2143062-DDPM-Part1

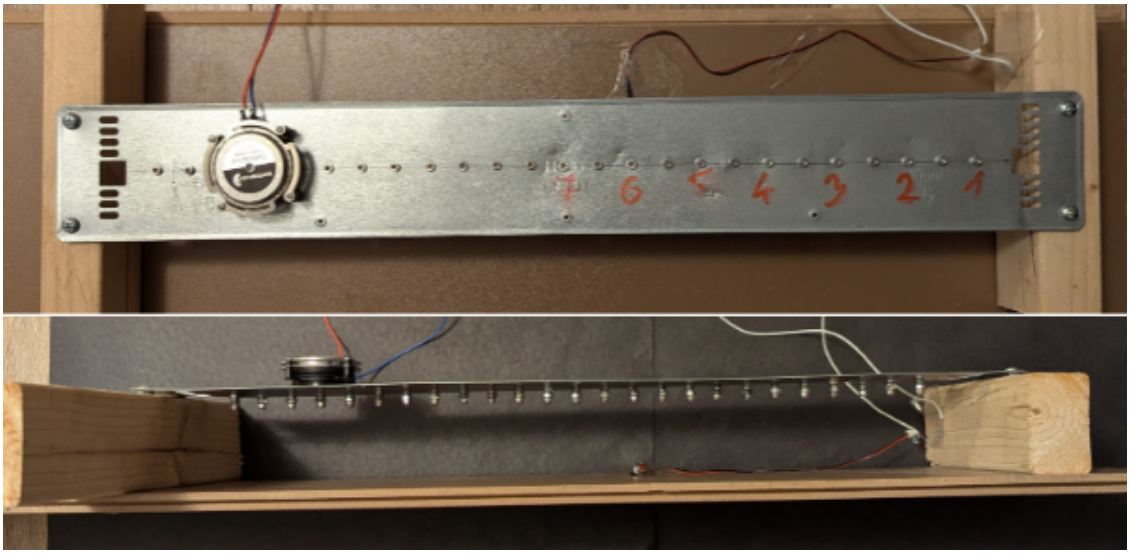
November 28, 2024

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

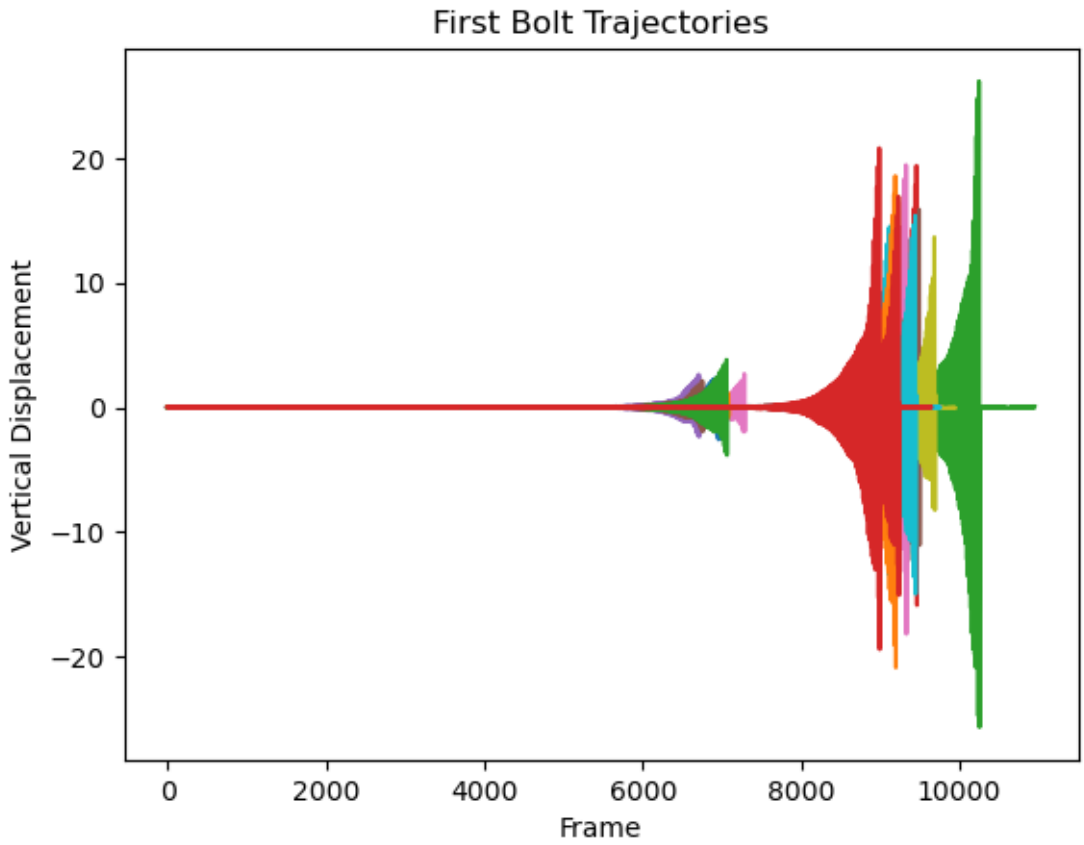
The following steps to preprocess data included:
1. Reversing to ensure chronological order 2. Removing the initial steady state 3. Eliminating decayed components 4. Normalisation and Scaling

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.

