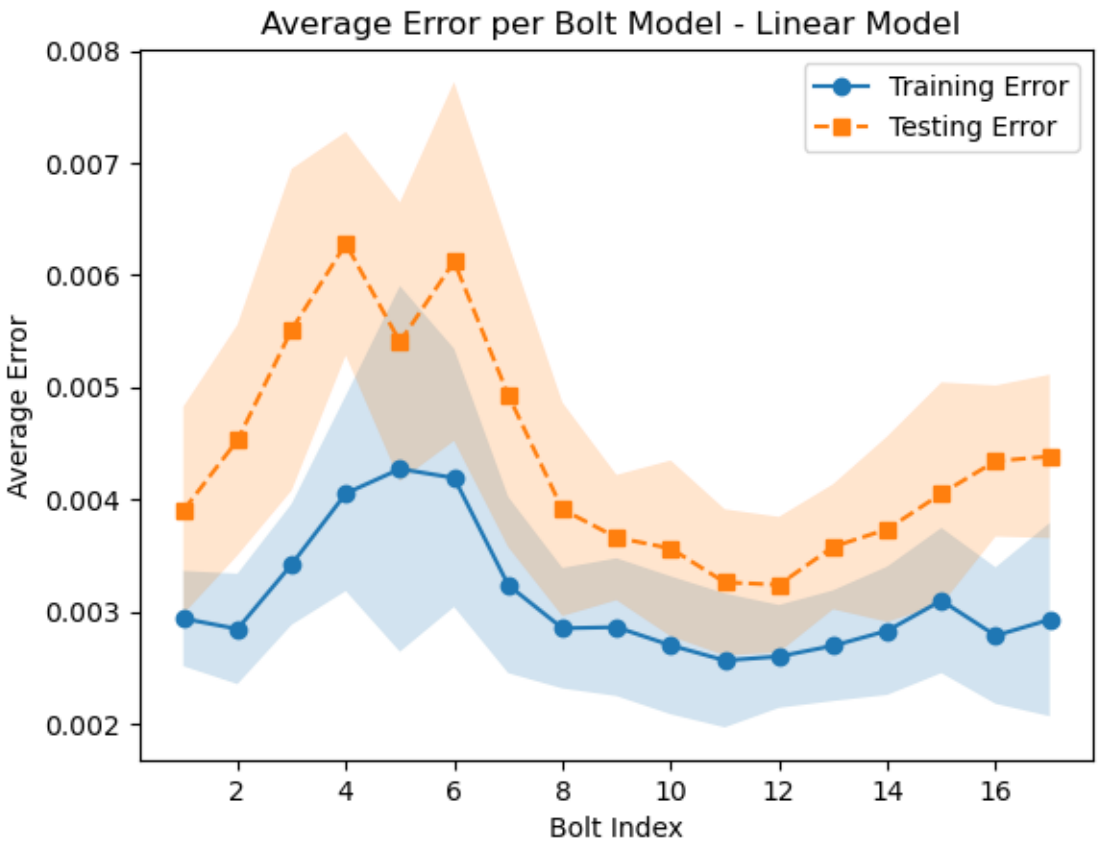
[888]:
```
#Training model
#--------------------------#
svd_rank = 5
poly_order = 1 #linear model
delay = 20
#--------------------------#

def train_DE_error(data, delay, svd_rank, poly_order=1):
    WW_full = []
    error_full = []

    for bolt in data:
        WW_bolt = []
        error_bolt = []

        for experiment in bolt:
            if len(experiment) >= delay + 1:  #Check delay requirement
                # Calculate errorDelay embedding
                XX, YY = delay_embedding(experiment, delay)
```

```
                # Decorrelation
                U, S, Ut = decorr(XX)

                # LLS Model
                WW = lin_model(XX, YY, U, svd_rank, poly_order)

                # calculate error
                XXhat = polyeval(np.transpose(U[:, :svd_rank]) @ XX, 1,␣
↪poly_order)
                mean_error = calculate_error(WW, XXhat, np.transpose(U[:, :
↪svd_rank]) @ YY, delay)

                WW_bolt.append(WW)
                error_bolt.append(mean_error)
            else:
                print("Insufficient delay")

        #Append model for bolt
        WW_full.append(WW_bolt)
        error_full.append(error_bolt)

    return WW_full, error_full
```

[890]:
```
# Training models
WW_train, error_train = train_DE_error(data_train, delay, svd_rank, poly_order)
WW_test, error_test = train_DE_error(data_test, delay, svd_rank, poly_order)

print(f"Number of bolts in training data: {len(WW_train)}")
print(f"Number of models per bolt (training): {len(WW_train[0])}")
print(f"Shape of a single training model: {WW_train[0][0].shape}")
print(f"Shape of training model: {np.shape(error_train)}")
print(f"Number of bolts in testing data: {len(WW_test)}")
print(f"Number of models per bolt (testing): {len(WW_test[0])}")
print(f"Shape of a single testing model: {WW_test[0][0].shape}")
print(f"Shape of testing model: {np.shape(error_test)}")
```

```
Number of bolts in training data: 17
Number of models per bolt (training): 19
Shape of a single training model: (5, 5)
Shape of training model: (17, 19)
Number of bolts in testing data: 17
Number of models per bolt (testing): 5
Shape of a single testing model: (5, 5)
Shape of testing model: (17, 5)
```

[892]:
```
# Simple averaging for visualisation purposes
avg_error_train = [np.mean(bolt_errors) for bolt_errors in error_train]
std_error_train = [np.std(bolt_errors) for bolt_errors in error_train]
avg_error_test = [np.mean(bolt_errors) for bolt_errors in error_test]
std_error_test = [np.std(bolt_errors) for bolt_errors in error_test]

# Visualisation of Training and Testing model errors for each bolt model with␣
 ↪confidence interval
plt.plot(range(1,18), avg_error_train, marker='o', label='Training Error',␣
 ↪linestyle='-')
plt.fill_between(
    range(1,18),
    np.array(avg_error_train) - np.array(std_error_train),
    np.array(avg_error_train) + np.array(std_error_train),
    alpha=0.2,
)

plt.plot(range(1,18), avg_error_test, marker='s', label='Testing Error',␣
 ↪linestyle='--')
plt.fill_between(
    range(1,18),
    np.array(avg_error_test) - np.array(std_error_test),
    np.array(avg_error_test) + np.array(std_error_test),
    alpha=0.2,
)

plt.title("Average Error per Bolt Model - Linear Model ")
plt.xlabel("Bolt Index")
plt.ylabel("Average Error")
plt.legend()
plt.show()
```



The linear model results show the average training and testing error for each of the linear models fitted to each of the 17 bolts. The reconstruction accuracy is consistently low across all bolt indices which suggest linear models are capable of accurately representing the vibration test and reasonable generalisation. Testing errors are expected higher than training errors.

Interestingly, the high testing error at bolt 3 likely corresponds with the nonlinear transducer, as shown in the experiment setup. The nonlinearities introduced may be difficult to accurately capture with a linear model, resulting in poor generalisation in the testing error and variance.

1.4.3 2.2 Model Validation

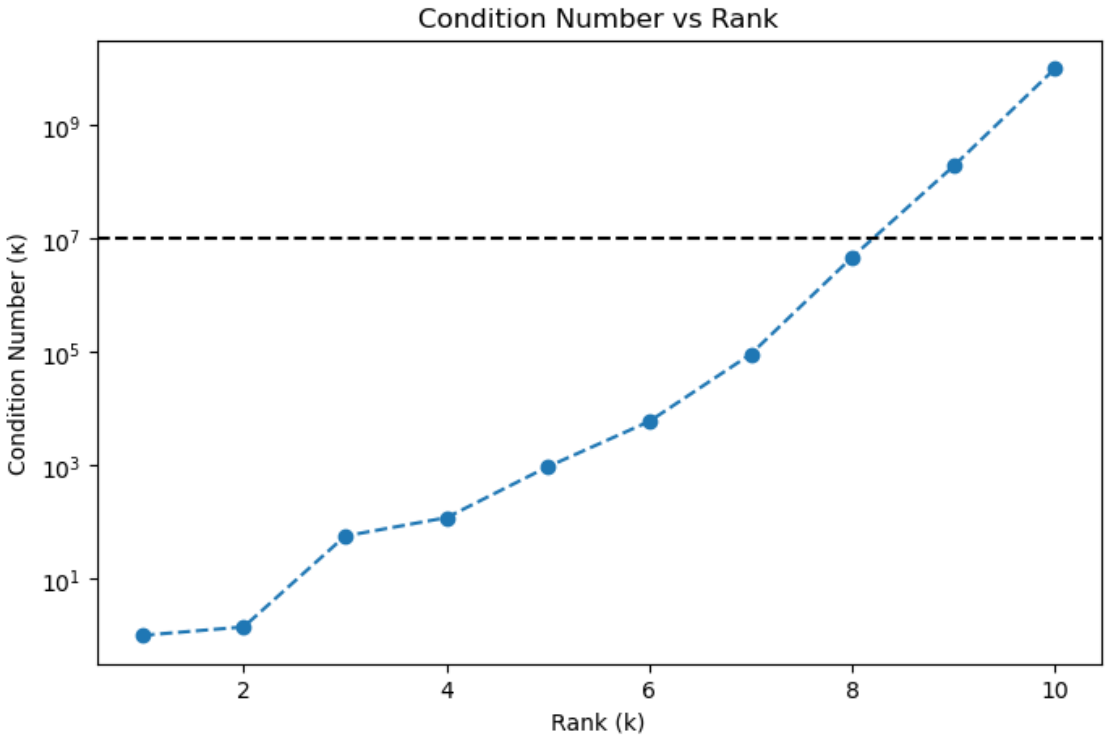
Condition Number Evaluation

To assess the quality of the matrices post-rank truncation, the following code calculates the ratio of the largest and smallest singular values - referred to as the condition number - of the decorrelation matrix U.

$$\kappa_k = \frac{\sigma_1}{\sigma_k}$$

[897]:
```
# Calculating the decorrelation matrix
XX_cn, YY = delay_embedding(data_train[0][0], delay = 10)
U_cn, S_cn, Ut_cn = decorr(XX_cn)
```

```python
# Condition number for each rank
condition_numbers = []
for k in range(1, len(S_cn) + 1):
    S_k_cn = S_cn[:k]
    condition_number = S_k_cn[0] / S_k_cn[-1]
    condition_numbers.append(condition_number)

# Visualise Condition number against truncation rank
plt.figure(figsize=(8, 5))
plt.plot(range(1, len(S_cn) + 1), condition_numbers, marker='o', linestyle='--')
plt.title("Condition Number vs Rank")
plt.xlabel("Rank (k)")
plt.ylabel("Condition Number (κ)")
plt.axhline(y=10e6,linestyle = "--", color = "k")
plt.yscale("log")  # Log scale for better visualization of large values
plt.show()
```



The plot shows the condition number increasing exponentially as k increases. Typically. This indicates numerical stability is retained before 10^4, [8] thus ranks $k \leq 6$ are acceptable truncation ranks, which can capture sufficient information without risking operational instability behaviours dominating.

Polynomial Model Evaluation `poly_eval()`

As noted earlier, the dynamics are governed by a nonlinear function F. In order to capture nonlinear relationships while utilising linear frameworks, F can be approximated as a linear combination of nonlinear functions. The discrete-time formulation redefines F such that the original state space x is extended to a higher dimension.

$$F = w_1\phi_1(x) + w_2\phi_2(x) + \cdots + w_m\phi_m(x) = W\Phi(x)$$

The model to train effectively becomes the weights of these nonlinear contributions

One simple method to increase the state space vector x employs monomial combinations, referred to as a polynomial library. The input feature is transformed into a polynomial feature basis $\Phi(x)$ which includes all polynomial contributions up to an appropriately selected order d.

$$\Phi(x) = (1, x_1, \ldots, x_n, x_1^2, x_1x_2, \ldots, x_n^2, \ldots, x_n^d)^T$$

The linear model equation becomes

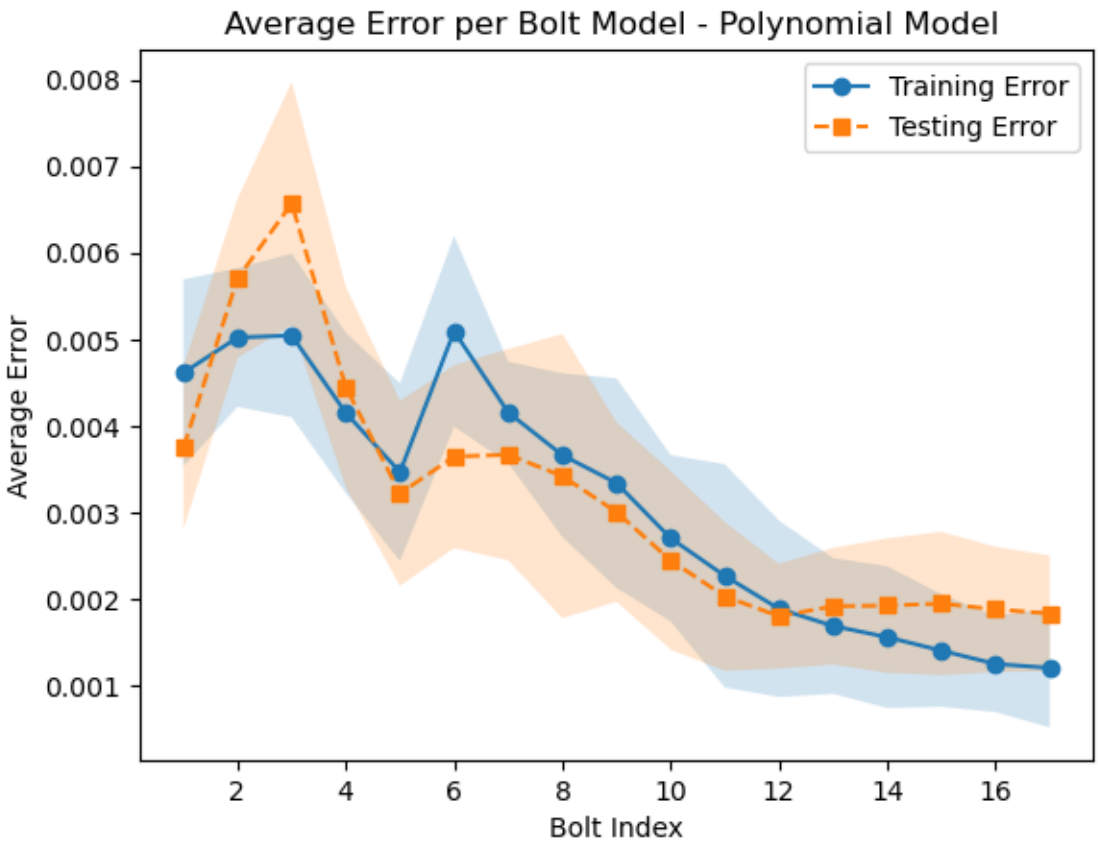
$$\tilde{W} = \hat{Y}(\Phi(\hat{X}))^T(\Phi(\hat{X})(\Phi(\hat{X}))^T)^\dagger$$

[901]:
```
# Training Polynomial model
#--------------------------#
poly_order_poly = 6
#--------------------------#

WW_train_poly, error_train_poly = train_DE_error(data_train, delay,␣
 ↪svd_rank=svd_rank, poly_order=poly_order_poly)
WW_test_poly, error_test_poly = train_DE_error(data_test, delay,␣
 ↪svd_rank=svd_rank, poly_order=poly_order_poly)

print(f"Number of bolts in training data: {len(WW_train_poly)}")
print(f"Number of models per bolt (training): {len(WW_train_poly[0])}")
print(f"Shape of a single training model: {WW_train_poly[0][0].shape}")
print(f"Shape of training model: {np.shape(error_train_poly)}")
print(f"Number of bolts in testing data: {len(WW_test_poly)}")
print(f"Number of models per bolt (testing): {len(WW_test_poly[0])}")
print(f"Shape of a single testing model: {WW_test_poly[0][0].shape}")
print(f"Shape of testing model: {np.shape(error_test_poly)}")
```

```
Number of bolts in training data: 17
Number of models per bolt (training): 19
Shape of a single training model: (5, 461)
Shape of training model: (17, 19)
Number of bolts in testing data: 17
Number of models per bolt (testing): 5
Shape of a single testing model: (5, 461)
Shape of testing model: (17, 5)
```

[902]:
```
# Simple averaging for visualisation purposes
poly_avg_error_train = [np.mean(bolt_errors) for bolt_errors in error_train_poly]
```

```
poly_std_error_train = [np.std(bolt_errors) for bolt_errors in error_train_poly]
poly_avg_error_test = [np.mean(bolt_errors) for bolt_errors in error_test_poly]
poly_std_error_test = [np.std(bolt_errors) for bolt_errors in error_test_poly]

# Visualisation of Training and Testing model errors for each bolt model with␣
 ↪confidence interval
plt.plot(range(1,18), poly_avg_error_train, marker='o', label='Training Error',␣
 ↪linestyle='-')
plt.fill_between(
    range(1,18),
    np.array(poly_avg_error_train) - np.array(poly_std_error_train),
    np.array(poly_avg_error_train) + np.array(poly_std_error_train),
    alpha=0.2,
)

plt.plot(range(1,18), poly_avg_error_test, marker='s', label='Testing Error',␣
 ↪linestyle='--')
plt.fill_between(
    range(1,18),
    np.array(poly_avg_error_test) - np.array(poly_std_error_test),
    np.array(poly_avg_error_test) + np.array(poly_std_error_test),
    alpha=0.2,
)

plt.title("Average Error per Bolt Model - Polynomial Model ")
plt.xlabel("Bolt Index")
plt.ylabel("Average Error")
plt.legend()
plt.show()
```



With a higher polynomial model, the increase in performance is noticeable, especially when examining the scale of the errors as compared to the linear model. With this order, the nonlinearities near the bolt 3 region are more effectively captured. However, this has led to increased relative errors at lower index bolts - likely a consequence of overfitting to bolts which predominantly exhibit linear behaviours.

1.4.4 2.3 Optimum Delay

The underpinning concept behind delay embedding for dynamical systems is Taken's Theorem,[9] which states a sufficiently high delay τ can be used to reconstruct spatial-temporal dependencies. In a practical sense, it can be thought of as the minimum number of delay states in a trajectory required to capture the dominant modes of the vibration test. The goal of the following section is to validate Taken's Theorem for the given dynamic system and investigate the optimal value of τ which recovers the oscillatory attractors.