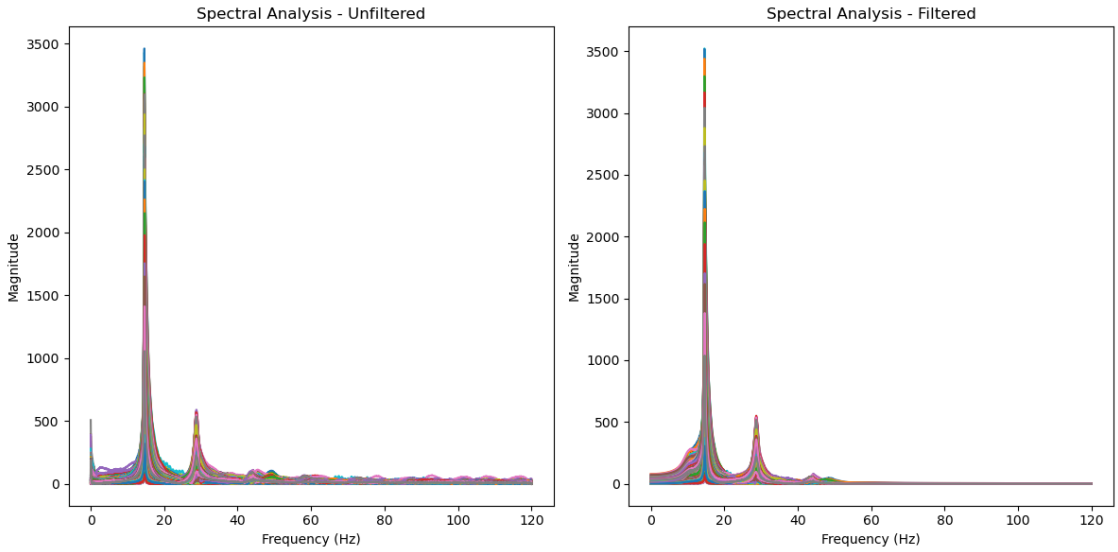
The Hilbert transform was applied to identify the position where vibration data decays into a steady state [3]. This was chosen as it consistently outperformed other truncation methods such as window gradient filtering and exponential flattening, likely as Hilbert analysis inherently tailors to frequency 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()
```

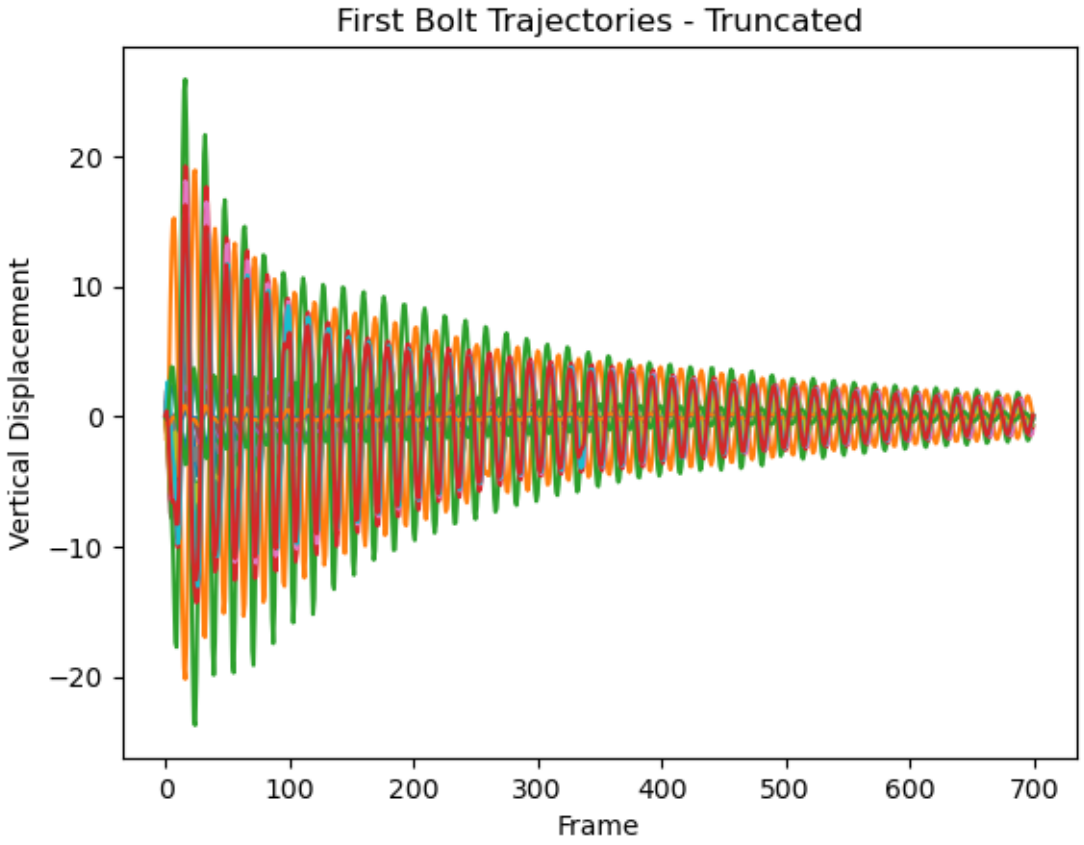


This signal was then passed into the `truncate_end()` function, which compares a sliding window of the experimental trajectory amplitude against the analytic signal envelope generated by the Hilbert transform. The following parameters were adjusted to ensure dynamic components were isolated. An arbitrary 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:

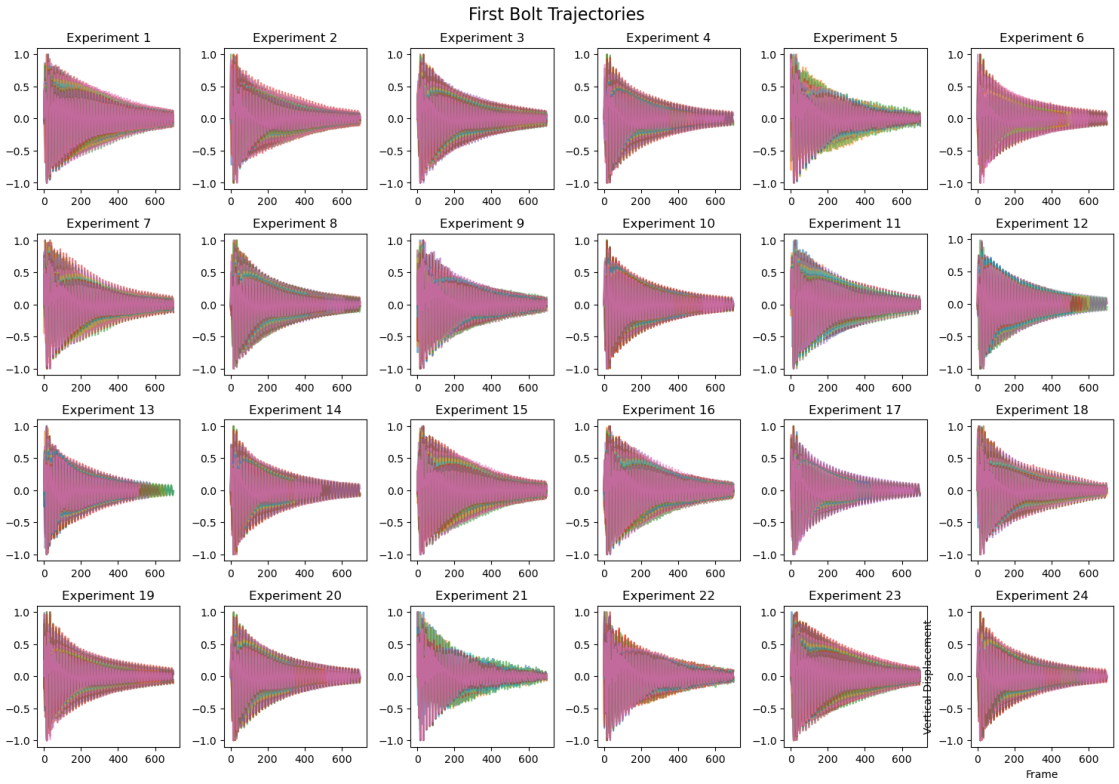
$$\text{RMS} = \sqrt{\frac{1}{N}\sum_{i}^{N} s_i^2}$$

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()
```

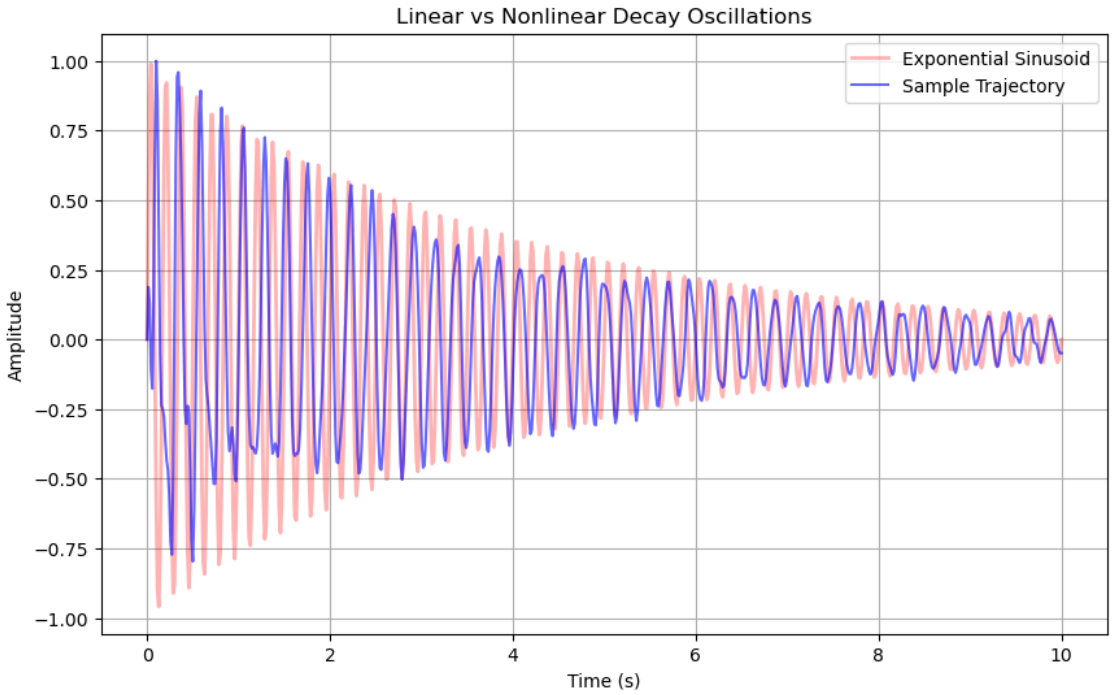


The grid of plots shows the comprehensive trajectory data for each experiment after preprocessing, aggregated into the list `data_norm`.