Selecting an appropriate delay ensures the embedding allows accurate reconstruction of the experiment while balancing generalisation and computational overload. For this report, the `optimum_delay` is a value found through iterating its reconstruction error, calculated as a weighted percentage error `weighted_error()`. For each delay value, both train and test linear models were computed using the same parameters as above, with the optimal value found as the first value that crosses the threshold of `10%`.

Error Calculation `weighted_error()`

Multiple methods [8] were considered for a preprocessing step before calculating the percentage error. The percentage error, although a robust and interpretable calculation for evaluating a model's generalisability, is prone to bias through outliers and absolute differences in magnitude, and does not account for the relative contribution of model sizes. While magnitude is addressed in the preprocessing stages, the function `weighted_error()`, formulated as

$$e = \sqrt{\sum) i^n \left(\frac{N_i}{n_m} \left(\frac{1}{N_i} \sum_{j}^{N_i} e_{i,j} \right)^2 \right)}$$

where $e_{i,j}$ are errrors calculated for model W_i, j, weighted by its contribution to the total number of experiments n_m. This formulation introduces a normalisation weight based on the experimental size to ensure equal contribution between training sets and more experiments than testing sets. An RMS value is also introduced to penalise outliers and improve the robustness of variations in the distribution.

[905]:
```python
# Calculate weighted error function
def weighted_error(error_data, n_exp):
    numerator = 0

    # Iterate through each bolt
    for err in error_data:
        # Weight
        bolt_weight = len(err) / n_exp

        # Max error for this bolt
        max_error = np.max(err)

        # Accumulate weighted max error
        numerator += bolt_weight * max_error

    return numerator  # Weighted maximum error
```

[907]:
```python
#--------------------------#
max_percentage_error = 10  # Stopping criterion
initial_delay = 1  # Starting delay value
max_delay = 50  # Maximum delay value
svd_rank_delay = 4
poly_order_delay = 2
#--------------------------#

n_exp_train = len(data_train[0]) * len(data_train)
n_exp_test = len(data_test[0]) * len(data_test)

delay_values = []
delay_avg_train_errors = []
```

```
delay_avg_test_errors = []
delay_std_train_errors = []
delay_std_test_errors = []
delay_percentage_errors = []

# Initialising
current_delay = initial_delay
min_percentage_error = float("inf")
optimum_delay = None
found_optimum = False

# Iteration loop
for current_delay in range(initial_delay, max_delay + 1, 1):

    WW_train_delay, error_train_delay = train_DE_error(data_train,␣
↪current_delay, svd_rank=svd_rank_delay, poly_order=poly_order_delay)
    WW_test_delay, error_test_delay = train_DE_error(data_test, current_delay,␣
↪svd_rank=svd_rank_delay, poly_order=poly_order_delay)

    # Compute weighted RMS errors
    mean_train_error = weighted_error(error_train_delay, n_exp_train)
    mean_test_error = weighted_error(error_test_delay, n_exp_test)
    train_std_error = np.std([np.mean(bolt_errors) for bolt_errors in␣
↪error_train_delay])
    test_std_error = np.std([np.mean(bolt_errors) for bolt_errors in␣
↪error_test_delay])

    # Calculate percentage error
    #percentage_error = 100 * (mean_test_error - mean_train_error) /␣
↪(mean_train_error + mean_test_error)  # Normalized Absolute PE
    percentage_error = 100 * (mean_test_error - mean_train_error) /␣
↪mean_train_error #Relative Train PE
    #percentage_error = 100 * (mean_test_error - mean_train_error) /␣
↪mean_test_error #Relative Test PE
    #percentage_error = 100 * (mean_test_error - mean_train_error) /␣
↪((mean_train_error + mean_test_error) / 2) #Mean Absolute PE

    # Track threshold percentage error update condition
    if not found_optimum and percentage_error < 10 and current_delay > 5:
        optimum_delay = current_delay
        found_optimum = True  # Set the flag to True to prevent further prints

    print(f"Delay: {current_delay}, Mean Train Error: {mean_train_error:.4f},␣
↪Mean Test Error: {mean_test_error:.4f}, Percentage Error: {percentage_error:.
↪2f}%")
```

```
    # Append results
    delay_values.append(current_delay)
    delay_avg_train_errors.append(mean_train_error)
    delay_avg_test_errors.append(mean_test_error)
    delay_std_train_errors.append(np.mean(train_std_error))
    delay_std_test_errors.append(np.mean(test_std_error))
    delay_percentage_errors.append(percentage_error)

if found_optimum:
    print(f"\nOptimum Delay Value: {optimum_delay}")
```

```
C:\Users\USER\AppData\Local\Temp\ipykernel_26876\656437747.py:38:
RuntimeWarning: divide by zero encountered in divide
  normalized_errors = errors / np.sqrt(delay - 1)
C:\Users\USER\anaconda3\Lib\site-packages\numpy\core\_methods.py:173:
RuntimeWarning: invalid value encountered in subtract
  x = asanyarray(arr - arrmean)
C:\Users\USER\AppData\Local\Temp\ipykernel_26876\1109455522.py:43:
RuntimeWarning: invalid value encountered in scalar subtract
  percentage_error = 100 * (mean_test_error - mean_train_error) /
mean_train_error #Relative Train PE
```

```
Delay: 1, Mean Train Error: inf, Mean Test Error: inf, Percentage Error: nan%
Delay: 2, Mean Train Error: 0.0212, Mean Test Error: 0.0234, Percentage Error:
10.34%
Delay: 3, Mean Train Error: 0.0109, Mean Test Error: 0.0126, Percentage Error:
15.89%
Delay: 4, Mean Train Error: 0.0032, Mean Test Error: 0.0039, Percentage Error:
22.24%
Delay: 5, Mean Train Error: 0.0042, Mean Test Error: 0.0052, Percentage Error:
23.16%
Delay: 6, Mean Train Error: 0.0054, Mean Test Error: 0.0066, Percentage Error:
23.21%
Delay: 7, Mean Train Error: 0.0064, Mean Test Error: 0.0079, Percentage Error:
23.65%
Delay: 8, Mean Train Error: 0.0070, Mean Test Error: 0.0087, Percentage Error:
24.35%
Delay: 9, Mean Train Error: 0.0071, Mean Test Error: 0.0089, Percentage Error:
25.49%
Delay: 10, Mean Train Error: 0.0069, Mean Test Error: 0.0087, Percentage Error:
25.71%
Delay: 11, Mean Train Error: 0.0067, Mean Test Error: 0.0082, Percentage Error:
21.71%
Delay: 12, Mean Train Error: 0.0066, Mean Test Error: 0.0074, Percentage Error:
12.64%
Delay: 13, Mean Train Error: 0.0060, Mean Test Error: 0.0066, Percentage Error:
9.84%
Delay: 14, Mean Train Error: 0.0053, Mean Test Error: 0.0059, Percentage Error:
```

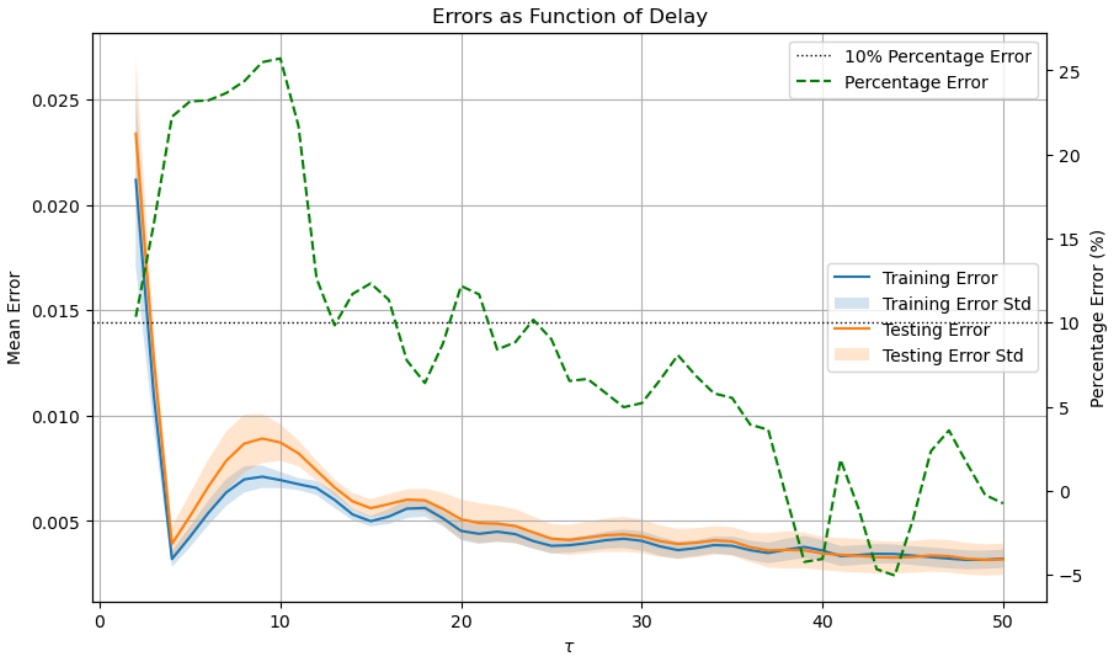
```
11.71%
Delay: 15, Mean Train Error: 0.0050, Mean Test Error: 0.0056, Percentage Error:
12.33%
Delay: 16, Mean Train Error: 0.0052, Mean Test Error: 0.0058, Percentage Error:
11.36%
Delay: 17, Mean Train Error: 0.0056, Mean Test Error: 0.0060, Percentage Error:
7.75%
Delay: 18, Mean Train Error: 0.0056, Mean Test Error: 0.0060, Percentage Error:
6.41%
Delay: 19, Mean Train Error: 0.0051, Mean Test Error: 0.0056, Percentage Error:
8.76%
Delay: 20, Mean Train Error: 0.0045, Mean Test Error: 0.0051, Percentage Error:
12.18%
Delay: 21, Mean Train Error: 0.0044, Mean Test Error: 0.0049, Percentage Error:
11.68%
Delay: 22, Mean Train Error: 0.0045, Mean Test Error: 0.0049, Percentage Error:
8.40%
Delay: 23, Mean Train Error: 0.0044, Mean Test Error: 0.0048, Percentage Error:
8.84%
Delay: 24, Mean Train Error: 0.0040, Mean Test Error: 0.0045, Percentage Error:
10.17%
Delay: 25, Mean Train Error: 0.0038, Mean Test Error: 0.0042, Percentage Error:
9.01%
Delay: 26, Mean Train Error: 0.0038, Mean Test Error: 0.0041, Percentage Error:
6.53%
Delay: 27, Mean Train Error: 0.0040, Mean Test Error: 0.0042, Percentage Error:
6.65%
Delay: 28, Mean Train Error: 0.0041, Mean Test Error: 0.0043, Percentage Error:
5.82%
Delay: 29, Mean Train Error: 0.0042, Mean Test Error: 0.0044, Percentage Error:
4.97%
Delay: 30, Mean Train Error: 0.0041, Mean Test Error: 0.0043, Percentage Error:
5.21%
Delay: 31, Mean Train Error: 0.0038, Mean Test Error: 0.0040, Percentage Error:
6.57%
Delay: 32, Mean Train Error: 0.0036, Mean Test Error: 0.0039, Percentage Error:
8.07%
Delay: 33, Mean Train Error: 0.0037, Mean Test Error: 0.0040, Percentage Error:
6.83%
Delay: 34, Mean Train Error: 0.0039, Mean Test Error: 0.0041, Percentage Error:
5.79%
Delay: 35, Mean Train Error: 0.0038, Mean Test Error: 0.0040, Percentage Error:
5.53%
Delay: 36, Mean Train Error: 0.0036, Mean Test Error: 0.0038, Percentage Error:
3.94%
Delay: 37, Mean Train Error: 0.0035, Mean Test Error: 0.0036, Percentage Error:
3.63%
Delay: 38, Mean Train Error: 0.0036, Mean Test Error: 0.0036, Percentage Error:
```

```
-0.38%
Delay: 39, Mean Train Error: 0.0038, Mean Test Error: 0.0036, Percentage Error:
-4.23%
Delay: 40, Mean Train Error: 0.0036, Mean Test Error: 0.0035, Percentage Error:
-4.05%
Delay: 41, Mean Train Error: 0.0033, Mean Test Error: 0.0034, Percentage Error:
1.85%
Delay: 42, Mean Train Error: 0.0034, Mean Test Error: 0.0033, Percentage Error:
-1.06%
Delay: 43, Mean Train Error: 0.0034, Mean Test Error: 0.0033, Percentage Error:
-4.66%
Delay: 44, Mean Train Error: 0.0034, Mean Test Error: 0.0033, Percentage Error:
-5.04%
Delay: 45, Mean Train Error: 0.0034, Mean Test Error: 0.0033, Percentage Error:
-1.74%
Delay: 46, Mean Train Error: 0.0033, Mean Test Error: 0.0034, Percentage Error:
2.36%
Delay: 47, Mean Train Error: 0.0032, Mean Test Error: 0.0033, Percentage Error:
3.59%
Delay: 48, Mean Train Error: 0.0032, Mean Test Error: 0.0032, Percentage Error:
1.62%
Delay: 49, Mean Train Error: 0.0032, Mean Test Error: 0.0032, Percentage Error:
-0.24%
Delay: 50, Mean Train Error: 0.0032, Mean Test Error: 0.0032, Percentage Error:
-0.75%

Optimum Delay Value: 13
```

The relationship between the training and testing errors means and standard deviations were shown with their respective percentage discrepancies illustrated as a function of the delay value. The 10% threshold is added to show the position of the optimal delay level for these specific parameters. From the optimisation, the optimum delay value using the given parameters is 13.

[924]:
```
# Visualisation of Training and Testing model errors for each delay value
fig, ax1 = plt.subplots(figsize=(10, 6))

# Training errors
ax1.plot(delay_values, delay_avg_train_errors, label='Training Error',␣
 ↪linestyle='-')
ax1.fill_between(
    delay_values,
    np.array(delay_avg_train_errors) - np.array(delay_std_train_errors),
    np.array(delay_avg_train_errors) + np.array(delay_std_train_errors),
    alpha=0.2,
    label='Training Error Std'
)

# Testing errors
```

```
ax1.plot(delay_values, delay_avg_test_errors, label='Testing Error',␣
 ↪linestyle='-')
ax1.fill_between(
    delay_values,
    np.array(delay_avg_test_errors) - np.array(delay_std_test_errors),
    np.array(delay_avg_test_errors) + np.array(delay_std_test_errors),
    alpha=0.2,
    label='Testing Error Std'
)


ax1.set_title("Errors as Function of Delay")
ax1.set_xlabel("$\\tau$")
ax1.set_ylabel("Mean Error")
ax1.legend(loc="center right")
ax1.grid(True)

ax2 = ax1.twinx()
ax2.axhline(y =10, color='black', linestyle=':', linewidth=1, label='10%␣
 ↪Percentage Error')
ax2.plot(delay_values, delay_percentage_errors, label='Percentage Error',␣
 ↪color='green', linestyle='--')
ax2.set_ylabel("Percentage Error (%)")
ax2.legend(loc="upper right")

plt.show()
```



Observations of the following graph show fairly expected values of the behaviour as τ increases. The training and testing errors decrease as the delay increases, which is expected as a larger delay provides more temporal context for modelling, which can improve the accuracy of predictions.

Small Delays ($\tau < 10$): Small delays appear unable to capture the temporal dynamics, as the delay vectors are highly correlated and are more closely approximate independent data points rather than a time series. The large discrepancy also reflects poor generalisation

Medium Delays ($10 < \tau < 30$): Both errors appear to decrease marginally over the mid-range of delay values, as increasing embedding length correlates to improved capture of oscillatory dynamics. The percentage error decreases below the 10% threshold.

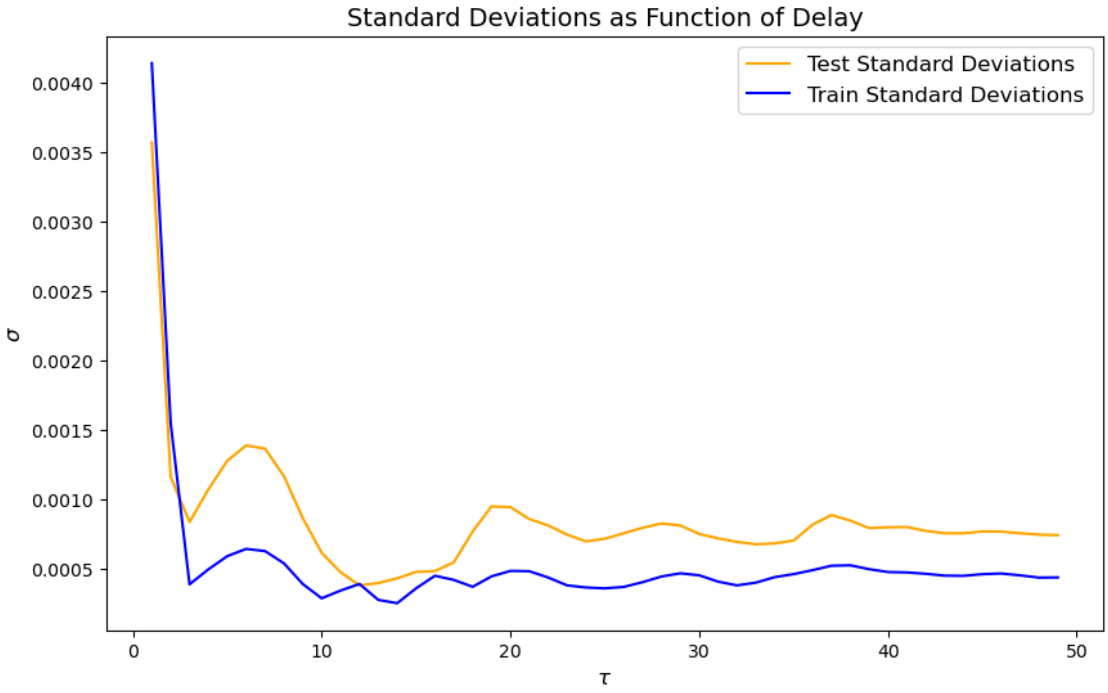
Large Delays ($\tau > 30$): Training and testing errors appear to stabilize at larger delays. This suggests the embedding has stabilised such that increasing the delay beyond a certain point adds diminishing returns in terms of accuracy, and the fluctuation of percentage errors into close negatives indicates the alignment and convergence of data points.