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()
```



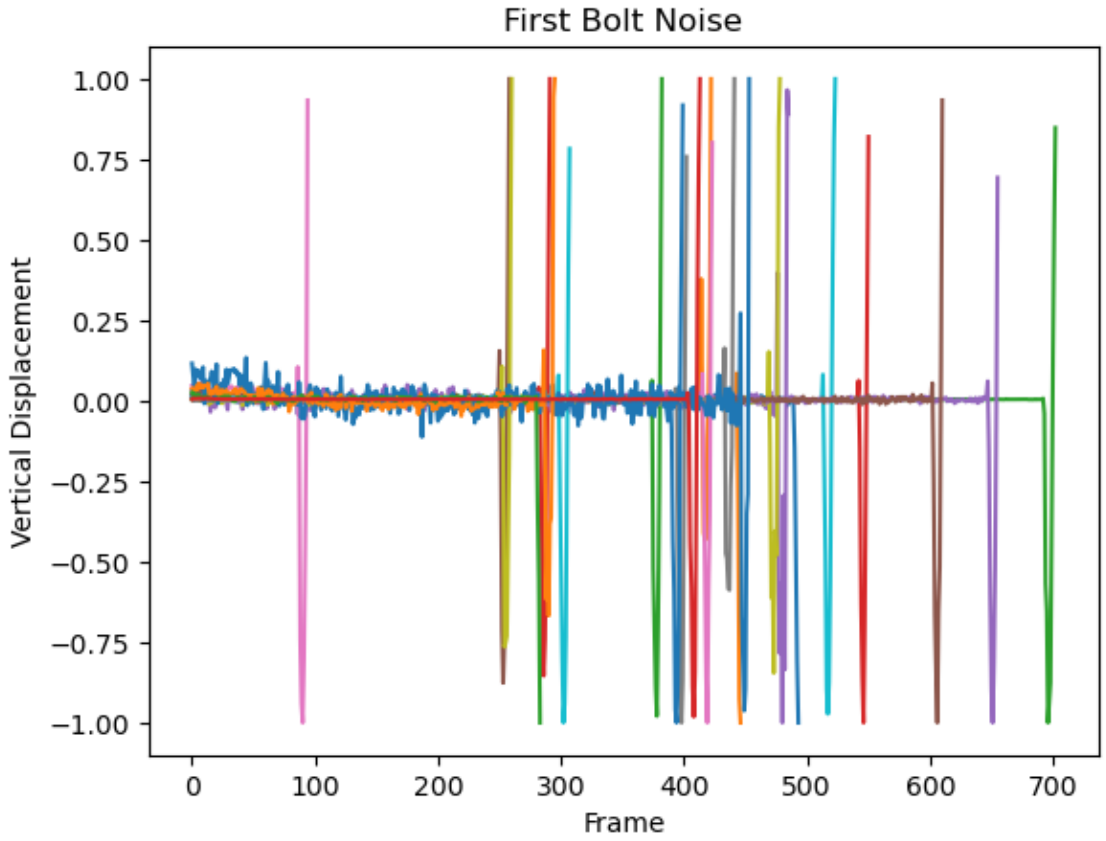
1.3.2 1.2 Noise Power Spectrum

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

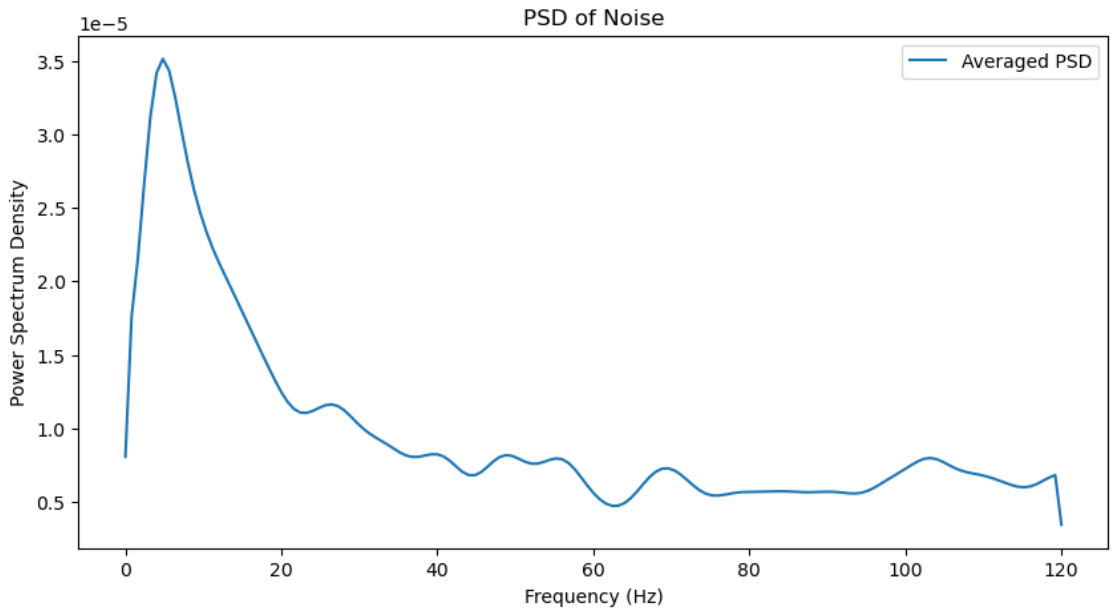
$$\text{PSD}(f) = \frac{1}{M}\sum_{i}^{M}\frac{|FFT(x)|^2}{L}$$

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.

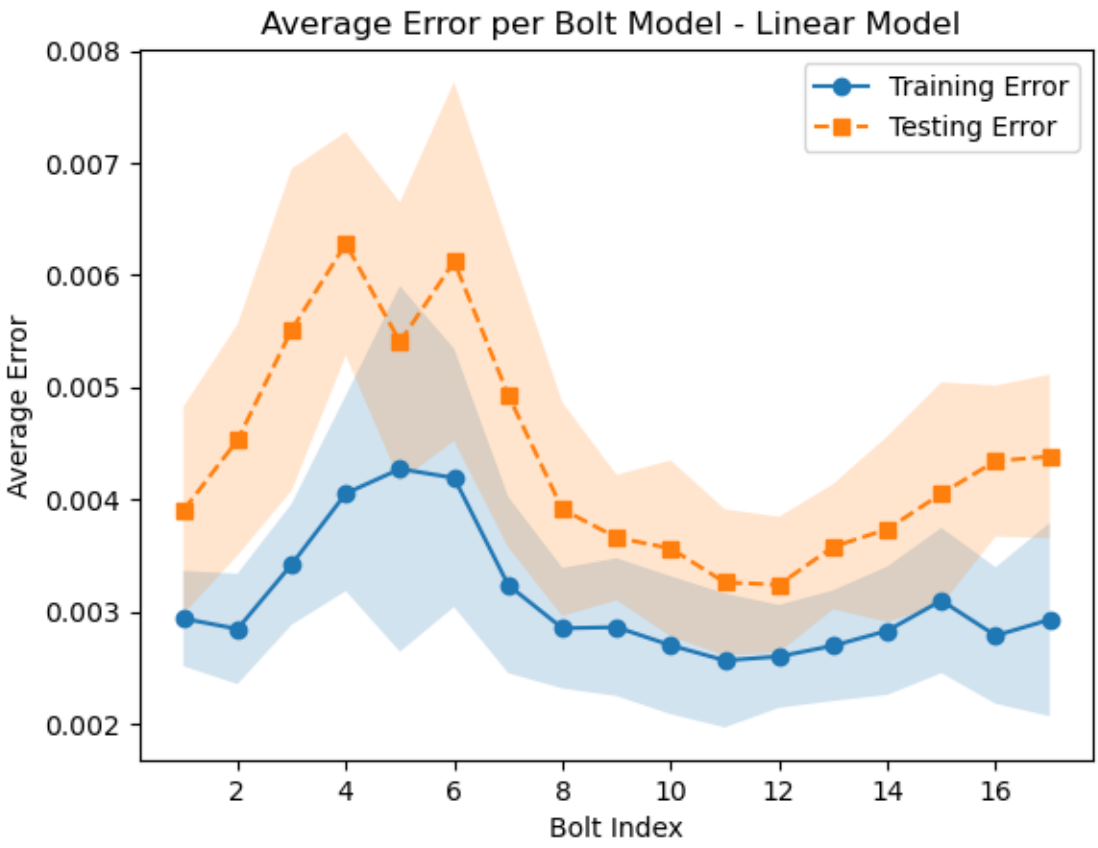
[888]:
```
#Training model
#--------------------------#
svd_rank = 5
poly_order = 1 #linear model
delay = 20
#--------------------------#