Standard Deviations

The plot below compares the standard values for both training and testing sets isolated from other components of the plot. The behaviour parallels the behaviour of the mean values across both data sets, with both plots showing a large decrease as τ increases. This makes the model underfit, poorly conditioned and unable to generalise leading to large variations in the error across both training and testing sets. The sharp decrease can be attributed to the increase in information captured, allowing noise to be smoothed from consistent dynamic features.

Past $\tau = 10$, the standard deviations appear consistent, suggesting persistent model variability. The model however appears to overfit, as the training error consistently evaluates to a lower value than the testing error - indicating possible model parameter revision, such as inspection of `poly_order` and `svd_rank`.

[913]:
```
# Visualisation of error standard deviations
plt.figure(figsize=(10, 6))
plt.plot(delay_std_test_errors, label="Test Standard Deviations",␣
 ↪color="orange", linestyle="-")
plt.plot(delay_std_train_errors, label="Train Standard Deviations",␣
 ↪color="blue", linestyle="-")

plt.title("Standard Deviations as Function of Delay", fontsize=14)
plt.xlabel("$\\tau$", fontsize=12)
plt.ylabel("$\\sigma$", fontsize=12)
plt.legend(fontsize=12)

plt.show()
```



1.5 3. Linear Model - Singular Value Decomposition

Rank Evaluation `svd_rank_eva()`

The following section illustrates the value of `svd_rank`. As defined earlier, this parameter defined the number of columns i.e. `U[:,0:rank]` used when projecting the data. In earlier sections, `svd_rank` was chosen to equal `5` arbitrarily as a conservative estimate. To fully visualise the effects of varying this number on a delay-embedded model, several iterations were performed for increments of truncation ranks to evaluate the training and testing error.

Each series of models was trained using the `optimum_delay` value identified from the previous section, isolating the effect of the singular value rank. Additionally, the analysis was also extended to investigate experimental cases for each model polynomial order ranging from a linear model (`order = 1`), to a cubic polynomial (`order = 3`).

[501]:
```
#-------------------------#
svd_ranks = range(1,16)
#-------------------------#

# SVD Loop evaluation function
def svd_rank_eval(data_train, data_test, poly_order_svd, svd_ranks = svd_ranks,␣
 ↪delay=optimum_delay):
    train_errors = []
    test_errors = []
    train_errors_std = []
```

```
    test_errors_std = []

    # Loop over SVD ranks
    for i in svd_ranks:
        # Train models
        _, error_train = train_DE_error(data_train, delay, i, poly_order_svd)
        _, error_test = train_DE_error(data_test, delay, i, poly_order_svd)

        # Mean
        train_errors_bolt = [np.mean(bolt_errors) for bolt_errors in error_train]
        test_errors_bolt = [np.mean(bolt_errors) for bolt_errors in error_test]
        train_errors.append(np.mean(train_errors_bolt))
        test_errors.append(np.mean(test_errors_bolt))

        # Standard deviation
        train_errors_std.append(np.std(train_errors_bolt))
        test_errors_std.append(np.std(test_errors_bolt))

    return (np.array(train_errors), np.array(test_errors),
            np.array(train_errors_std), np.array(test_errors_std))
```

R^2 Calculation

In addition to the raw error values, the coefficient of determination, R^2 was also calculated for each model. This value is a common metric used as a measure of training/testing accuracy by measuring the model's ability to predict variance within the data [10].

$$R^2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}}$$

where SS_{tot} represents the residual sum of squares, and SS_{tot} represents the total variance within the data, each formulated as a modification of least squares;

$$\begin{cases} SS_{\text{res}} = \sum_i^n (y_i - \hat{y}_i)^2 \\ SS_{\text{tot}} = \sum_i^n (y_i - \bar{y}_i)^2 \end{cases}$$

for which $\hat{y}$ is the predicted model values, and $\bar{y}$ represents the mean of observed values.

The value of R^2 ranges from (0,1), and complements the absolute error values by contexualising its performance accounting for variance and data complexity. This is especially relevant as performance is compared between models of varying complexities (polynomial orders).

[514]:
```
# Computing R^2 function
def train_R2(data, delay, svd_rank, poly_order=1):
    R2_full = []

    for bolt in data:
        R2_bolt = []  # R^2 values for the current bolt

        for experiment in bolt:
```

```
            if len(experiment) >= delay + 1:  # Check delay requirement
                # Delay embedding
                XX, YY = delay_embedding(experiment, delay)

                # Decorrelation
                U, S, Ut = decorr(XX)

                # LLS Model
                WW = lin_model(XX, YY, U, svd_rank, poly_order)

                # Polynomial expansion
                XX_proj = np.transpose(U[:, :svd_rank]) @ XX
                XXhat = polyeval(XX_proj, 1, poly_order)
                YY_proj = np.transpose(U[:, :svd_rank]) @ YY

                # Predicted output
                YY_pred = WW @ XXhat

                # Compute R^2
                SS_res = np.sum((YY_proj - YY_pred) ** 2)
                SS_tot = np.sum((YY_proj - np.mean(YY_proj, axis=1,␣
↪keepdims=True)) ** 2)
                R2 = 1 - (SS_res / SS_tot)

                R2_bolt.append(R2)
            else:
                print("Insufficient delay")
                R2_bolt.append(None)  # Append None for insufficient data

        # Append R^2 values for the bolt
        R2_full.append(R2_bolt)

    return R2_full

# SVD Loop R^2 evaluation function
def svd_rank_eval_r2(data_train, data_test, poly_order_svd, svd_ranks =␣
↪svd_ranks, delay=optimum_delay):
    train_r2 = []
    test_r2 = []
    train_r2_std = []
    test_r2_std = []

    # Loop over SVD ranks
    for svd_rank in svd_ranks:

        #Training
        r2_train = train_R2(data_train, delay, svd_rank, poly_order_svd)
```

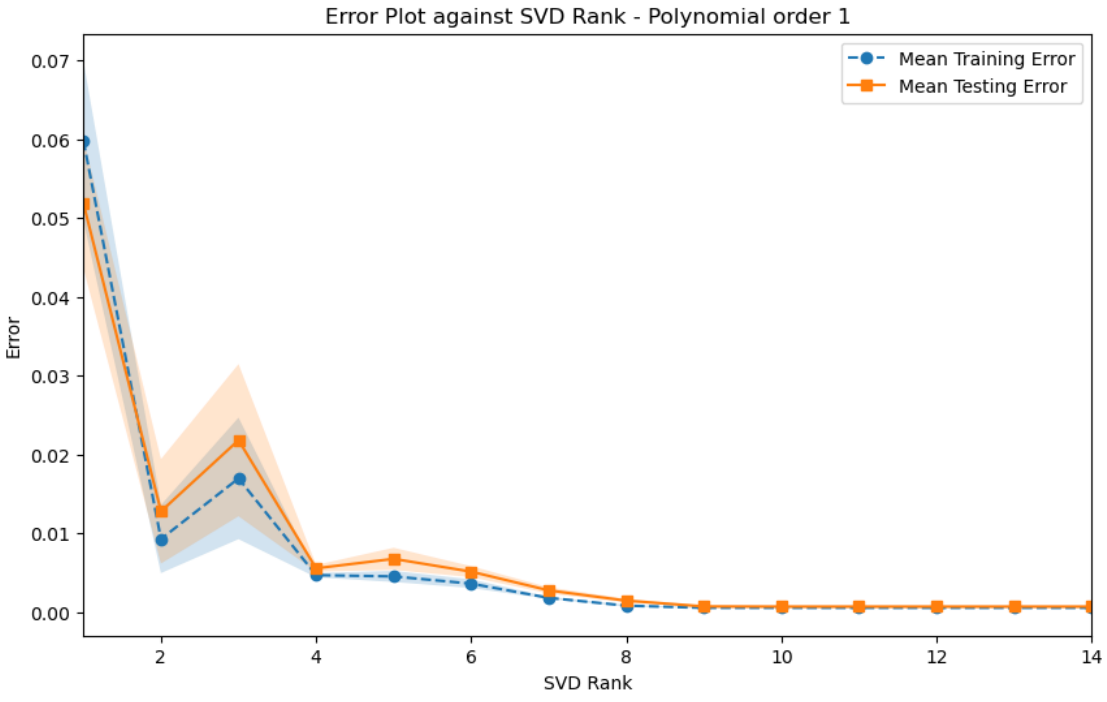
```
        r2_test = train_R2(data_test, delay, svd_rank, poly_order_svd)

        train_r2_bolt = []
        test_r2_bolt = []

        for bolt_r2 in r2_train:
            valid_r2 = [r2 for r2 in bolt_r2 if r2 is not None]
            if valid_r2:
                train_r2_bolt.append(np.mean(valid_r2))

        for bolt_r2 in r2_test:
            valid_r2 = [r2 for r2 in bolt_r2 if r2 is not None]
            if valid_r2:
                test_r2_bolt.append(np.mean(valid_r2))

        # Median
        train_r2.append(np.median(train_r2_bolt))
        test_r2.append(np.median(test_r2_bolt))

        # Standard deviations
        train_r2_std.append(np.std(train_r2_bolt))
        test_r2_std.append(np.std(test_r2_bolt))

    return (np.array(train_r2), np.array(test_r2),
            np.array(train_r2_std), np.array(test_r2_std))
```

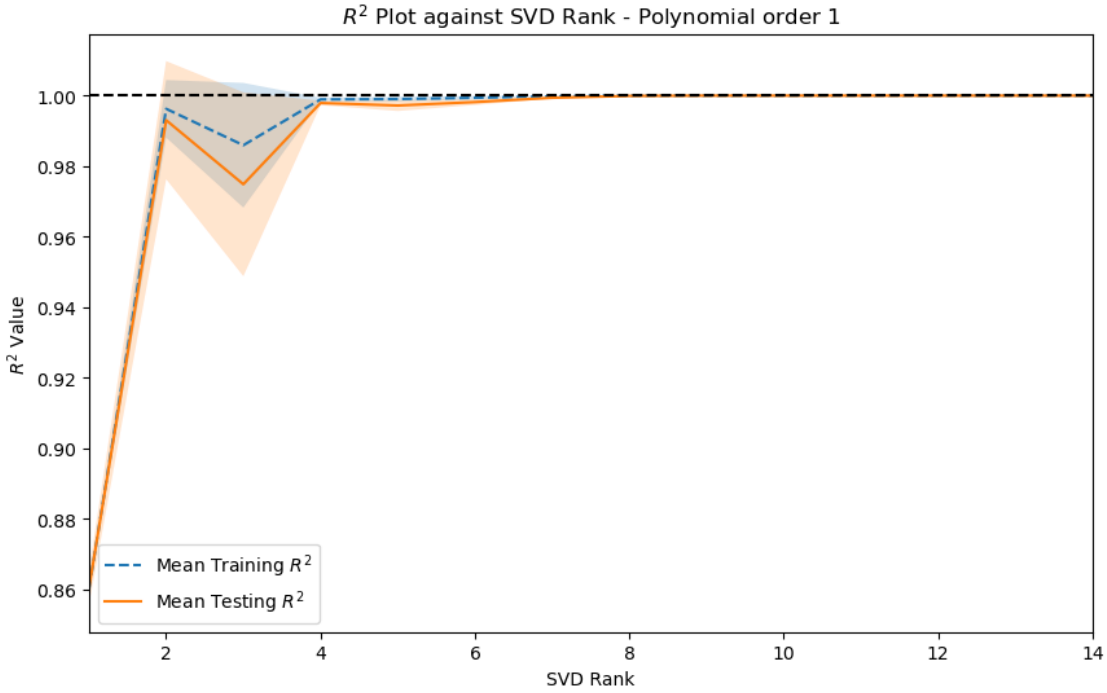
[544]:
```
# Visualisation of svd ranks and errors function
def plot_svd_error(svd_ranks, train_errors, test_errors, train_errors_std,␣
↪test_errors_std,order):
    plt.figure(figsize=(10, 6))

    # Training errors
    plt.plot(svd_ranks, train_errors, label='Mean Training Error',␣
↪linestyle='--', marker='o')
    plt.fill_between(
        svd_ranks,
        train_errors - train_errors_std,
        train_errors + train_errors_std,
        alpha=0.2,
    )

    # Testing errors
    plt.plot(svd_ranks, test_errors, label='Mean Testing Error', linestyle='-',␣
↪marker='s')
    plt.fill_between(
        svd_ranks,
        test_errors - test_errors_std,
```

```
        test_errors + test_errors_std,
        alpha=0.2,
    )

    plt.xlabel("SVD Rank")
    plt.ylabel("Error")
    plt.xlim(1,14)
    plt.title(f"Error Plot against SVD Rank - Polynomial order {order}")
    plt.legend()
    plt.show()



# Visualisation of svd ranks and R^2 function
def plot_svd_r2(svd_ranks, train_r2, test_r2, train_r2_std, test_r2_std, order):
    plt.figure(figsize=(10, 6))

    # Training R^2
    plt.plot(svd_ranks, train_r2, label='Mean Training $R^2$', linestyle='--')
    plt.fill_between(
        svd_ranks,
        train_r2 - train_r2_std,
        train_r2 + train_r2_std,
        alpha=0.2,
    )

    # Testing R^2
    plt.plot(svd_ranks, test_r2, label='Mean Testing $R^2$', linestyle='-')
    plt.fill_between(
        svd_ranks,
        test_r2 - test_r2_std,
        test_r2 + test_r2_std,
        alpha=0.2,
    )

    plt.axhline(y = 1.00,linestyle = '--',color = 'black')
    plt.xlabel("SVD Rank")
    plt.ylabel("$R^2$ Value")
    plt.xlim(1,14)
    plt.title(f"$R^2$ Plot against SVD Rank - Polynomial order {order}")
    plt.legend()

    plt.show()
```

The following code perform the calculations of `svd_rank_eval()` and `svd_rank_eval_r2` for each order of polynomial for both testing and training splits.

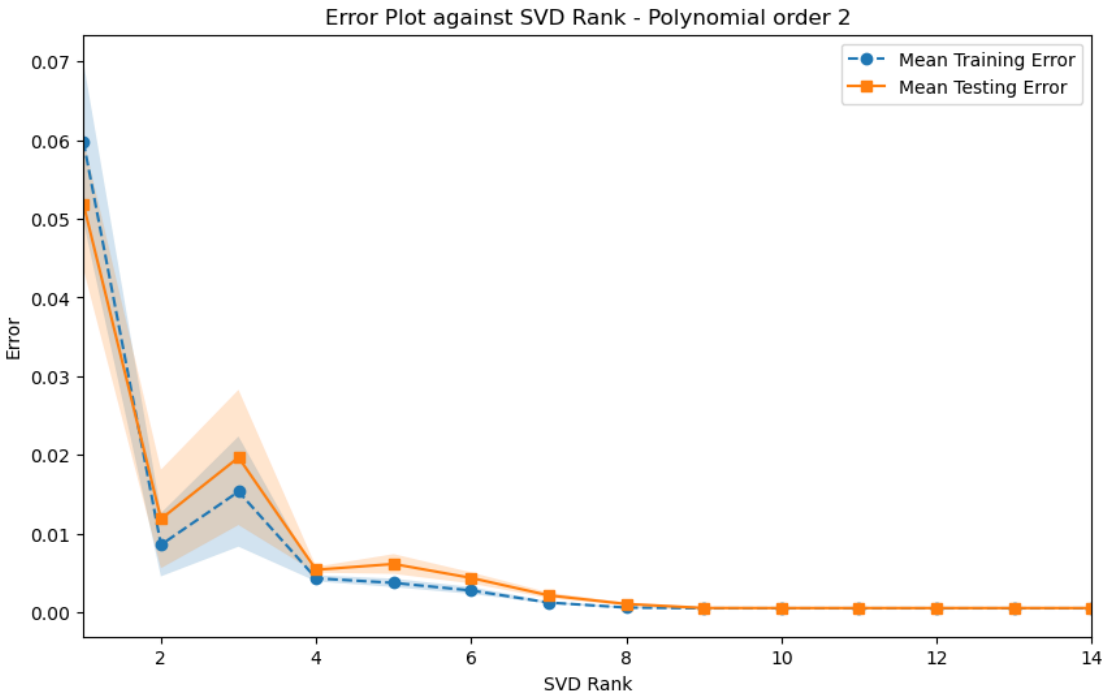
[546]:
```
#Linear
train_errors_svd_1, test_errors_svd_1, train_errors_std_svd_1,␣
 ↪test_errors_std_svd_1 = svd_rank_eval(
    data_train, data_test, poly_order_svd = 1
)
train_r2_svd_1, test_r2_svd_1, train_r2_std_svd_1, test_r2_std_svd_1 =␣
 ↪svd_rank_eval_r2(
    data_train, data_test, poly_order_svd = 1
)

plot_svd_error(
    svd_ranks=svd_ranks,
    train_errors=train_errors_svd_1,
    test_errors=test_errors_svd_1,
    train_errors_std=train_errors_std_svd_1,
    test_errors_std=test_errors_std_svd_1,
    order = 1
)

plot_svd_r2(
    svd_ranks=svd_ranks,
    train_r2 =train_r2_svd_1,
    test_r2 =test_r2_svd_1,
    train_r2_std=train_r2_std_svd_1,
    test_r2_std=test_r2_std_svd_1,
    order = 1
)
```

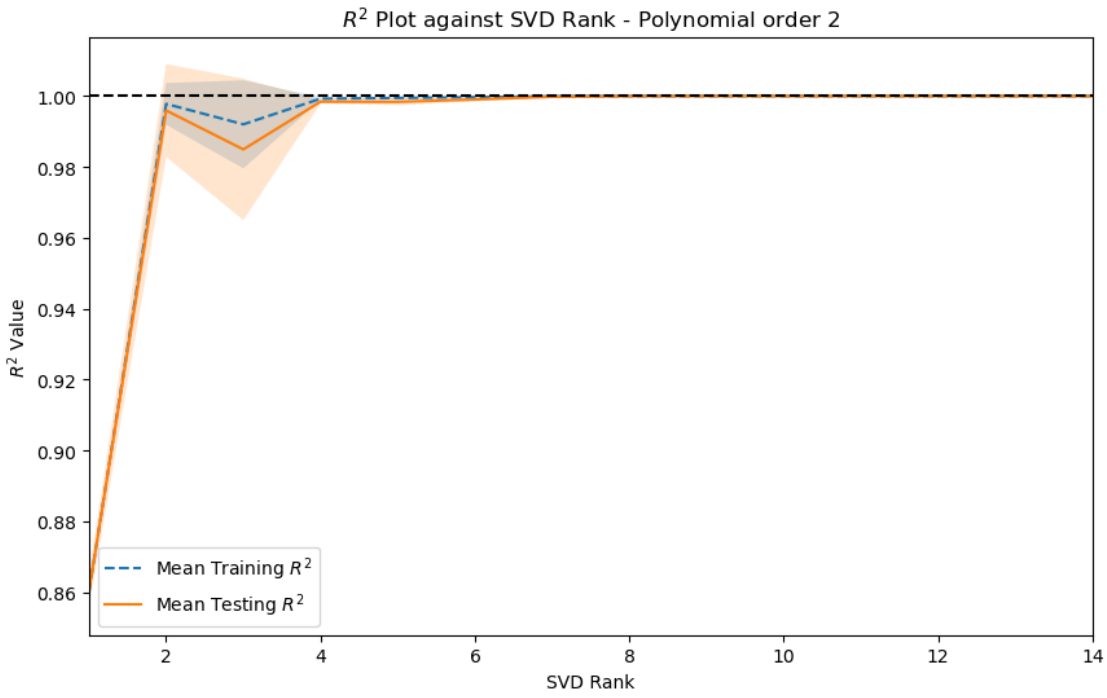




[547]:
```
#Quadratic
train_errors_svd_2, test_errors_svd_2, train_errors_std_svd_2,␣
 ↪test_errors_std_svd_2 = svd_rank_eval(
    data_train, data_test, poly_order_svd = 2
)
train_r2_svd_2, test_r2_svd_2, train_r2_std_svd_2, test_r2_std_svd_2 =␣
 ↪svd_rank_eval_r2(
    data_train, data_test, poly_order_svd = 2
)

plot_svd_error(
    svd_ranks=svd_ranks,
    train_errors=train_errors_svd_2,
    test_errors=test_errors_svd_2,
    train_errors_std=train_errors_std_svd_2,
    test_errors_std=test_errors_std_svd_2,
    order = 2
)

plot_svd_r2(
    svd_ranks=svd_ranks,
```

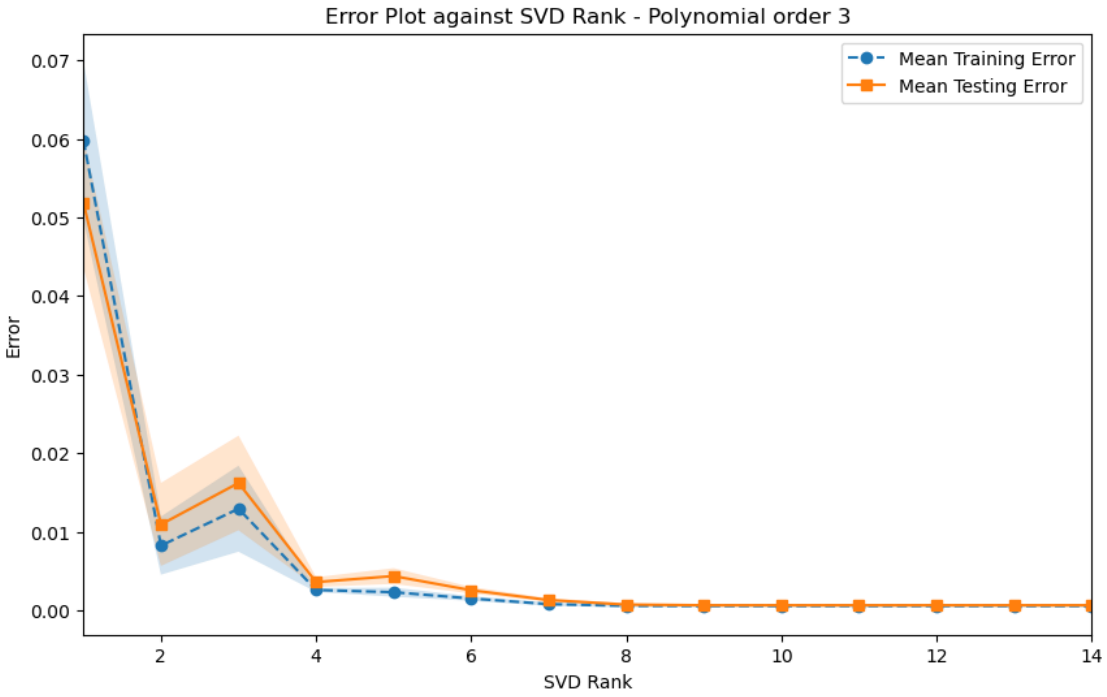
```
    train_r2 =train_r2_svd_2,
    test_r2 =test_r2_svd_2,
    train_r2_std=train_r2_std_svd_2,
    test_r2_std=test_r2_std_svd_2,
    order = 2
)
```

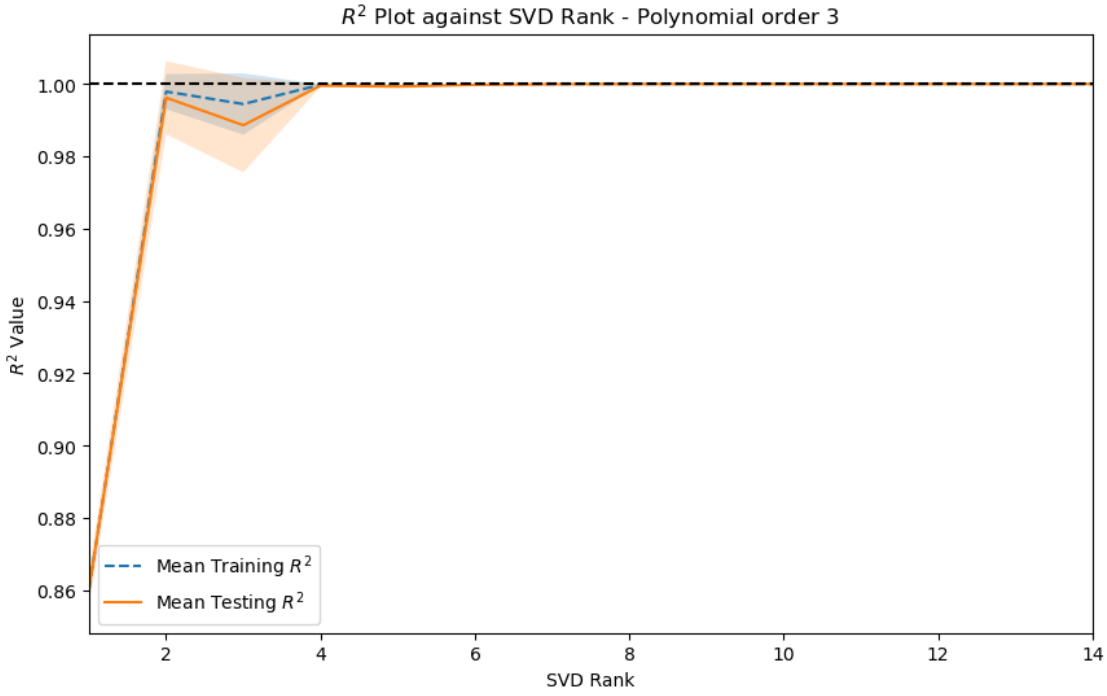




[550]:
```
#Cubic
train_errors_svd_3, test_errors_svd_3, train_errors_std_svd_3,␣
 ↪test_errors_std_svd_3 = svd_rank_eval(
    data_train, data_test,poly_order_svd = 3
)
train_r2_svd_3, test_r2_svd_3, train_r2_std_svd_3, test_r2_std_svd_3 =␣
 ↪svd_rank_eval_r2(
    data_train, data_test, poly_order_svd = 3
)

plot_svd_error(
    svd_ranks=svd_ranks,
    train_errors=train_errors_svd_3,
    test_errors=test_errors_svd_3,
    train_errors_std=train_errors_std_svd_3,
    test_errors_std=test_errors_std_svd_3,
    order = 3
)

plot_svd_r2(
    svd_ranks=svd_ranks,
    train_r2 =train_r2_svd_3,
    test_r2 =test_r2_svd_3,
    train_r2_std=train_r2_std_svd_3,
```

```
    test_r2_std=test_r2_std_svd_3,
    order = 3
)
```





Comparison of Orders

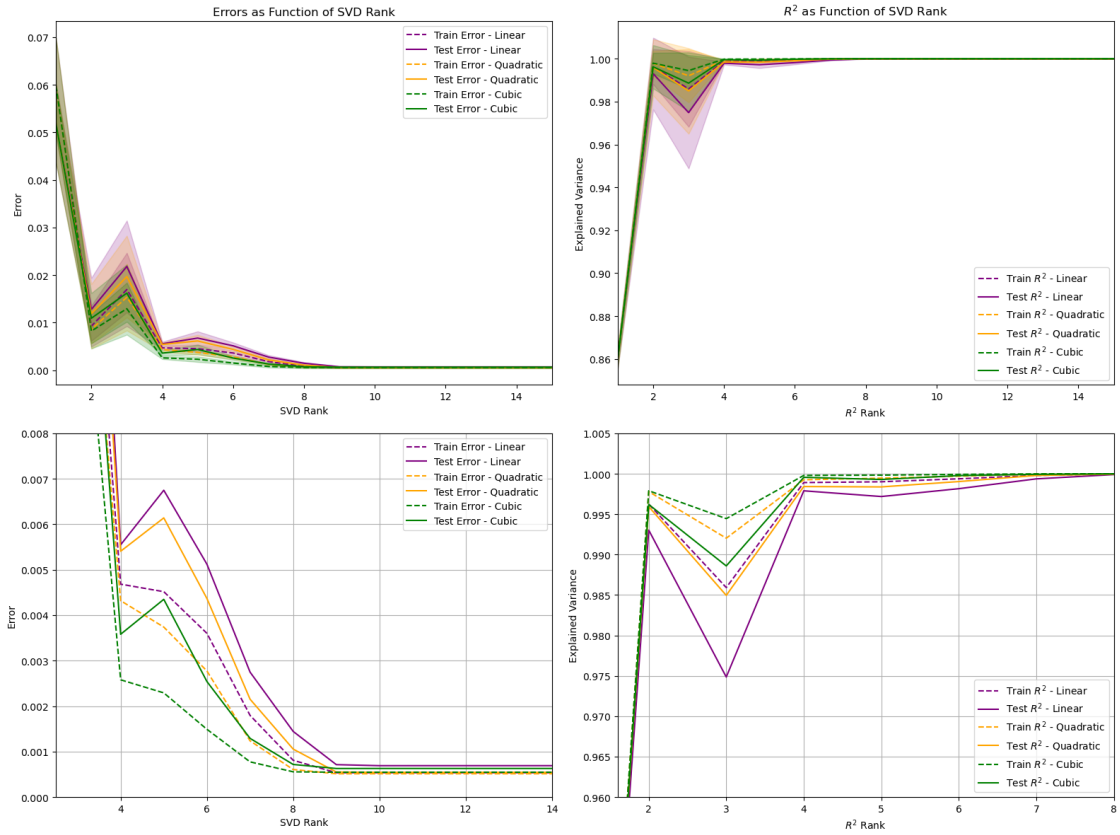
[1078]:
```
fig, axes = plt.subplots(2, 2, figsize=(16, 12))  # 2x2 layout

# Visualization 1 - Errors as Function of SVD Rank
axes[0, 0].plot(
    svd_ranks, train_errors_svd_1, label='Train Error - Linear', linestyle='--',␣
 ↪color='purple'
)
axes[0, 0].fill_between(
    svd_ranks,
    train_errors_svd_1 - train_errors_std_svd_1,
    train_errors_svd_1 + train_errors_std_svd_1,
    alpha=0.2,
    color='purple'
)
axes[0, 0].plot(
    svd_ranks, test_errors_svd_1, label='Test Error - Linear', linestyle='-',␣
 ↪color='purple'
)
axes[0, 0].fill_between(
    svd_ranks,
    test_errors_svd_1 - test_errors_std_svd_1,
    test_errors_svd_1 + test_errors_std_svd_1,
    alpha=0.2,
    color='purple'
)

# Quadratic
axes[0, 0].plot(
    svd_ranks, train_errors_svd_2, label='Train Error - Quadratic',␣
 ↪linestyle='--', color='orange'
)
axes[0, 0].fill_between(
    svd_ranks,
    train_errors_svd_2 - train_errors_std_svd_2,
    train_errors_svd_2 + train_errors_std_svd_2,
    alpha=0.2,
    color='orange'
)
axes[0, 0].plot(
    svd_ranks, test_errors_svd_2, label='Test Error - Quadratic', linestyle='-',␣
 ↪color='orange'
)
axes[0, 0].fill_between(
```

```
    svd_ranks,
    test_errors_svd_2 - test_errors_std_svd_2,
    test_errors_svd_2 + test_errors_std_svd_2,
    alpha=0.2,
    color='orange'
)

# Cubic
axes[0, 0].plot(
    svd_ranks, train_errors_svd_3, label='Train Error - Cubic', linestyle='--',␣
 ↪color='green'
)
axes[0, 0].fill_between(
    svd_ranks,
    train_errors_svd_3 - train_errors_std_svd_3,
    train_errors_svd_3 + train_errors_std_svd_3,
    alpha=0.2,
    color='green'
)
axes[0, 0].plot(
    svd_ranks, test_errors_svd_3, label='Test Error - Cubic', linestyle='-',␣
 ↪color='green'
)
axes[0, 0].fill_between(
    svd_ranks,
    test_errors_svd_3 - test_errors_std_svd_3,
    test_errors_svd_3 + test_errors_std_svd_3,
    alpha=0.2,
    color='green'
)

axes[0, 0].set_xlabel("SVD Rank")
axes[0, 0].set_ylabel("Error")
axes[0, 0].set_title("Errors as Function of SVD Rank")
axes[0, 0].legend()
axes[0, 0].set_xlim(1, 15)

# Visualization 2 - Mean only plot, scaled
axes[1, 0].plot(
    svd_ranks, train_errors_svd_1, label='Train Error - Linear', linestyle='--',␣
 ↪color='purple'
)
axes[1, 0].plot(
    svd_ranks, test_errors_svd_1, label='Test Error - Linear', linestyle='-',␣
 ↪color='purple'
)
```

```
axes[1, 0].plot(
    svd_ranks, train_errors_svd_2, label='Train Error - Quadratic',␣
 ↪linestyle='--', color='orange'
)
axes[1, 0].plot(
    svd_ranks, test_errors_svd_2, label='Test Error - Quadratic', linestyle='-',␣
 ↪color='orange'
)

axes[1, 0].plot(
    svd_ranks, train_errors_svd_3, label='Train Error - Cubic', linestyle='--',␣
 ↪color='green'
)
axes[1, 0].plot(
    svd_ranks, test_errors_svd_3, label='Test Error - Cubic', linestyle='-',␣
 ↪color='green'
)

axes[1, 0].set_xlabel("SVD Rank")
axes[1, 0].set_ylabel("Error")
axes[1, 0].legend()
axes[1, 0].grid(True)
axes[1, 0].set_xlim(2.5, 14)
axes[1, 0].set_ylim(0, 0.008)

# Visualization 3 - R^2 as Function of SVD Rank
#Linear
axes[0, 1].plot(
    svd_ranks, train_r2_svd_1, label='Train $R^2$ - Linear', linestyle='--',␣
 ↪color='purple'
)
axes[0, 1].fill_between(
    svd_ranks,
    train_r2_svd_1 - train_r2_std_svd_1,
    train_r2_svd_1 + train_r2_std_svd_1,
    alpha=0.2,
    color='purple'
)
axes[0, 1].plot(
    svd_ranks, test_r2_svd_1, label='Test $R^2$ - Linear', linestyle='-',␣
 ↪color='purple'
)
axes[0, 1].fill_between(
    svd_ranks,
    test_r2_svd_1 - test_r2_std_svd_1,
    test_r2_svd_1 + test_r2_std_svd_1,
```

```
    alpha=0.2,
    color='purple'
)

# Quadratic
axes[0,1].plot(
    svd_ranks, train_r2_svd_2, label='Train $R^2$ - Quadratic', linestyle='--',␣
 ↪color='orange'
)
axes[0,1].fill_between(
    svd_ranks,
    train_r2_svd_2 - train_r2_std_svd_2,
    train_r2_svd_2 + train_r2_std_svd_2,
    alpha=0.2,
    color='orange'
)
axes[0,1].plot(
    svd_ranks, test_r2_svd_2, label='Test $R^2$ - Quadratic', linestyle='-',␣
 ↪color='orange'
)
axes[0,1].fill_between(
    svd_ranks,
    test_r2_svd_2 - test_r2_std_svd_2,
    test_r2_svd_2 + test_r2_std_svd_2,
    alpha=0.2,
    color='orange'
)

# Cubic
axes[0,1].plot(
    svd_ranks, train_r2_svd_3, label='Train $R^2$ - Cubic', linestyle='--',␣
 ↪color='green'
)
axes[0,1].fill_between(
    svd_ranks,
    train_r2_svd_3 - train_r2_std_svd_3,
    train_r2_svd_3 + train_r2_std_svd_3,
    alpha=0.2,
    color='green'
)
axes[0,1].plot(
    svd_ranks, test_r2_svd_3, label='Test $R^2$ - Cubic', linestyle='-',␣
 ↪color='green'
)
axes[0,1].fill_between(
    svd_ranks,
    test_r2_svd_3 - test_r2_std_svd_3,
```

```
    test_r2_svd_3 + test_r2_std_svd_3,
    alpha=0.2,
    color='green'
)

axes[0,1].set_xlabel("$R^2$ Rank")
axes[0,1].set_ylabel("Explained Variance")
axes[0,1].set_title("$R^2$ as Function of SVD Rank")
axes[0,1].legend()
axes[0,1].set_xlim(1, 15)

# Visualization 4 - R^2 only plot, scaled
axes[1, 1].plot(
    svd_ranks, train_r2_svd_1, label='Train $R^2$ - Linear', linestyle='--',␣
 ↪color='purple'
)
axes[1, 1].plot(
    svd_ranks, test_r2_svd_1, label='Test $R^2$ - Linear', linestyle='-',␣
 ↪color='purple'
)
axes[1, 1].plot(
    svd_ranks, train_r2_svd_2, label='Train $R^2$ - Quadratic', linestyle='--',␣
 ↪color='orange'
)
axes[1, 1].plot(
    svd_ranks, test_r2_svd_2, label='Test $R^2$ - Quadratic', linestyle='-',␣
 ↪color='orange'
)
axes[1, 1].plot(
    svd_ranks, train_r2_svd_3, label='Train $R^2$ - Cubic', linestyle='--',␣
 ↪color='green'
)
axes[1, 1].plot(
    svd_ranks, test_r2_svd_3, label='Test $R^2$ - Cubic', linestyle='-',␣
 ↪color='green'
)

axes[1, 1].set_xlabel("$R^2$ Rank")
axes[1, 1].set_ylabel("Explained Variance")
axes[1, 1].legend()
axes[1, 1].grid(True)
axes[1, 1].set_xlim(1.6, 8)
axes[1, 1].set_ylim(0.96,1.005)

plt.tight_layout()
plt.show()
```



Error and R^2 Plot Analysis

The plot of errors exhibits steep declination before stabilising at higher SVD ranks past `9`. This behaviour mirrors the error trend observed when increasing delay in the previous section. This is likely as both τ and the SVD rank play similar roles in retaining underlying temporal characteristics of the vibration, in such way that low extremes (in this case, the number of singular vectors retained) fail to capture the uncorrelated signals. As more singular vectors or delay states are retained, the accuracy improves up until the limits of the model complexity.