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

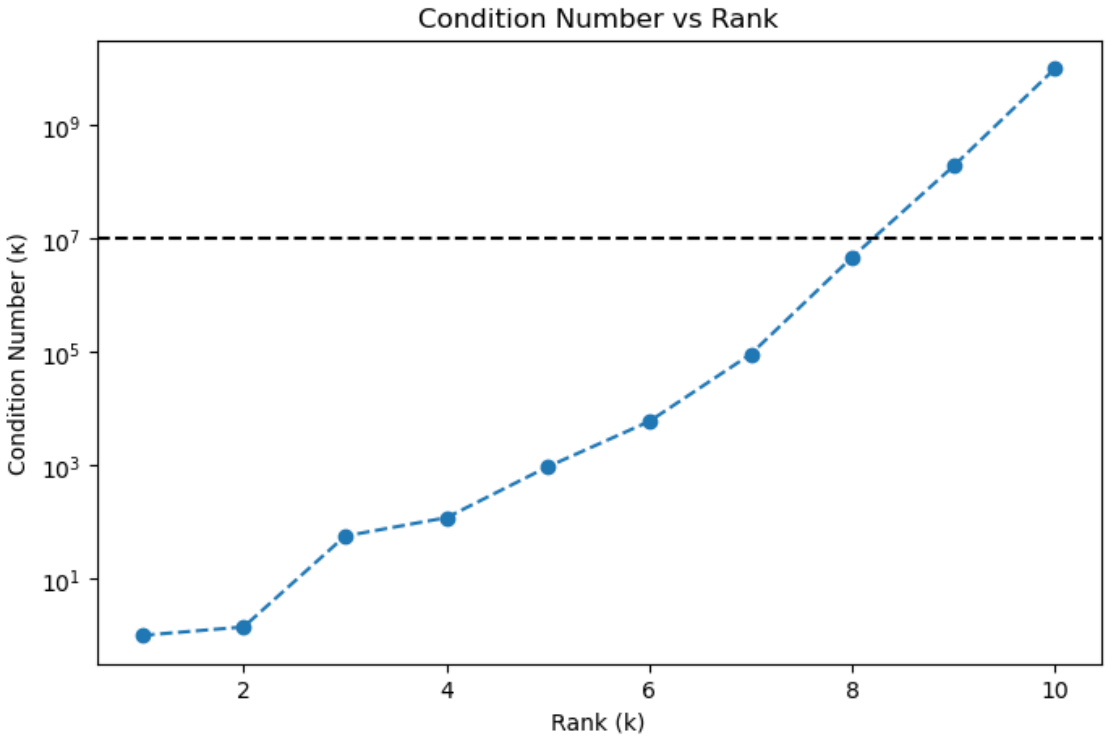
Condition Number Evaluation

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

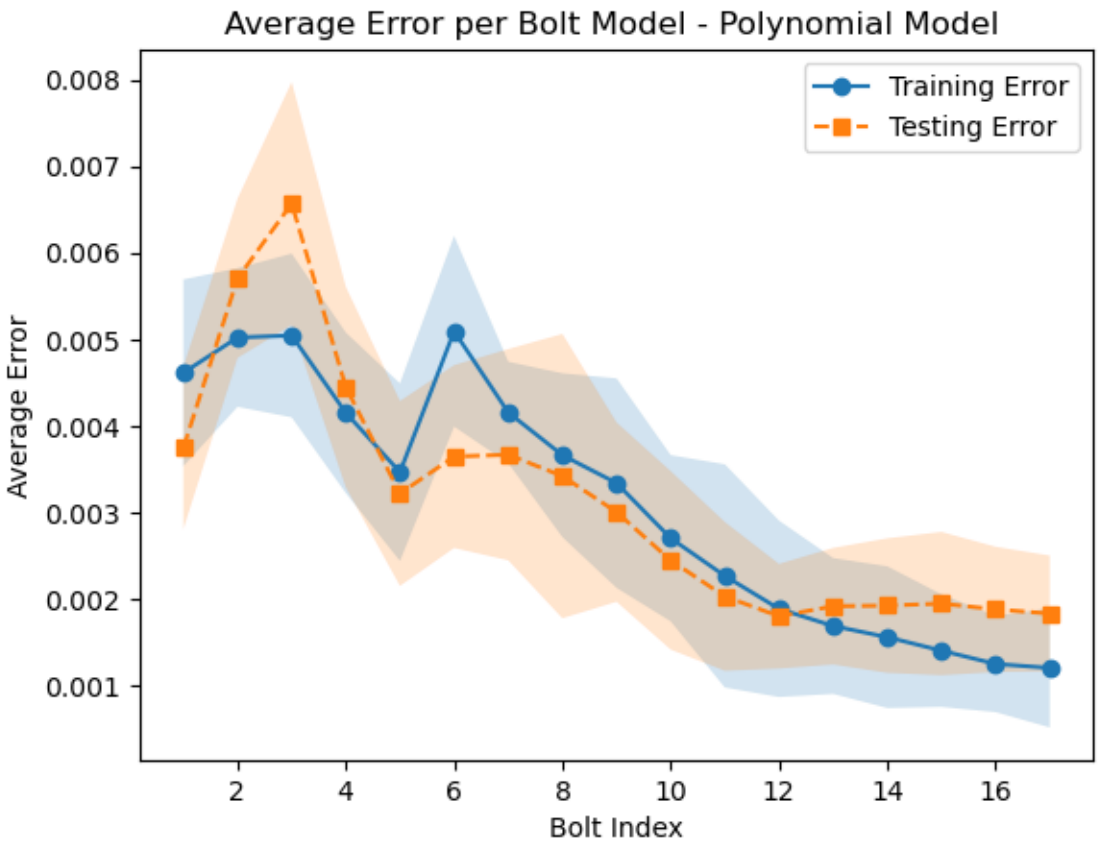
$$\tilde{W} = \hat{Y}(\Phi(\hat{X}))^T(\Phi(\hat{X})(\Phi(\hat{X}))^T)^\dagger$$