The R^2 graphs show the inverse effect - rapidly increasing at lower ranks before converging to a value $R^2 \approx 1$ as truncation rank rises. As R^2 is a measure of retained variance, the significance of this graph shows the contribution of each singular vector to explaining variance in the data. Past an SVD rank of `4-6`, the system has encapsulated most of the system dynamics which allow effective generalisation.

It is also worth exploring the trends specific to each polynomial order. As shown, quadratic and cubic consistently achieve lower errors when compared to linear models - the trend holding for both testing and training sets. Additionally, convergence appears to be faster relative to lower-order models, likely as these models are inherently more flexible and better equipped to handle nonlinear complexities embedded by the extension of the feature space.

Standard Deviation of Testing Errors

When the standard deviation of testing error increases with increasing SVD rank, this behaviour

suggests that the model becomes more sensitive to artefacts and noise in the test data as the complexity of the model increases. Higher SVD ranks retain more singular vectors, which can capture finer details and noise within the data. While this might improve the model's ability to fit the training data, it often leads to overfitting, where the model starts to learn noise and irrelevant patterns. This sensitivity manifests as higher variability (standard deviation) in testing error across different test cases, highlighting the reduced robustness and generalizability of the model.

However, the opposite effect is observed; the standard deviation of testing error decreases with an increase in SVD rank. This could imply that the additional retained singular vectors retain dynamics rather than noise, such that despite all variance being virtually explained. At high ranks, the singular values maintain stability in encoding general behaviour even as complexity increases. This deviation from expected behavior may be due to well-conditioned data as demonstrated in the previous section, or potentially effective preprocessing stages.

Eckhart-Young Optimisation

To validate the optimal truncated rank k approximation of a matrix A, the problem can be restructured as another minimisation formula

$$e(k) = \text{argmin}||A - A_k||_F$$

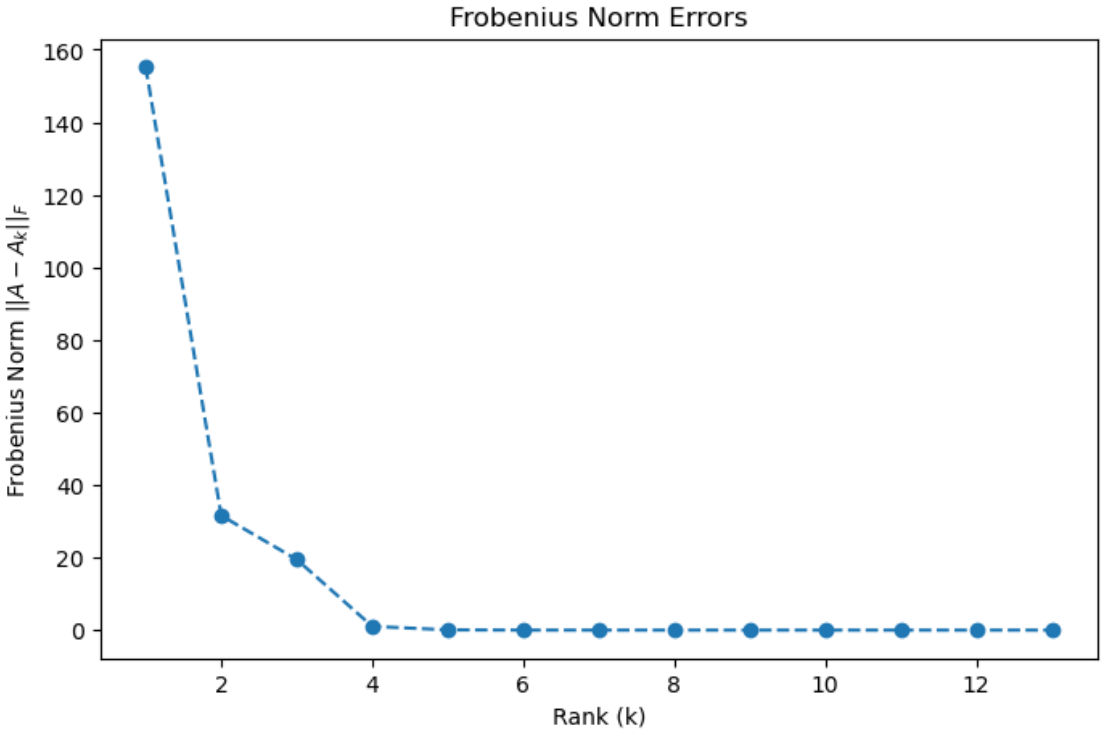
where $||\cdot||_F$ represents the Frobenius norm `frobenius_error()` and A_k is the reconstruction of A from the truncated ranks of its singular matrices. This is the physical application of the Eckhart-Young Theorem[11].

As the optimisation is to find the ideal rank of the decorrelation matrix U, k can be found by computing its parent matrix - the covariance matrix XX. As this is purely to illustrate the concept, only one model was optimised however this analysis can be extended to all models.

The plot below validates the optimum truncation rank is between `2-4` using the elbow method. Using the current rank of `5` may have led to model overfitting behaviours observed in the error plots above.

[844]:
```
#Eckhart-Young Optimisation
X_optim,__ = delay_embedding(data_train[10][10], delay=optimum_delay)
C_optim = X_optim @ np.transpose(X_optim)

# Perform SVD
U_optim, S_optim, Vt_optim = np.linalg.svd(C_optim, full_matrices=False)

# Frobenius norm calculation function
def frobenius_error(C, U, S, Vt, k):
    # Truncate to rank-k
    U_k = U[:, :k]
    S_k = np.diag(S[:k])
    Vt_k = Vt[:k, :]
    C_k = U_k @ S_k @ Vt_k

    # Error
    error = np.linalg.norm(C - C_k, 'fro')
    return error
```

```
# Compute Frobenius norm for each rank
C_k_errors = []
for k in range(1, len(S_optim) + 1):
    k_error = frobenius_error(C_optim, U_optim, S_optim, Vt_optim, k)
    C_k_errors.append(k_error)

# Visualise Frobenius error over k ranks
plt.figure(figsize=(8, 5))
plt.plot(range(1, len(S_optim) + 1), C_k_errors, marker='o', linestyle='--')
plt.title("Frobenius Norm Errors")
plt.xlabel("Rank (k)")
plt.ylabel("Frobenius Norm $||A - A_k||_F$")
plt.show()
```



1.6 4. Dynamic Mode Decomposition

1.6.1 4.0 Koopman Operator

Dynamic Mode Decomposition (DMD) is another data-driven modelling technique, used to analyse complex nonlinear dynamical systems through temporal embedding for similar functionality to delay embedding. It is rooted in Koopman Operator theory, for which a nonlinear system - when extended to a higher dimensional basis - can be evolved in time using a linear operator $\mathcal{K}$ over discrete time steps. The underlying principle is similar to delay embedding and all other linear models, where

the problem is to identify an approximation of the Koopman operator, A such that

$$Y = AX$$

where in most concepts, the state space of X is extended to higher dimensions. For delay embedding, these mappings have been canonically evaluating X over a polynomial library of functions. Under DMD, the same idea applies, where these invariant maps are known as eigenfunctions; denoted by h.

The significance of A is derived from its SVD matrices. The eigenvalues and eigenfunctions which correspond to the system's dynamic modes and its singular vectors provide interpretability to spatial (U) and temporal dynamics (V). In the context of the NTMD system, these can include properties such as its characteristic frequencies and dominant vibration modes. This interpretability is a major advantage, and why DMD was applied to the linear models evaluated from Section 2.

1.6.2 4.1 Linear Model Preprocessing

The following steps formulate and preprocess the linear models identified in delay embedding to be used for DMD calculations. To maintain variable hygiene, the linear models used are created as `WW_train_dmd` and `WW_test_dmd`, but effectively are identical.

[1941]:
```
#--------------------------#
svd_rank_dmd = 5
delay_dmd = optimum_delay
#--------------------------#

# Create separate WW models
WW_train_dmd, _ = train_DE_error(data_train, delay = optimum_delay, svd_rank =␣
↪svd_rank_dmd, poly_order = 1)
```

1.6.3 4.2 Dynamic Mode Decomposition Matrix

DMD is a form of reduced-order modelling, performed to reduce the complexity of a system while retaining its essential dynamics. The DMD matrix A computed represents a linear approximation of the system, which can be used for making predictions or understanding the long-term behavior of the system.

Dynamic Mode Decomposition `dmd()`

To compute the DMD of a model, and for this context the list of linear models for each bolt derived from delay embedding W identified earlier, the following algorithm implements the formulation below for each input model.
A higher-order mapping was first applied to extend the dimensionality. As with previous models, the `polyeval()` basis was applied for its simplicity.

$$W_c = [\Phi(W_1)\ \Phi(W_2)\ \ldots\ \Phi(W_n)]$$

The matrices were then concatenated directly along the feature dimension. Initial preprocessing of W to a normalised mean was attempted. However, this led to over-regulation and loss of oscillatory features across the models.

After the SVD of W_c, a regularisation parameter was applied. Despite the well-conditioned W_c matrix, and normalised values which have been filtered for noise, the formulation persistently failed to capture relevant oscillatory modes. One attempt to mitigate potential numerical instability in the eigenvalue was through use of ridge (L2) regression [12], adding a regularisation parameter ϵ for stability. Rearranging A and substituting the SVD, it can be written that

$$A = (YV\Sigma^{\dagger}U^{T}) + \epsilon I$$

where † denotes a Moores-Penrose pseudo-inverse. The reduced order is achieved in the following step, as the projection of U is applied to the equation, resulting in

$$\tilde{A} = U^{T}YV\Sigma^{\dagger}$$

in which $\tilde{A}$ is the reduced DMD operator.

To calculate the eigenvalues and corresponding eigenvectors of A, an eigenvalue problem can be formulated as

$$\tilde{A}v_j = \lambda_j v_k$$

. This relationship holds as the eigenvalues of A are identical to $\tilde{A}$, leading to the equivalence

$$v_j = \tilde{v}_j$$

[2206]:
```
def dmd(WW, poly_order=1, epsilon=1e-8):

    dmd_operators = []
    dmd_modes = []
    eigenvalues = []

    # Iterate over bolts
    for bolt_index, bolt_models in enumerate(WW):

        polyeval_W_list = []

        # Apply polynomial mapping
        for W in bolt_models:
            polyeval_W = polyeval(W, min_order=1, max_order=poly_order)
            polyeval_W_list.append(polyeval_W)

        W_concat = np.hstack([W for W in bolt_models])
        cond_number = np.linalg.cond(W_concat)

        # SVD of W_concat
        U, S, Vt = np.linalg.svd(W_concat, full_matrices=False)

        # Compute reduced order DMD operator
        A_tilde = U.T @ W_concat @ Vt.T @ np.linalg.inv(np.diag(S)) + epsilon *␣
↪np.eye(W_concat.shape[0])
        dmd_operators.append(A_tilde)
```

```
        # Eigenvalue and Eigenvector computation
        eig_vals, eig_vecs = np.linalg.eig(A_tilde)

        eigenvalues.append(eig_vals)
        dmd_modes.append(U @ eig_vecs)

    print(f"Condition number of W_concat: {cond_number}")
    return dmd_operators, dmd_modes, eigenvalues
```

[2208]:
```
# Computing DMD for the described model
#--------------------------#
poly_order_dmd = 5
epsilon=1e-8          #Training model
#--------------------------#
dmd_operators_train, dmd_modes_train, eigenvalues_train =␣
 ↪dmd(WW_train_dmd,poly_order = poly_order_dmd)

print("DMD Matrix Shape :",np.shape(dmd_operators_train))
print("DMD Modes:", dmd_modes_train[:5])
print("Eigenvalues:", eigenvalues_train[:5])
```

```
Condition number of W_concat: 2.1503872755878595
DMD Matrix Shape : (17, 5, 5)
DMD Modes: [array([[ 0.76850181,  0.16504839,  0.66081054, -0.01264682,
-0.1218056 ],
       [ 0.59289929,  0.02420144, -0.18466896, -0.20696145, -0.08042724],
       [ 0.18620124, -0.03888225, -0.25371773,  0.77574696,  0.05099786],
       [-0.14273605, -0.26332946, -0.22829203,  0.33120525, -0.97518054],
       [ 0.05320645,  0.94937826,  0.64244601, -0.49550657, -0.15848342]]),
array([[ 0.64830172+0.17432697j,  0.64830172-0.17432697j,
         0.0467091 +0.j        ,  0.41389998+0.j        ,
        -0.36447821+0.j        ],
       [ 0.16919181-0.21098727j,  0.16919181+0.21098727j,
        -0.14583862+0.j        ,  0.07838895+0.j        ,
         0.57737598+0.j        ],
       [ 0.5876521 +0.12077717j,  0.5876521 -0.12077717j,
         0.91706748+0.j        ,  0.8782882 +0.j        ,
         0.43594685+0.j        ],
       [-0.17025076+0.10220273j, -0.17025076-0.10220273j,
        -0.34405443+0.j        , -0.22603103+0.j        ,
        -0.25868089+0.j        ],
       [-0.12329144-0.24823351j, -0.12329144+0.24823351j,
        -0.13100821+0.j        ,  0.00786108+0.j        ,
         0.52614366+0.j        ]]), array([[ 0.49269264,  0.84123898,
0.03267807, -0.70324453, -0.08762429],
       [ 0.52876274, -0.47497345, -0.46870737,  0.24312169,  0.6159346 ],
       [ 0.25346543,  0.00568081,  0.67714027,  0.48462508, -0.06105975],
```

```
       [-0.61320158,  0.18636922, -0.1870892 , -0.25833743,  0.71499151],
       [ 0.19339862, -0.17874966,  0.53453179, -0.38044616, -0.31305813]]),
array([[ 0.46973968, -0.22640635, -0.18444534, -0.00820501, -0.17268568],
       [ 0.63504152, -0.37039058,  0.43002329,  0.32096754,  0.49294066],
       [-0.20594946, -0.887686  , -0.68542112, -0.17097754, -0.21690214],
       [-0.57733592, -0.13153534, -0.54316651,  0.23457103,  0.71481714],
       [ 0.01830173, -0.07913914, -0.12738885,  0.90147414, -0.41131385]]),
array([[ 0.47255478,  0.09031184, -0.9319426 , -0.47264437,  0.13331092],
       [ 0.61820703,  0.54912049,  0.35946546,  0.54498982, -0.50400871],
       [-0.27987379,  0.77902519, -0.03305281,  0.50513585,  0.16333455],
       [-0.56123855,  0.06323726,  0.03209158,  0.40711803, -0.8152628 ],
       [ 0.03455441, -0.28183551, -0.01205054,  0.2422521 ,  0.19202035]])]
Eigenvalues: [array([1.00000001, 1.00000001, 1.00000001, 1.00000001,
1.00000001]), array([1.00000001+1.46868701e-16j, 1.00000001-1.46868701e-16j,
       1.00000001+0.00000000e+00j, 1.00000001+0.00000000e+00j,
       1.00000001+0.00000000e+00j]), array([1.00000001, 1.00000001, 1.00000001,
1.00000001, 1.00000001]), array([1.00000001, 1.00000001, 1.00000001, 1.00000001,
1.00000001]), array([1.00000001, 1.00000001, 1.00000001, 1.00000001,
1.00000001])]
```

This calculation computes the DMD matrix formulated for each W trained. In order to aggregate the DMD into a single representative matrix, several methods were evaluated with the objective in mind. As the aim is to capture both global and local system dynamics in a generalisable model - both the variations and significant dynamics should be preserved.

This meant further dimensionality reduction techniques such as PCA were not used. Although theoretically suited, PCA assumes a linear model, and the model itself is not computationally expensive. Other methods such as generalised nonlinear autoencoders were deemed too computationally expensive for the quantity of data available.

The report opted to implement an adaptive method to aggregate the models based on the product of their respective variance and magnitude contributions [13]

Weight Calculation `weights()` The following preceding function calculates the weights for mode aggregation to preserve variance as contribution by bolts is not equalised - i.e. singular values are treated with equal importance. As mentioned, the weight $\mathcal{W}$ was computed using the product of the magnitude and variance, or mathematically

$$\mathcal{W}_i = ||W_{c,i}||_F \times \text{Var}(||W_k||)_F)$$

where i is the index of a bolt model, and k is the index of an experiment. As with all other sections, $||\cdot||_F$ denotes the Frobenius norm.

Modal Aggregation`mode_aggregation()` Modal aggregation involved selecting the top k modes sorted by the largest magnitudes of their corresponding eigenvalues, effectively scaling these modes according to the weight before aggregating into a single matrix

$$\begin{cases} \phi'_j = \mathcal{W}_i \phi_j & \text{for each mode } \phi_j \\ \lambda'_j = \mathcal{W}_i \lambda_j & \text{for each mode } \lambda_j \end{cases}$$

The solution to the final aggregated DMD matrix can then be constructed as

$$\tilde{A}_{\text{agg}} = \Phi' \Lambda \Phi'^{-1}$$

[2211]:
```
# Adaptive weight calculation function
def weights(WW):
    # Frobenius weights
    frobenius_weights = []
    for bolt in WW_train_norm:
        # Concatenate
        concatenated_experiments = np.hstack(bolt)

        # Frobenius norm
        frobenius_norm = np.linalg.norm(concatenated_experiments, ord='fro')
        frobenius_weights.append(frobenius_norm)

    # Variance weights
    variance_weights = []
    for bolt in WW_train_norm:

        # Compute Frobenius
        norms = []
        for exp in bolt:
            norm = np.linalg.norm(exp, ord='fro')
            norms.append(norm)

        # Compute variance of norms
        bolt_variance = np.var(norms)
        variance_weights.append(bolt_variance)

    # Frobenius and Variance product
    weights = []
    for frobenius_weight, variance_weight in zip(frobenius_weights,␣
 ↪variance_weights):
        weights.append(frobenius_weight * variance_weight)

    weights = np.array(weights)
    return weights

# Weighted modal aggregation function
def mode_aggregation(dmd_operators, dmd_modes, eigenvalues, top_k, weights=None):
    agg_modes = []
    agg_eigvals = []

    # Iterate for DMD matrix
    for bolt_modes, bolt_eigvals, weight in zip(dmd_modes, eigenvalues, weights):

        # Sort by magnitude of evals
```

```
        sorted_indices = np.argsort(-np.abs(bolt_eigvals))
        top_indices = sorted_indices[:top_k]

        # Select top-k modes and eigenvalues
        top_modes = bolt_modes[:, top_indices]
        top_k_evals = bolt_eigvals[top_indices]

        # Apply weight
        top_modes *= weight
        top_k_evals *= weight

        agg_modes.append(top_modes)
        agg_eigvals.append(top_k_evals)

    agg_modes = np.concatenate(agg_modes, axis=1)
    agg_eigvals = np.hstack(agg_eigvals)
    aggregated_A = agg_modes @ np.diag(agg_eigvals) @ np.linalg.pinv(agg_modes)

    return aggregated_A
```

1.6.4 4.3 PCA

DMD gives a set of dynamic modes that describe the spatial and temporal evolution of a system. However, DMD may produce a large number of modes, many of which may not be essential. PCA acts to identify the top principal components of maximum variance and projects the DMD operators onto these principal directions. This reduces the dimensionality while retaining significant information. Although the DMD itself for this application is relatively small with an unlikely possibility of redundant modes, the report decided to explore the behaviour and impacts of applying PCA to the computed DMD [15]

Principle Component Analysis`pca()` PCA was used to identify orthogonal principle directions of maximum variance. First, the matrix X is centred by subtraction of the mean

$$B = X - \bar{x}$$

for which the correlation matrix and its SVD are computed i.e.

$$C = \frac{1}{n-1}\Sigma^2$$

The quantity of variance explained by each principal component is proportional to the square of each singular value σ_i^2. The `top_k` components are selected and used to project B

PCA on DMD`pca_dmd()` Applies PCA using the previous function on the concatenated DMD operators. The DMD operators are first concatenated over their feature dimension, and PCA is applied to the concatenated DMD matrix - with the number of retained components dictated by the parameter `pca_components`. The reduced matrix A_{red} is then calculated via

$$A_{\text{red}} = BV^T[:k]$$

where $V^T[:k]$ are the top k principal components

[2276]:
```
# PCA function
def pca(data, pca_components):
    # Center the data by subtracting the mean of each feature
    data_centered = data - np.mean(data, axis=0)

    # Compute SVD
    U, S, Vt = np.linalg.svd(data_centered, full_matrices=False)

    # Singular values and variance calculation
    explained_variance = S**2 / (data.shape[0] - 1)
    total_variance = np.sum(explained_variance)
    explained_variance_ratio = explained_variance / total_variance

    # Top components
    components = Vt.T[:pca_components, :]
    reduced_data = data_centered @ components.T

    return reduced_data, components, explained_variance_ratio[:pca_components]

def pca_dmd(dmd_operators, pca_components):
    dmd_operators_reshaped = [op.reshape(-1, 1) if op.ndim == 1 else op for op␣
 ↪in dmd_operators]

    # Concatenate DMD operators
    concatenated_A = np.hstack(dmd_operators_reshaped)


    # Check the rank of the concatenated matrix
    rank = np.linalg.matrix_rank(concatenated_A)

    # Adjust PCA components dynamically
    effective_pca_components = min(pca_components, rank)

    # Perform PCA
    reduced_A, components, explained_variance_ratio = pca(concatenated_A,␣
 ↪effective_pca_components)
    return reduced_A, components, explained_variance_ratio
```

DMD Matrix Calculation

[2288]:
```
#-------------------------#
top_k = 10
pca_components = 3
#-------------------------#

# DMD Matrix for both conditions
```

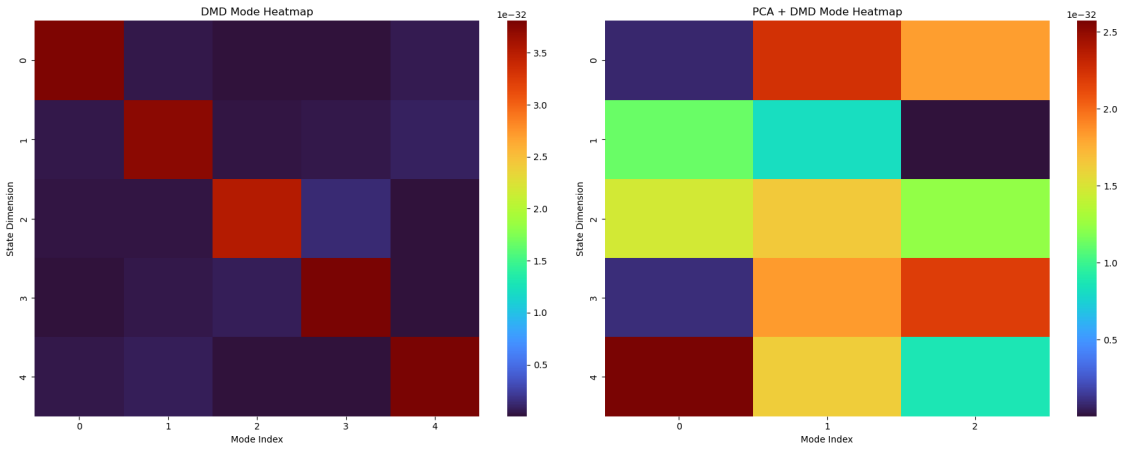
```
DMD_matrix = mode_aggregation(dmd_operators_train, dmd_modes_train,␣
 ↪eigenvalues_train, top_k,weights = weights(WW_train_norm))
DMD_matrix_reduced, components, explained_variance_ratio = pca_dmd(DMD_matrix,␣
 ↪pca_components)

print("DMD Matrix Shape:",DMD_matrix.shape)
print("PCA + DMD Matrix Shape:", DMD_matrix_reduced.shape)
print("DMD Matrix:",DMD_matrix)
print("PCA + DMD Matrix:", DMD_matrix_reduced)
print("\n")
print("Principal Components:", components)
print("Explained Variance Ratio:", explained_variance_ratio)
```

```
DMD Matrix Shape: (5, 5)
PCA + DMD Matrix Shape: (5, 3)
DMD Matrix: [[ 3.78432592e-32+0.j -3.80976682e-34+0.j -6.11592495e-35+0.j
  -8.26729578e-35+0.j -5.04171802e-34+0.j]
 [-4.52106748e-34+0.j  3.72720332e-32+0.j  2.33642341e-34+0.j
  -4.72423711e-34+0.j -9.04059578e-34+0.j]
 [-1.88594178e-34+0.j  3.06091690e-34+0.j  3.52459626e-32+0.j
  -1.22245881e-33+0.j -1.79878636e-34+0.j]
 [-3.21183405e-35+0.j -4.39592575e-34+0.j -7.13750596e-34+0.j
   3.80498385e-32+0.j  1.63342558e-34+0.j]
 [-3.70380029e-34+0.j -7.33884710e-34+0.j  5.34292627e-35+0.j
   1.42878621e-34+0.j  3.80800485e-32+0.j]]
PCA + DMD Matrix: [[ 7.34831513e-34+0.j  2.24484684e-32+0.j  1.81928624e-32+0.j]
 [ 1.12338264e-32+0.j -8.18709226e-33+0.j  2.33309892e-35+0.j]
 [ 1.47643871e-32+0.j -1.64450552e-32+0.j  1.23099806e-32+0.j]
 [-1.01849623e-33+0.j  1.83780614e-32+0.j -2.18554275e-32+0.j]
 [-2.57145487e-32+0.j -1.61943823e-32+0.j -8.67074650e-33+0.j]]


Principal Components: [[ 0.24478956+0.j  0.51589065+0.j  0.65731313+0.j
0.22178217+0.j
  -0.4389613 +0.j]
 [ 0.56369697+0.j -0.23416184+0.j -0.47613312+0.j  0.4474853 +0.j
  -0.44773666+0.j]
 [ 0.23405862+0.j -0.25874539+0.j  0.07932438+0.j -0.8102632 +0.j
  -0.46416438+0.j]]
Explained Variance Ratio: [0.26631942 0.25815134 0.25380095]
```

A heatmap was used to illustrate the DMD modes across both tests to identify dominant patterns and modal contributions

[2280]:
```
# Visualisation of modal contributions on heat map
mode_matrix = np.abs(DMD_matrix)

fig, axes = plt.subplots(1, 2, figsize=(18, 7))
```

```
# Mode matrix
sns.heatmap(mode_matrix, cmap="turbo", cbar=True, ax=axes[0])
axes[0].set_title("DMD Mode Heatmap")
axes[0].set_xlabel("Mode Index")
axes[0].set_ylabel("State Dimension")

# Plot the second DMD mode matrix
mode_matrix_reduced = np.abs(DMD_matrix_reduced)
sns.heatmap(mode_matrix_reduced, cmap="turbo", cbar=True, ax=axes[1])
axes[1].set_title("PCA + DMD Mode Heatmap")
axes[1].set_xlabel("Mode Index")
axes[1].set_ylabel("State Dimension")

# Adjust layout for better spacing
plt.tight_layout()
plt.show()
```



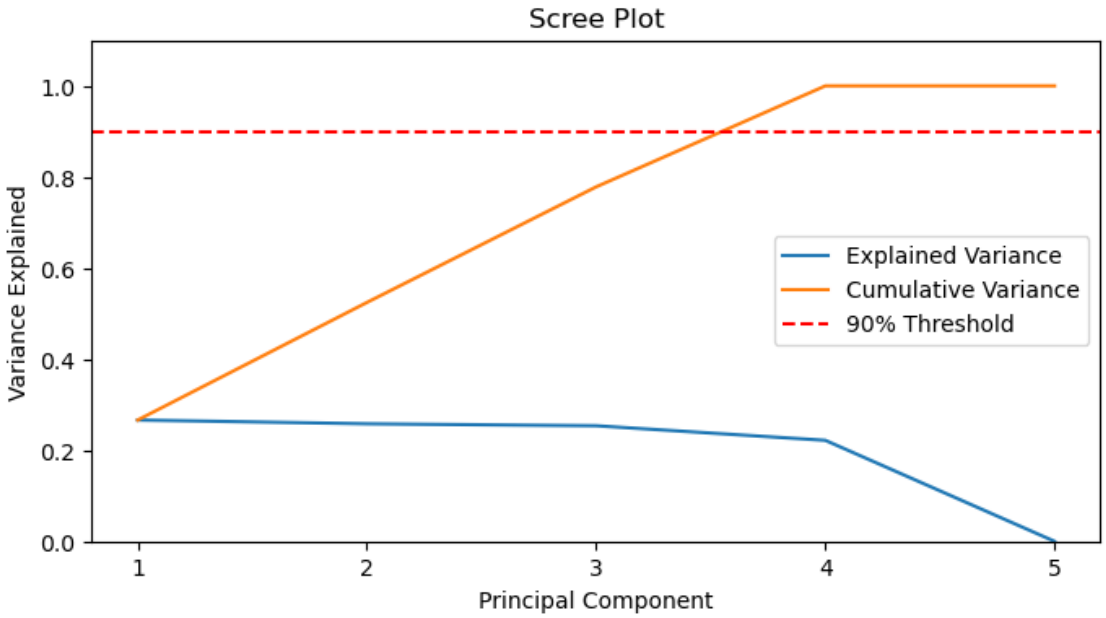
In the DMD heatmap, the most significant modes are likely to be those that show clear, consistent patterns across the state dimensions, however, no clear periodic patterns and sparsely populated dynamics are displayed with a lack of oscillatory components in the original DMD. After applying PCA, the modes that remain prominent after dimensionality reduction, are the most significant towards variance and influence on the system behaviour. Mode 2 State 1 is an example of a dominant mode which is exhibited and retained after PCA compression.

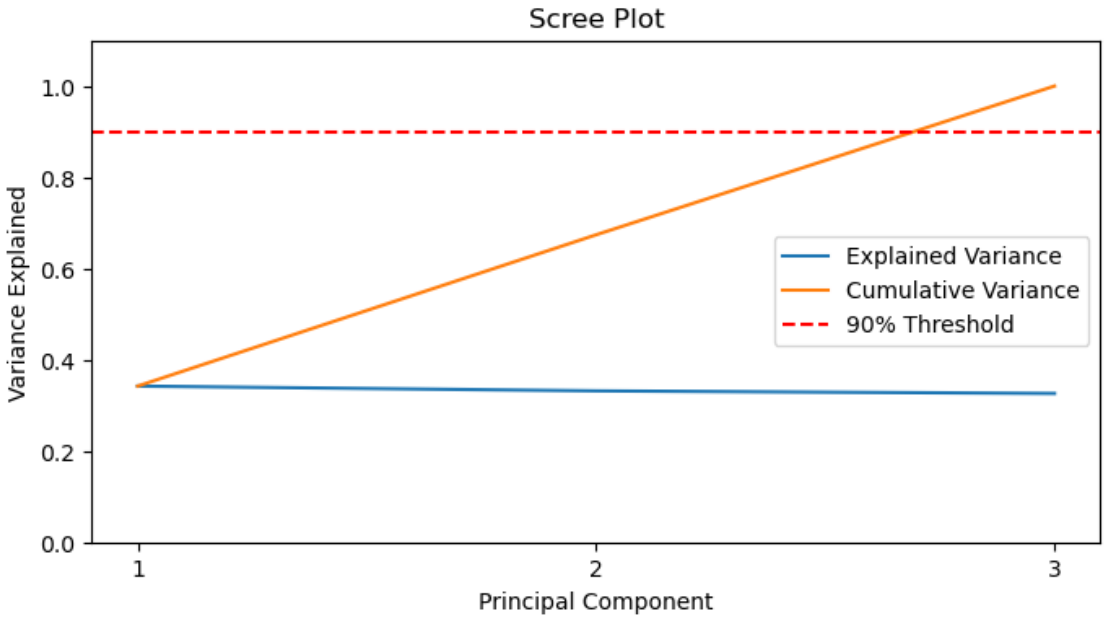
Scree Plot `compute_scree_plot()`

A scree plot was used to illustrate the explained variance of each principal component and to identify the most important components in the data. In SVD analysis, the R^2 parallels this method in validating variance as a function of a parameter. In a similar ideology, PCA is used to validate the number of components to compute to capture a threshold majority > 90 of variance for unsupervised analysis.

[ ]:
```
# Visualise variance explained using Scree plot
def compute_scree_plot(data, max_components=10):
    # Center data
    data_centered = data - np.mean(data, axis=0)

    # SVD
    U, S, Vt = np.linalg.svd(data_centered, full_matrices=False)

    # Explained variance
    explained_variance = (S ** 2) / (data.shape[0] - 1)
    total_variance = np.sum(explained_variance)
    explained_variance_ratio = explained_variance / total_variance  # Normalized␣
↪variance

    # Cumulative explained variance
    cumulative_variance = np.cumsum(explained_variance_ratio)

    # Scree plot
    components = np.arange(1, len(explained_variance_ratio) + 1)
    plt.figure(figsize=(8, 4))
    plt.plot(
        components[:max_components],
        explained_variance_ratio[:max_components], linestyle='-',␣
↪label='Explained Variance'
    )
    plt.plot(
        components[:max_components],
        cumulative_variance[:max_components], linestyle='-', label='Cumulative␣
↪Variance'
    )
    plt.axhline(y=0.9, color='r', linestyle='--', label='90% Threshold')
    plt.xlabel('Principal Component')
    plt.ylabel('Variance Explained')
    plt.title('Scree Plot')
    plt.ylim(0,1.1)
    plt.xticks(components[:max_components])
    plt.legend(loc ="center right")
    plt.show()

    return explained_variance_ratio
```

[2274]:
```
# Visualising Scree plot
evr = compute_scree_plot(DMD_matrix, max_components=10)
evr_red = compute_scree_plot(DMD_matrix_reduced, max_components=10)

print(evr)
print(evr_red)
```






```
(array([2.66355575e-01, 2.58128116e-01, 2.53758303e-01, 2.21758006e-01,
       1.47342832e-34]), array([0.26635557, 0.52448369, 0.77824199, 1.        ,
1.        ]))
(array([0.34219337, 0.33169822, 0.32610841]), array([0.34219337, 0.67389159, 1.
]))
```

The Scree plot shows that the first four modes explain almost all the variance (over 78%), with

the remaining modes contributing negligibly. In contrast, the PCA+DMD plot has a more gradual distribution of explained variance, with each of the first three modes contributing about 33%, suggesting a more even spread of variance across modes. This indicates that PCA has contributed towards balancing and redistributing the variance, likely reducing the dominance of a few modes in the DMD analysis.

1.6.5 4.4 Residual DMD

The following question applies Residual Dynamic Mode Decomposition (RDMD) to compute the necessary residuals used in the following analysis. Residual DMD is an extension on the DMD methods described earlier - focusing on numerical optimisation over identifying dominant modes as opposed to linear operator approximation.

The residual DMD acts to compute and minimise residuals, which are the error measurements between the actual system dynamics and the predicted system, mathematically expressed as

$$\min_A ||\text{res}||_F^2 = \min_A ||x' - Ax||_F^2$$

where x and Ax' are the current and discrete time reconstruction future snapshots respectively, and A the DMD operator. Recalling that for DMD, A is the approximation of the Koopman operator $\mathcal{K}$ in the eigenfunction subspace, the residual is a function of a specific eigenpair (λ, h) and can be shown as

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

where $h(x) = v^H \Phi(x)$ is an eigenvector, and v^H denotes the hermitian transpose. for which minimisation over all data points leads to the expansion

$$\text{res}(\lambda, h) \approx \frac{\sum_i^N ||v^H \Phi(y_i) - \lambda v^H \Phi(x_i)||^2}{\sum_i^N ||v^H \Phi(x_i)||^2}$$

leading to

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

where the matrices - $G = XX^T$ is the current (Gramian) covariance matrix - $H = YX^T$ is the cross-covariance matrix - $L = YY^T$ is the output covariance matrix

Residual DMD addresses the underlying linear mapping of conventional DMD methods, allowing for dynamic adaptation of the operator A to changes in system behaviour and robustness to non-linearities.