[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:
```

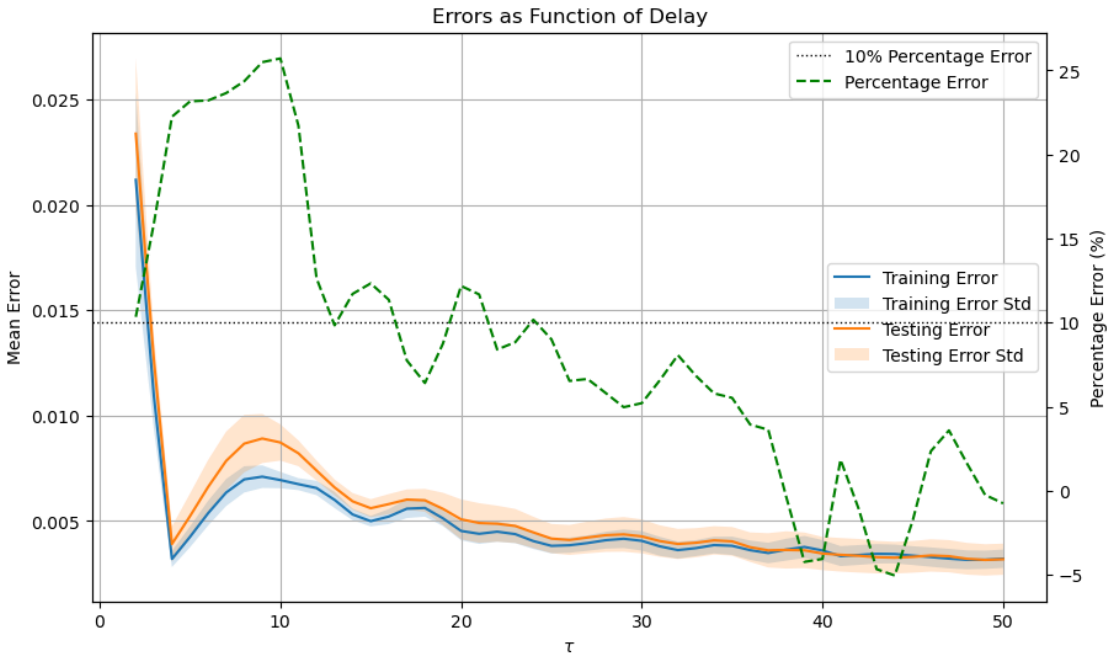
```
11.71%
Delay: 15, Mean Train Error: 0.0050, Mean Test Error: 0.0056, Percentage Error:
12.33%
Delay: 16, Mean Train Error: 0.0052, Mean Test Error: 0.0058, Percentage Error:
11.36%
Delay: 17, Mean Train Error: 0.0056, Mean Test Error: 0.0060, Percentage Error:
7.75%
Delay: 18, Mean Train Error: 0.0056, 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.

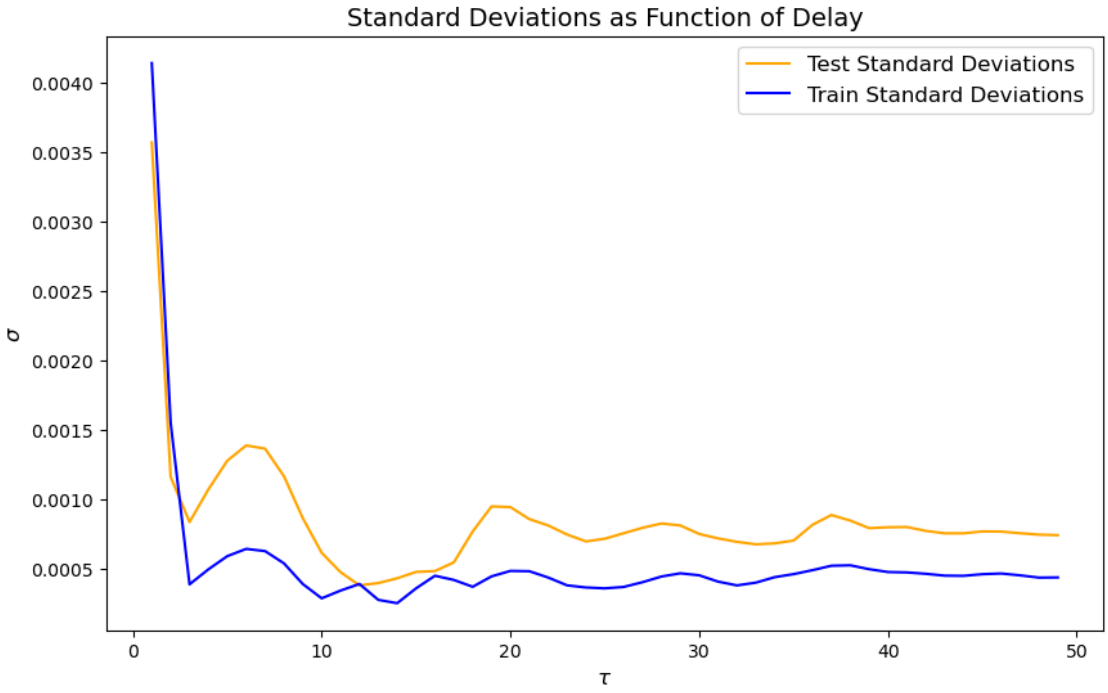
Large Delays ($\tau > 30$): Training and testing errors appear to stabilize at larger delays. This suggests the embedding has stabilised such that increasing the delay beyond a certain point adds diminishing returns in terms of accuracy, and the fluctuation of percentage errors into close negatives indicates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