[1227]:
```
# Residual dmd computation for time series function
def res_dmd(G, H, L):

    # Compute eigenvalues
    eig_vals, eig_vecs = np.linalg.eig(np.linalg.inv(G) @ H)

    # Iterates for each eigenpair
    res_arr = []
    for i in range(len(eig_vals)):
        #Eigenvalue and eigenvector
```

```
        lambda_i = eig_vals[i]
        v = eig_vecs[:, i]

        # Normalise with G
        v = v / np.sqrt(v.conj().T @ G @ v)

        # Residual formula: (v^H(L - λH - λH^T + |λ|^2G)v) / (v^H G v)
        residual_numerator = (
            v.conj().T @ (L - lambda_i * H - np.conj(lambda_i) * H.T +␣
↪abs(lambda_i)**2 * G) @ v
        )
        residual_denominator = v.conj().T @ G @ v
        residual = np.abs(residual_numerator / residual_denominator)
        res_arr.append(residual)

    return res_arr, eig_vals, eig_vecs


# Reisidual dmd computation for entire data set function
def residual_dmd_full(data, delay, svd_rank, poly_order):
    res_arr = []
    evals_arr = []
    evecs_arr = []

    # Iterate for each bolt
    for bolt in data:
        exp_res_arr = []
        exp_evals_arr = []
        exp_evecs_arr = []

        # Iterae for each experiment
        for experiment in bolt:
            if len(experiment) >= delay + 1:

                # Delay embedding
                XX, YY = delay_embedding(experiment, delay)

                # Decorrelation
                U, S, Ut = decorr(XX)

                # Polynomial basis
                XX_proj = np.transpose(U[:, :svd_rank]) @ XX
                YY_proj = np.transpose(U[:, :svd_rank]) @ YY
                XXhat = polyeval(XX_proj, 1, poly_order)
                YYhat = polyeval(YY_proj, 1, poly_order)

                # Compute G, H, and L
```

```
                G = XXhat @ XXhat.T  # XX^T
                H = YYhat @ XXhat.T  # YX^T
                L = YYhat @ YYhat.T  # YY^T

                res, evals, evecs = res_dmd(G, H, L)

                exp_res_arr.append(res)
                exp_evals_arr.append(evals)
                exp_evecs_arr.append(evecs)
            else:
                print("Insufficient delay for embedding.")

        res_arr.append(exp_res_arr)
        evals_arr.append(exp_evals_arr)
        evecs_arr.append(exp_evecs_arr)

    return res_arr, evals_arr, evecs_arr
```

Residual Plots

Residuals were calculated for each trajectory within `data_train` with the parameters from the delay-embedded model to mimic the same model. These residuals are then sorted in ascending order, where it is then plotted on an Argand diagram along with its corresponding eigenvalue λ. These residuals aid in identifying the accuracy of mode extraction by highlighting discrepancies between the model and data. Insights gained can reveal the quality of the decomposition using the residuals' complex components i.e. the presence of noise, and the behaviour of the system's dynamics, such as damping or frequency separation.

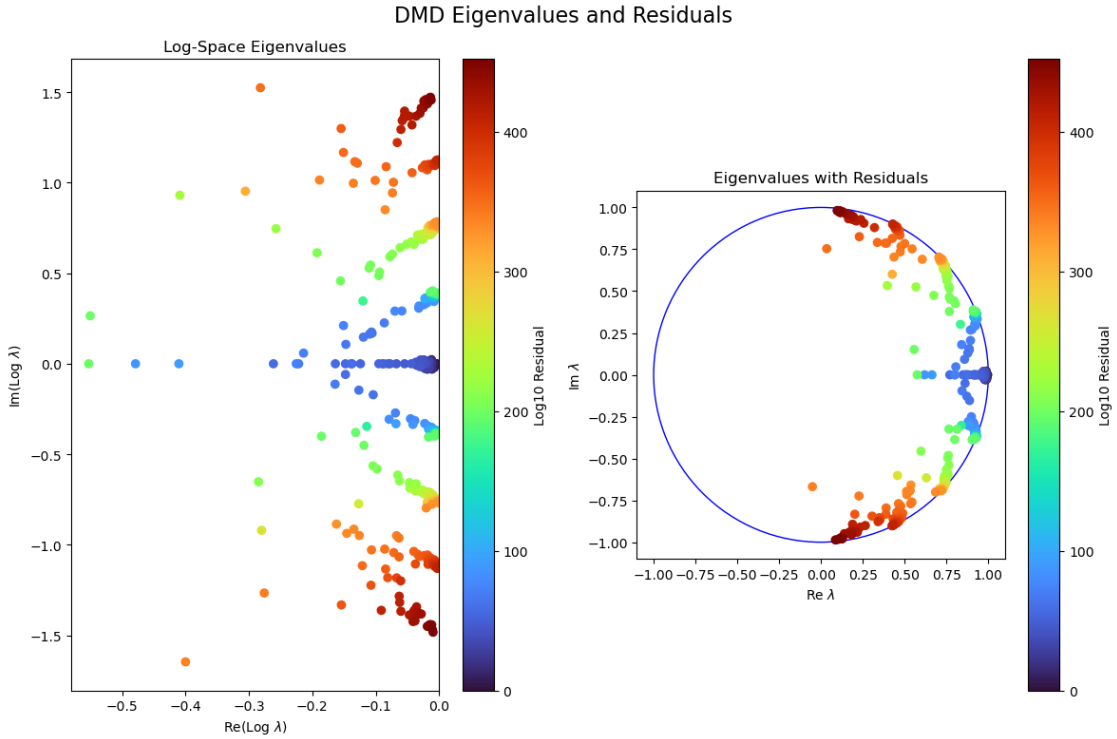
[1504]:
```
#Residuals
res_arr_train,evals_arr_train,evecs_arr_train = residual_dmd_full(data_train,␣
 ↪delay=optimum_delay, svd_rank = svd_rank_dmd, poly_order=1)

# Flatten
flat_residuals = []
flat_evals = []
flat_evecs = []

# Dimensionality fix
for bolt, exp_res, exp_evals, exp_evecs in zip(data_train, res_arr_train,␣
 ↪evals_arr_train, evecs_arr_train):
    for res, evals, evecs in zip(exp_res, exp_evals, exp_evecs):
        flat_residuals.extend(res)
        flat_evals.extend(evals)
        flat_evecs.extend(evecs)

# Numpy arrays
flat_residuals = np.array(flat_residuals)
flat_evals = np.array(flat_evals)
```

```
flat_evecs = np.array(flat_evecs)

# Sort arrays
sort_idx = np.argsort(flat_residuals)

sorted_res = flat_residuals[sort_idx]
sorted_evals = flat_evals[sort_idx]
sorted_evecs = flat_evecs[sort_idx]


print("Smallest Residuals:", sorted_res[:10])
print("Corresponding Eigenvalues:", sorted_evals[:10])
```

```
Smallest Residuals: [0.00125471 0.00199595 0.00212984 0.00528586 0.00543011
0.00724058
 0.00771023 0.00792642 0.00810652 0.00857645]
Corresponding Eigenvalues: [0.99551975+0.j 0.99563008+0.j 0.99511792+0.j
0.9917105 +0.j
 0.99723315+0.j 0.99622321+0.j 0.99532946+0.j 0.99517325+0.j
 0.99453342+0.j 0.99548104+0.j]
```

[1508]:
```
# Flatten residuals and corresponding eigenvalues across all bolts
all_res = np.hstack(sorted_res)
all_evals = np.hstack(sorted_evals)

log_evals = np.log(all_evals)


fig, (ax_log, ax_eig) = plt.subplots(1, 2, figsize=(12, 8))

# Scatter plot of Log Eigenvalues
map2 = ax_log.scatter(
    np.real(log_evals),
    np.imag(log_evals),
    c=np.arange(len(all_res)),
    cmap="turbo"
)

ax_log.set_xlabel('Re(Log $\\lambda$)')
ax_log.set_xlim(right = 0)
ax_log.set_ylabel('Im(Log $\\lambda$)')
ax_log.set_title("Log-Space Eigenvalues")
cbar_log = plt.colorbar(map2, ax=ax_log, label="Residual")

# Scatter plot of Eigenvalues on Unit Circle
circle = plt.Circle((0, 0), 1.0, color='b', fill=False)
ax_eig.add_patch(circle)
```

```
map1 = ax_eig.scatter(
    np.real(all_evals),
    np.imag(all_evals),
    c=np.arange(len(all_res)),
    cmap="turbo"
)
ax_eig.set_xlabel('Re $\\lambda$')
ax_eig.set_ylabel('Im $\\lambda$')
ax_eig.set_aspect("equal")
ax_eig.set_title("Eigenvalues with Residuals")
cbar_eig = plt.colorbar(map1, ax=ax_eig, label="Residual")

fig.suptitle("DMD Eigenvalues and Residuals", fontsize=16)
fig.tight_layout()
plt.show()
```



From the Argand diagram, the presence of residuals distributed along both axes suggests that the model captures certain dynamics, though a lack of clear concentration near the unit circle implies that the system may not be either purely oscillatory or stable. The spread along the real axis could indicate a non-negligible damping effect, while values along the imaginary axis are indicators of oscillatory behaviour with varying frequencies. The overall trend suggests that the system might exhibit a mix of decaying oscillations or a non-oscillatory trend.

However, the major unexpected trend is the lack of negative eigenvalues, which are indicators of non-dissipative properties in terms of dynamical behaviour, which is unexpected as the oscillations in Section 1 show clear decaying components. If the `svd_rank`, `poly_order`, and`delay` are fixed for this model - this may imply improper formulation of the `res_dmd()` function, potentially as consquence of poor calculations of the G, H, and L matrices, or through poor inverting conditions. A small regularisation term ϵ was attempted to mitigate this computational instability however did not lead to improvements in the residuals.

The concentration of low residual magnitudes near the origin suggests that the DMD decomposition is effectively capturing the dominant oscillatory modes of the system, which are likely characterized by neutral stability or under damping, where the eigenvalues have real parts close to zero. This indicates that the system's primary oscillations are well approximated by the model, leading to minimal error in those modes. However, potential issues to address are the issues with mode resolution or separation, especially if the system exhibits closely spaced frequencies or if the SVD rank used in the decomposition is too low, leading to an incomplete representation of higher-frequency dynamics. This behaviour suggests that the model is most accurate for capturing low-frequency oscillations, but additional refinement may be needed to better represent higher-frequency components and especially capturing nonlinear damping effects.

1.6.6 4.5 Bagging

Bagging is a method which leverages stochastic sampling to mitigate overfitting and validate over multiple experiments. In the course, two methods of bagging are noted - dense and sparse bagging. Dense Bagging refers to the process where each bagging iteration uses the full set of features, with no zero or missing entries in the data. In contrast, Sparse Bagging uses a subset of features, typically resulting in more zeros and missing values in the data. For the purposes of this analysis, dense bagging was considered to ensure all experiments and features are considered towards the model to ensure all underlying variances are captured, and the number of features is relatively low.

This loop performs dense bagging to compute the most robust eigenvalues across `n_iteration` bootstrap iterations from the input dataset. For each iteration, a bootstrap sample is generated by randomly selecting experiment indices (with replacement) for each bolt. The concatenated data is used to compute a reduced-order model $\tilde{A}$ using SVD.

Eigenvalues and their corresponding residuals are then calculated via a direct matrix formulation as opposed to individual time series analysis for computational efficiency

$$\text{res}(\lambda, h) = \frac{||\tilde{A}\hat{v} - \lambda_i \hat{v}||_F}{||\hat{v}||_F}$$

where $\hat{v}$ is the truncated eigenvector, projected through the reduced operator $\tilde{A}$

The eigenvalue with the smallest residual and a valid non-zero frequency is selected as the eigenvalues representative of the system's global dynamics, with their distributions to be explored.

[960]:
```
# Dense Bagging
#--------------------------#
n_iterations = 300
#--------------------------#

bagged_eigenvalues = []
```

```
n_bolts = WW_norm_train.shape[0]
n_experiments = WW_norm_train.shape[1]

#Iteration number
for iteration in range(n_iterations):
    print(f"Iteration: {iteration}")

    # Initialise container for sample
    bootstrap_sample = []

    # All bolts
    for bolt_index in range(n_bolts):

        # Random sample with replacement
        sampled_indices = np.random.choice(n_experiments, size=n_experiments,␣
↪replace=True)

        # Sampled experiments
        sampled_experiments = []
        for exp_idx in sampled_indices:
            sampled_experiments.append(WW_norm_train[bolt_index, exp_idx])

        bootstrap_sample.append(np.array(sampled_experiments))

    bootstrap_sample = np.array(bootstrap_sample)


    # Concatenate linear models
    W_concat = np.hstack([np.vstack(bolt_sample) for bolt_sample in␣
↪bootstrap_sample])
    # Polynomial evaluation (not used)
    W_concat = polyeval(W_concat, min_order=1, max_order=poly_eval_dmd)

    # SVD
    U, S, Vt = np.linalg.svd(W_concat, full_matrices=False)
    truncation_rank = min(W_concat.shape[0], W_concat.shape[1] // 2)
    U_k = U[:, :truncation_rank]
    S_k = np.diag(S[:truncation_rank])
    V_k = Vt[:truncation_rank, :]

    # DMD operator in reduced space A_tilde
    A_tilde = U_k.T @ W_concat @ V_k.T @ np.linalg.inv(S_k)

    # Eigenvalues and Eigenvectors
    eig_vals, eig_vecs = np.linalg.eig(A_tilde)

    # Residual computation
```

```
    residuals = []
    for i in range(len(eig_vals)):
        v_reduced = eig_vecs[:, i]
        lambda_i = eig_vals[i]
        residual = np.linalg.norm(A_tilde @ v_reduced - lambda_i * v_reduced,␣
↪ord=2) / np.linalg.norm(v_reduced, ord=2)
        residuals.append(residual)

    # Smallest Eigenvalue
    frequencies = np.imag(np.log(eig_vals)) / (2 * np.pi)
    valid_idx = [i for i, freq in enumerate(frequencies) if abs(freq) > 1e-18]

    if valid_idx:
        selected_idx = valid_idx[np.argmin([residuals[i] for i in valid_idx])]
        bagged_eigenvalues.append(eig_vals[selected_idx])

bagged_eigenvalues = np.array(bagged_eigenvalues)
```

```
Iteration: 0
Iteration: 1
Iteration: 2
Iteration: 3
Iteration: 4
Iteration: 5
Iteration: 6
Iteration: 7
Iteration: 8
Iteration: 9
Iteration: 10
Iteration: 11
Iteration: 12
Iteration: 13
Iteration: 14
Iteration: 15
Iteration: 16
Iteration: 17
Iteration: 18
Iteration: 19
Iteration: 20
Iteration: 21
Iteration: 22
Iteration: 23
Iteration: 24
Iteration: 25
Iteration: 26
Iteration: 27
Iteration: 28
Iteration: 29
```

```
Iteration: 30
Iteration: 31
Iteration: 32
Iteration: 33
Iteration: 34
Iteration: 35
Iteration: 36
Iteration: 37
Iteration: 38
Iteration: 39
Iteration: 40
Iteration: 41
Iteration: 42
Iteration: 43
Iteration: 44
Iteration: 45
Iteration: 46
Iteration: 47
Iteration: 48
Iteration: 49
Iteration: 50
Iteration: 51
Iteration: 52
Iteration: 53
Iteration: 54
Iteration: 55
Iteration: 56
Iteration: 57
Iteration: 58
Iteration: 59
Iteration: 60
Iteration: 61
Iteration: 62
Iteration: 63
Iteration: 64
Iteration: 65
Iteration: 66
Iteration: 67
Iteration: 68
Iteration: 69
Iteration: 70
Iteration: 71
Iteration: 72
Iteration: 73
Iteration: 74
Iteration: 75
Iteration: 76
Iteration: 77
```

```
Iteration: 78
Iteration: 79
Iteration: 80
Iteration: 81
Iteration: 82
Iteration: 83
Iteration: 84
Iteration: 85
Iteration: 86
Iteration: 87
Iteration: 88
Iteration: 89
Iteration: 90
Iteration: 91
Iteration: 92
Iteration: 93
Iteration: 94
Iteration: 95
Iteration: 96
Iteration: 97
Iteration: 98
Iteration: 99
Iteration: 100
Iteration: 101
Iteration: 102
Iteration: 103
Iteration: 104
Iteration: 105
Iteration: 106
Iteration: 107
Iteration: 108
Iteration: 109
Iteration: 110
Iteration: 111
Iteration: 112
Iteration: 113
Iteration: 114
Iteration: 115
Iteration: 116
Iteration: 117
Iteration: 118
Iteration: 119
Iteration: 120
Iteration: 121
Iteration: 122
Iteration: 123
Iteration: 124
Iteration: 125
```

```
Iteration: 126
Iteration: 127
Iteration: 128
Iteration: 129
Iteration: 130
Iteration: 131
Iteration: 132
Iteration: 133
Iteration: 134
Iteration: 135
Iteration: 136
Iteration: 137
Iteration: 138
Iteration: 139
Iteration: 140
Iteration: 141
Iteration: 142
Iteration: 143
Iteration: 144
Iteration: 145
Iteration: 146
Iteration: 147
Iteration: 148
Iteration: 149
Iteration: 150
Iteration: 151
Iteration: 152
Iteration: 153
Iteration: 154
Iteration: 155
Iteration: 156
Iteration: 157
Iteration: 158
Iteration: 159
Iteration: 160
Iteration: 161
Iteration: 162
Iteration: 163
Iteration: 164
Iteration: 165
Iteration: 166
Iteration: 167
Iteration: 168
Iteration: 169
Iteration: 170
Iteration: 171
Iteration: 172
Iteration: 173
```

```
Iteration: 174
Iteration: 175
Iteration: 176
Iteration: 177
Iteration: 178
Iteration: 179
Iteration: 180
Iteration: 181
Iteration: 182
Iteration: 183
Iteration: 184
Iteration: 185
Iteration: 186
Iteration: 187
Iteration: 188
Iteration: 189
Iteration: 190
Iteration: 191
Iteration: 192
Iteration: 193
Iteration: 194
Iteration: 195
Iteration: 196
Iteration: 197
Iteration: 198
Iteration: 199
Iteration: 200
Iteration: 201
Iteration: 202
Iteration: 203
Iteration: 204
Iteration: 205
Iteration: 206
Iteration: 207
Iteration: 208
Iteration: 209
Iteration: 210
Iteration: 211
Iteration: 212
Iteration: 213
Iteration: 214
Iteration: 215
Iteration: 216
Iteration: 217
Iteration: 218
Iteration: 219
Iteration: 220
Iteration: 221
```

```
Iteration: 222
Iteration: 223
Iteration: 224
Iteration: 225
Iteration: 226
Iteration: 227
Iteration: 228
Iteration: 229
Iteration: 230
Iteration: 231
Iteration: 232
Iteration: 233
Iteration: 234
Iteration: 235
Iteration: 236
Iteration: 237
Iteration: 238
Iteration: 239
Iteration: 240
Iteration: 241
Iteration: 242
Iteration: 243
Iteration: 244
Iteration: 245
Iteration: 246
Iteration: 247
Iteration: 248
Iteration: 249
Iteration: 250
Iteration: 251
Iteration: 252
Iteration: 253
Iteration: 254
Iteration: 255
Iteration: 256
Iteration: 257
Iteration: 258
Iteration: 259
Iteration: 260
Iteration: 261
Iteration: 262
Iteration: 263
Iteration: 264
Iteration: 265
Iteration: 266
Iteration: 267
Iteration: 268
Iteration: 269
```

```
Iteration: 270
Iteration: 271
Iteration: 272
Iteration: 273
Iteration: 274
Iteration: 275
Iteration: 276
Iteration: 277
Iteration: 278
Iteration: 279
Iteration: 280
Iteration: 281
Iteration: 282
Iteration: 283
Iteration: 284
Iteration: 285
Iteration: 286
Iteration: 287
Iteration: 288
Iteration: 289
Iteration: 290
Iteration: 291
Iteration: 292
Iteration: 293
Iteration: 294
Iteration: 295
Iteration: 296
Iteration: 297
Iteration: 298
Iteration: 299
```