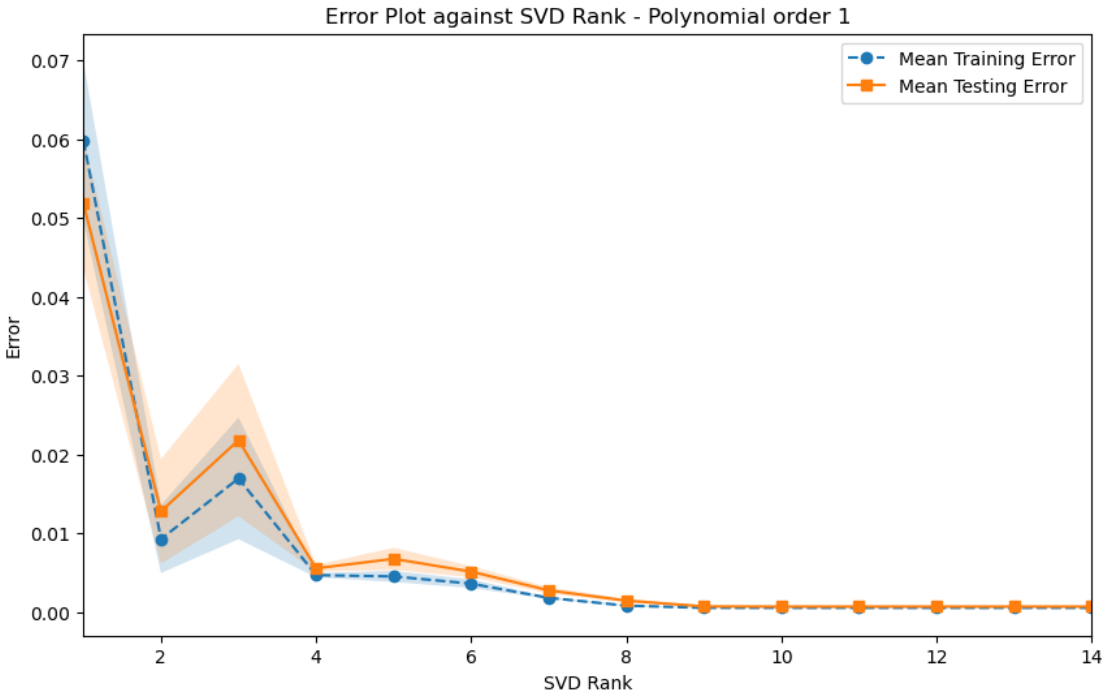
```
        r2_test = train_R2(data_test, delay, svd_rank, poly_order_svd)

        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))
```

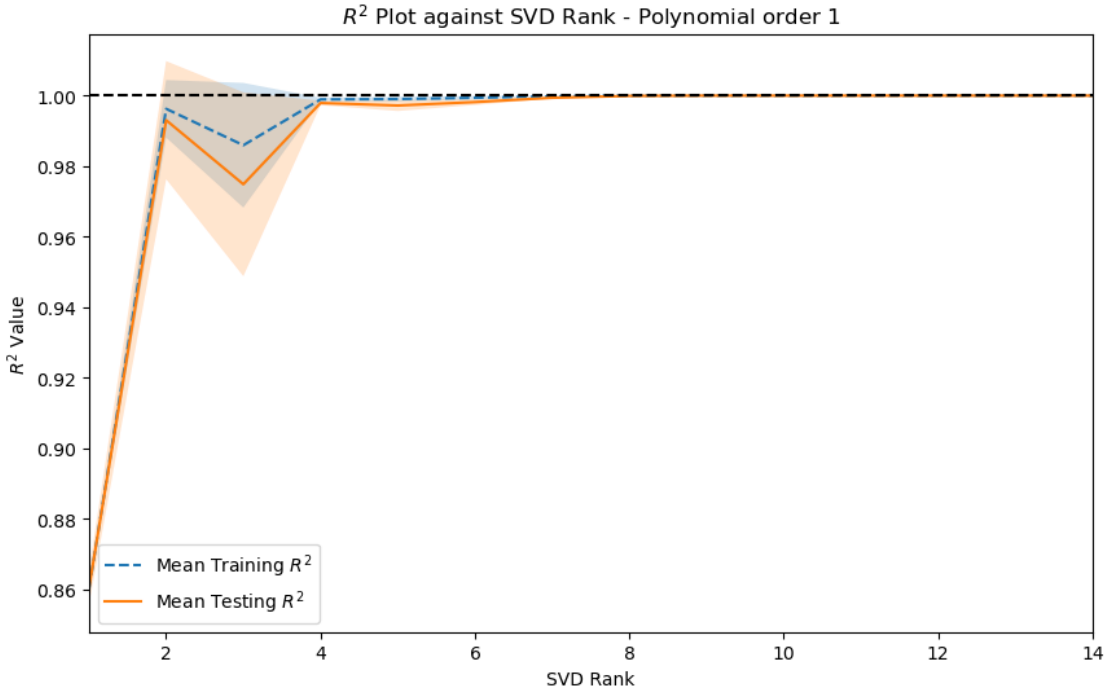
[544]:
```
# Visualisation of svd ranks and errors function
def plot_svd_error(svd_ranks, train_errors, test_errors, train_errors_std,␣
↪test_errors_std,order):
    plt.figure(figsize=(10, 6))

    # 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.

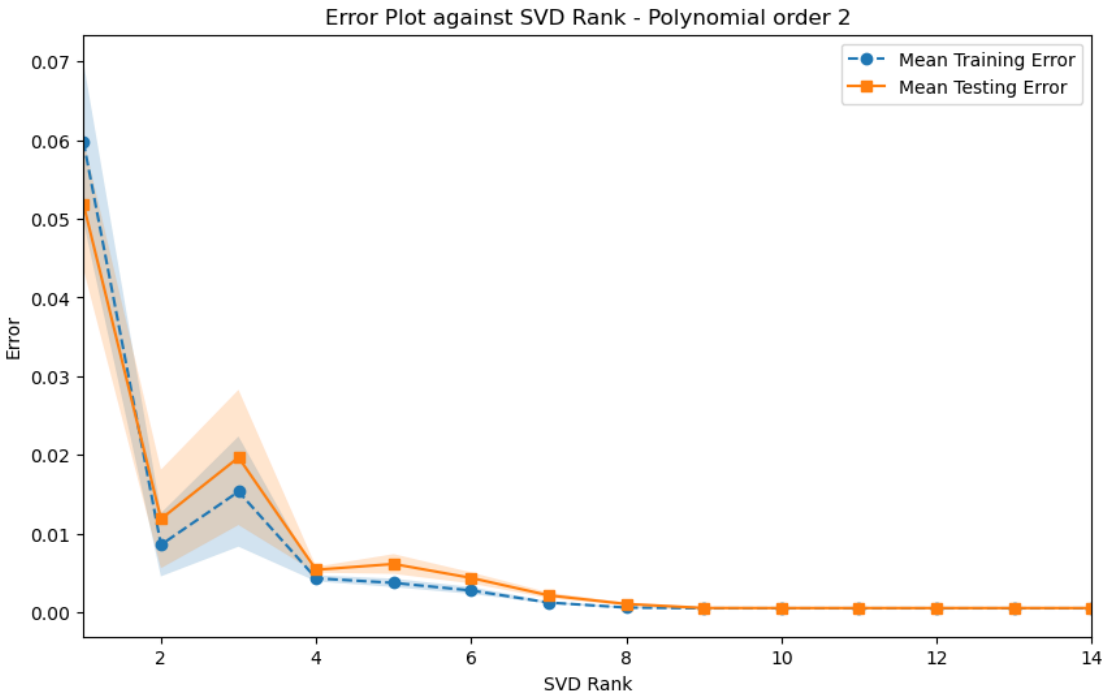
[546]:
```
#Linear
train_errors_svd_1, test_errors_svd_1, train_errors_std_svd_1,␣
 ↪test_errors_std_svd_1 = svd_rank_eval(
    data_train, data_test, poly_order_svd = 1
)
train_r2_svd_1, test_r2_svd_1, train_r2_std_svd_1, test_r2_std_svd_1 =␣
 ↪svd_rank_eval_r2(
    data_train, data_test, poly_order_svd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