Gaussian Distribution Plots

A Normal (Gaussian) distribution;

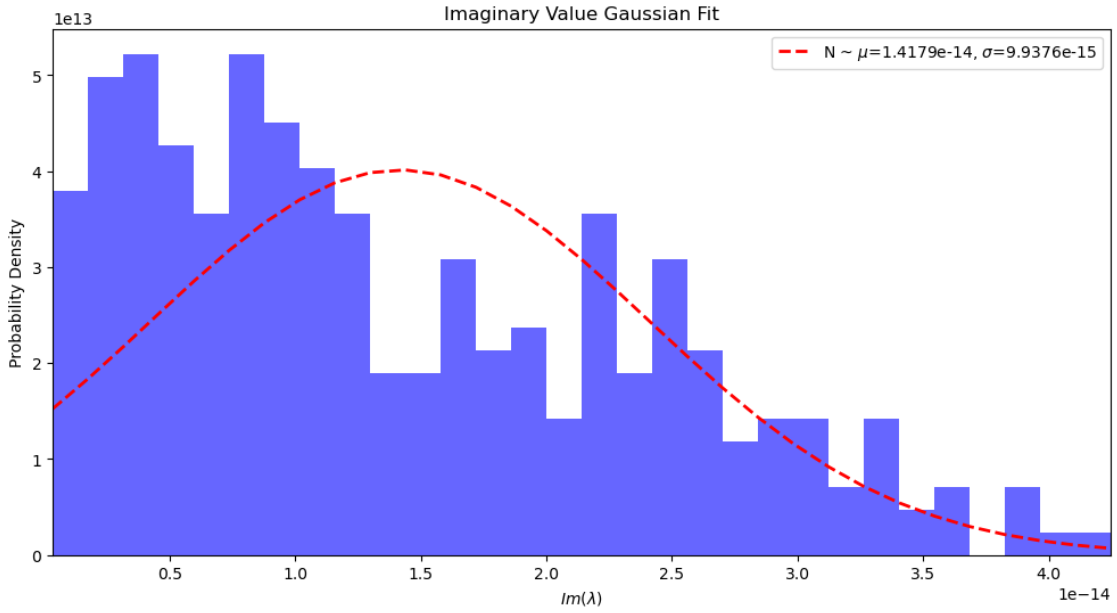
$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{(x-\mu)^2}{2\sigma^2}\right)$$

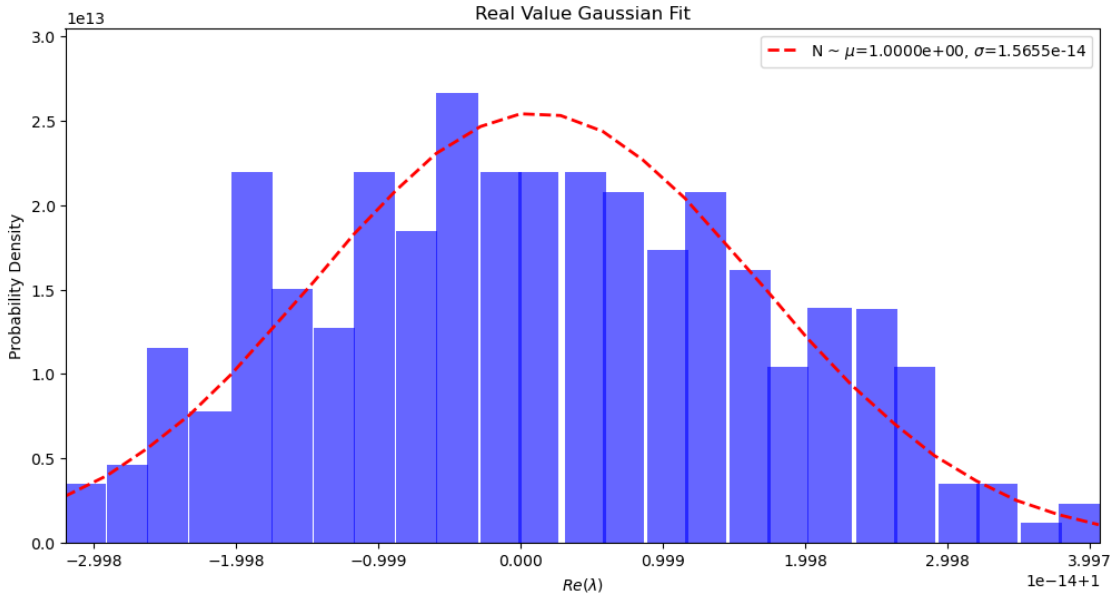
was fitted to the smallest residual eigenvalues for each bagging iteration. The plots are visualised as follows; for each complex basis and an overall contour plot. These visualisations were used to illustrate residual eigenvalue distributions across multiple bagging iterations, providing insight into the consistency and variability of the eigenvalue distribution for each complex basis.

A Gaussian distribution was used with the assumption of random, independently sampled residuals with a characteristic zero mean and constant variance.

[1991]:
```
# Extract components
real_values = np.real(bagged_eigenvalues)
imag_values = np.imag(bagged_eigenvalues)
```

```
# Fit Gaussian distribution
mu_imag, sigma_imag = norm.fit(imag_values, method="MLE")
mu_real, sigma_real = norm.fit(real_values, method="MLE")

# Visualisation of Real Value distribution
plt.figure(figsize=(12, 6))
n, bins, _ = plt.hist(
    imag_values,
    bins=30,
    density=True,
    alpha=0.6,
    color='blue',
)

y_imag = norm.pdf(bins, mu_imag, sigma_imag)
plt.plot(bins, y_imag, 'r--', linewidth=2, label=f"N ~ $\\mu$={mu_imag:.4e},␣
 ↪$\\sigma$={sigma_imag:.4e}")
plt.xlim(bins.min(), bins.max())
plt.xlabel(f'$Im(\\lambda)$')
plt.ylabel('Probability Density')
plt.title('Imaginary Value Gaussian Fit')
plt.legend()
plt.show()

# Visualisation of Imaginary Value distribution
plt.figure(figsize=(12, 6))
n, bins, _ = plt.hist(
    real_values,
    bins=25,
    density=True,
    alpha=0.6,
    color='blue',
)

y_real = norm.pdf(bins, mu_real, sigma_real)
plt.plot(bins, y_real, 'r--', linewidth=2, label=f"N ~ $\\mu$={mu_real:.4e},␣
 ↪$\\sigma$={sigma_real:.4e}")

plt.xlabel(f'$Re(\\lambda)$')
plt.ylabel('Probability Density')
plt.ylim(0, max(y_real) * 1.2)
plt.title('Real Value Gaussian Fit')
plt.legend()
plt.xlim(bins.min(), bins.max())
plt.show()
```





Distribution Statistics

Calculating the mean and covariance matrices

[1555]:
```
#Covariance
bagged_eigenvalues = np.array(bagged_eigenvalues)

# Separate the real and imaginary parts
```

```
real_parts = np.real(bagged_eigenvalues)
imag_parts = np.imag(bagged_eigenvalues)

# Mean
mean_eigenvalue = np.mean(bagged_eigenvalues)
mean_real = np.mean(real_parts)
mean_imag = np.mean(imag_parts)

# Mean vector
mean_vector = np.array([mean_real, mean_imag])

# Covariance
real_imag_array = np.vstack([real_parts, imag_parts])
covariance_matrix = np.cov(real_imag_array)


print(f"Mean Eigenvalue: {mean_eigenvalue}")
print(f"Mean Eigenvector: {mean_vector}")
print("Covariance Matrix:")
print(covariance_matrix)
```

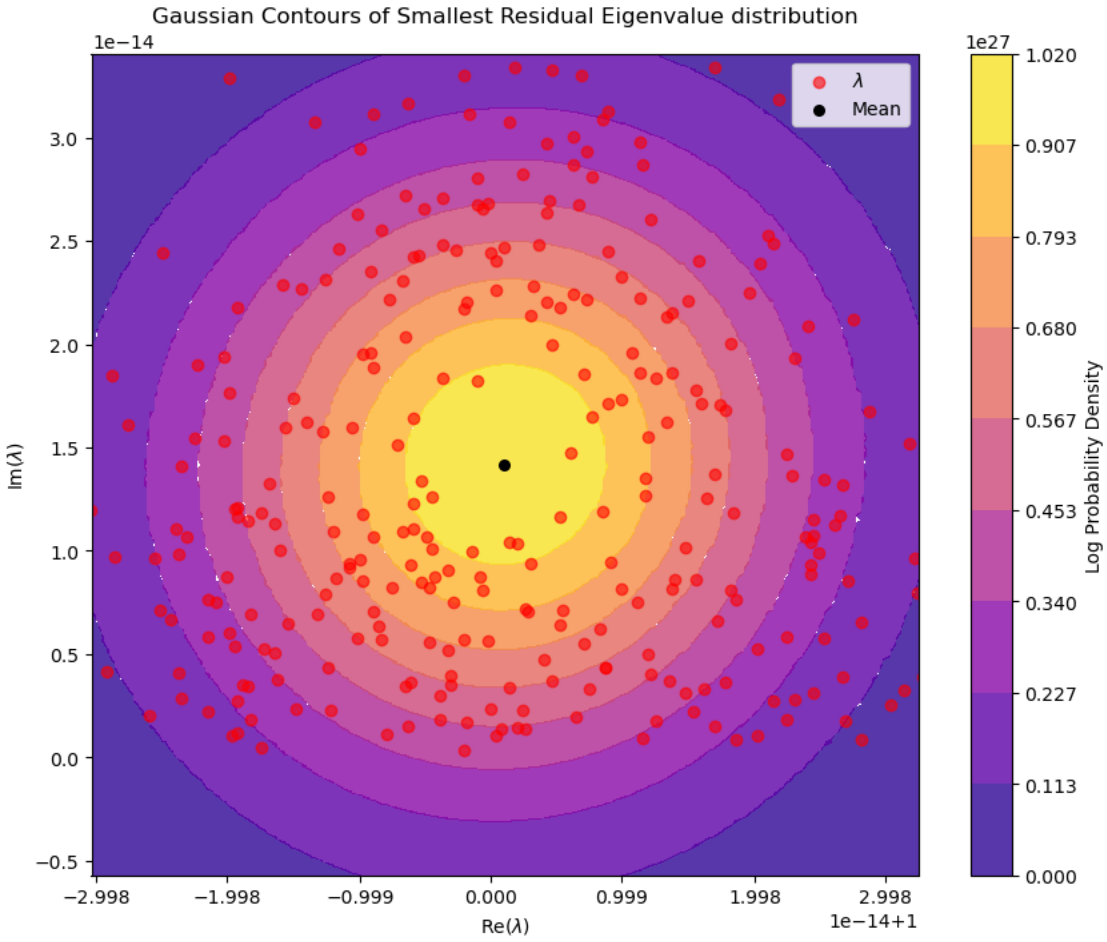
```
Mean Eigenvalue: (1.0000000000000013+1.41785701710148e-14j)
Mean Eigenvector: [1.00000000e+00 1.41785702e-14]
Covariance Matrix:
[[2.45888389e-28 5.00105787e-30]
 [5.00105787e-30 9.90866080e-29]]
```

The mean value is ≈ 1, suggesting the associated mode represented by the eigenvector corresponding to the smallest residual corresponds to a stable and neutral mode with no oscillatory response - which is unusual for the vibrating system. The small covariance values also suggest the variance of this dominant mode across all iterations remains constant. As the data was previously globally normalised, this is an observation which was to be expected.

The smallest residual associated with this eigenvalue and eigenvector indicates that the system's dynamics are well approximated by the corresponding eigenvalue-eigenvector pair, even in the reduced space. Although PCA analysis from earlier shows the system exhibits shared variance over the first 4 modes, the above analysis suggests otherwise with a single dominant stable mode.

[1598]:
```
# Visualisation on contour Gaussian distribution
scale_factor = 10

std_dev_real = np.std(real_parts)
std_dev_imag = np.std(imag_parts)
x_min, x_max = mean_real - scale_factor * std_dev_real, mean_real + scale_factor␣
 ↪* std_dev_real
y_min, y_max = mean_imag - scale_factor * std_dev_imag, mean_imag + scale_factor␣
 ↪* std_dev_imag
```

```
x, y = np.linspace(x_min, x_max, 1000), np.linspace(y_min, y_max, 1000)
X, Y = np.meshgrid(x, y)

# Multivariate PDF
pos = np.dstack((X, Y))
pdf = multivariate_normal(mean=mean_vector, cov=covariance_matrix).pdf(pos)

# Clip PDF to avoid issues with very small values
pdf = np.clip(pdf, 1e-10, None)

# Log-transform the PDF for better visualization

# Define contour levels
levels = np.linspace(pdf.min(), pdf.max(), 10)  # Adjust the number of levels

# Scaling
zoom_factor_real = 2
zoom_factor_imag = 2
x_min_zoom, x_max_zoom = mean_real - zoom_factor_real * std_dev_real, mean_real␣
↪+ zoom_factor_real * std_dev_real
y_min_zoom, y_max_zoom = mean_imag - zoom_factor_imag * std_dev_imag, mean_imag␣
↪+ zoom_factor_imag * std_dev_imag
plt.figure(figsize=(10, 8))

# Filled contour
filled_contour = plt.contourf(X, Y, pdf, levels=levels, cmap="plasma", alpha=0.8)
plt.colorbar(filled_contour, label="Log PDF")

# Line contour
contour = plt.contour(X, Y, pdf_log_scaled, levels=levels, colors="white",␣
↪linestyles="solid", linewidths=1.5)
plt.clabel(contour, inline=True, fontsize=8, fmt="%.2f")

# Scatter points
plt.scatter(real_parts, imag_parts, alpha=0.6, color="red", label="$\\lambda$")
plt.scatter(mean_real, mean_imag, s=30, color="black", label="Mean",␣
↪edgecolor="black")

plt.xlim(x_min_zoom, x_max_zoom)
plt.ylim(y_min_zoom, y_max_zoom)
plt.xlabel(f'Re($\\lambda$)')
plt.ylabel(f'Im($\\lambda$)')
plt.title("Gaussian Contours of Smallest Residual Eigenvalue distribution")
plt.legend()
plt.show()
```



The Gaussian fit to the imaginary and real components of the residuals provides valuable insight into the statistical behaviour of the system's eigenvalue residuals. It can be observed that the imaginary values do not fall below zero, which is consistent with the physical constraints imposed by energy-dissipating systems. This is because the imaginary part of eigenvalues is attributed to oscillatory action, and negative imaginary values would indicate unstable, unphysical solutions that do not converge to a steady state.

The real values, however, support the assumption of random sampling across bagging iterations - showing signs of convergence towards the Central Limit Theorem. The physical implications on the uncertainty of the vibration frequency corresponding to the dynamic mode in Problem 3 are that residuals are influenced by small independent parameters in experimental variability, resulting in small deviations. The uncertainties are also considered relatively small, implying residuals all possess random experimental fluctuations rather than the DMD or the system model being fundamentally flawed.

1.6.7 References

[1] Szalai, R. (n.d.). Data Driven Physical Modelling - SEMTM0007. University of Bristol. https://www.ole.bris.ac.uk/ultra/courses/

[2] Goldstein, H. (2002). Classical Mechanics (3rd ed.). Addison-Wesley
[3] M. J. Ablowitz, H. Segur, "Solitons and the Inverse Scattering Transform," (1981)
[4] Bishop, C. M. (2006). Pattern Recognition and Machine Learning. Springer
[5] Kaiser, H. F. (1960). The Application of Electronic Computers to Factor Analysis. Educational and Psychological Measurement, 20(1), 141-151
[6] Spectral density (2024) Wikipedia. Available at: https://en.wikipedia.org/wiki/Spectral_density
[7] Sauer, T., Yorke, J. A., & Casdagli, M. (1991). "Embedology." Journal of Statistical Physics, 65(3-4), 579-616
[8] Trefethen, L. N., & Bau, D. III. (1997). Numerical Linear Algebra. Society for Industrial and Applied Mathematics (SIAM)
[9] Takens, F. (1981). Detecting strange attractors in turbulence. Physica D: Nonlinear Phenomena, 20(2-3), 303-317. https://doi.org/10.1016/0167-2789(81)90094-2
[10] Seber, G. A. F., & Lee, A. J. (2003). Linear Regression Analysis (2nd ed.). Wiley-Interscience
[11] Eckhart, C., & Young, C. (1936). The approximation of one matrix by another of lower rank. Psychometrika, 1(3), 211-218. doi:10.1007/BF02288367
[12] Kutz, J. N., Brunton, S. L., Proctor, J. L., & Kallus, N. L. (2016). Dynamic Mode Decomposition: A Review of the Methodology and Applications. SIAM Review, 58(3), 485-529. https://doi.org/10.1137/15M1021037
[13] Hastie, T., Tibshirani, R., & Friedman, J. (2009). The Elements of Statistical Learning: Data Mining, Inference, and Prediction (2nd ed.). Springer