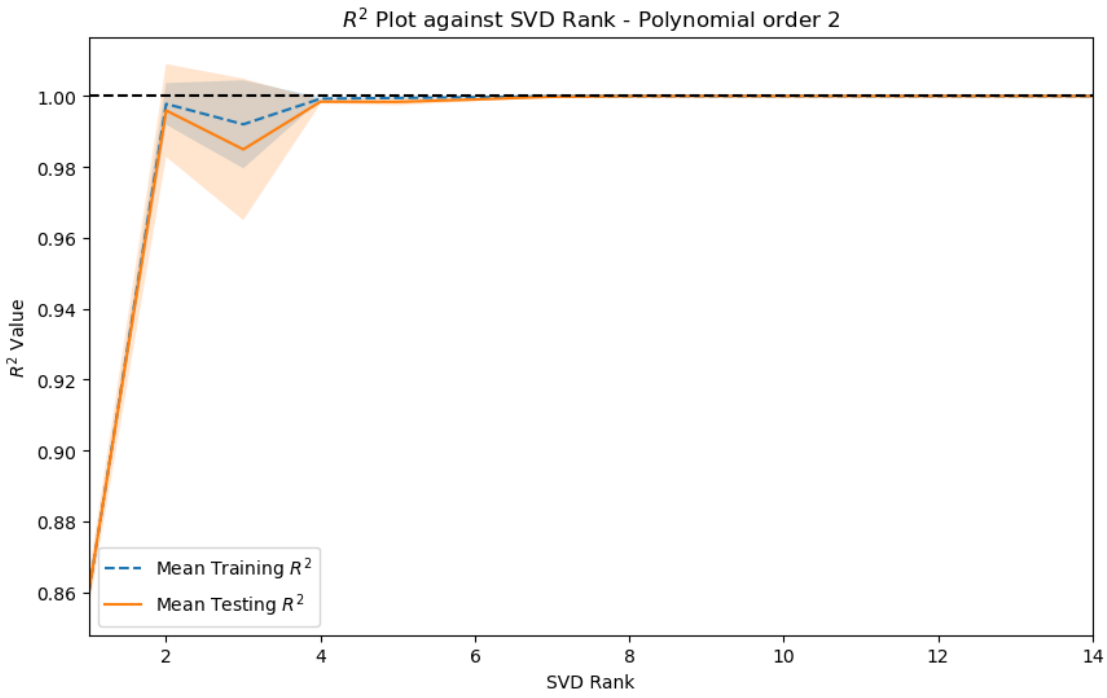




[547]:
```
#Quadratic
train_errors_svd_2, test_errors_svd_2, train_errors_std_svd_2,␣
 ↪test_errors_std_svd_2 = svd_rank_eval(
    data_train, data_test, poly_order_svd = 2
)
train_r2_svd_2, test_r2_svd_2, train_r2_std_svd_2, test_r2_std_svd_2 =␣
 ↪svd_rank_eval_r2(
    data_train, data_test, poly_order_svd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

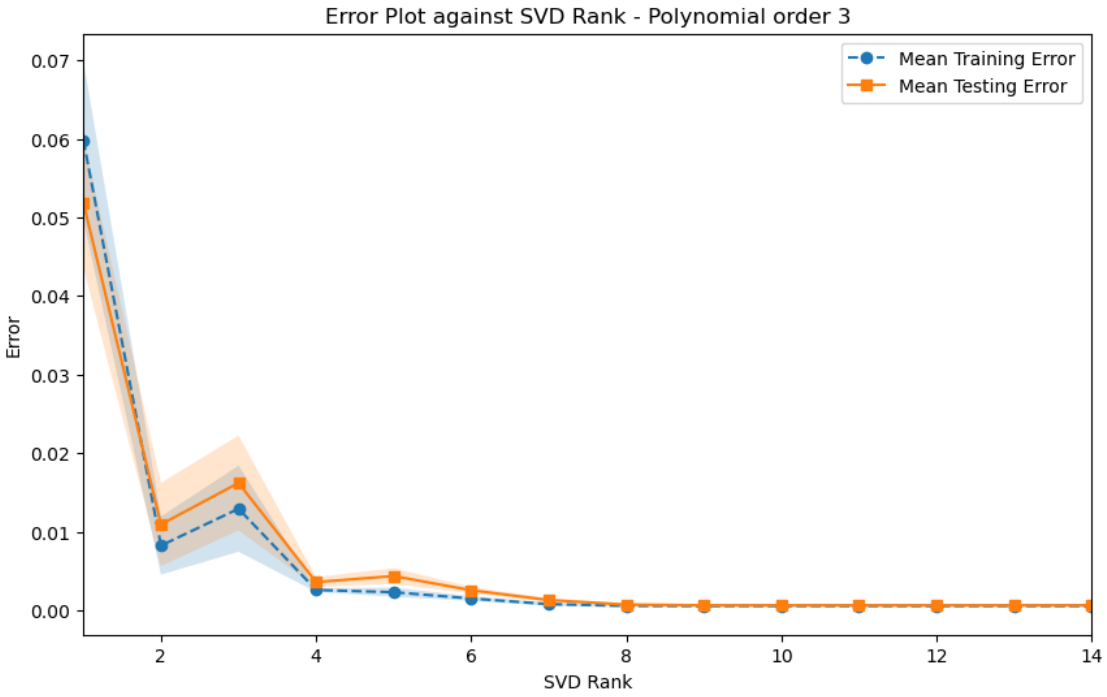
```
    train_r2 =train_r2_svd_2,
    test_r2 =test_r2_svd_2,
    train_r2_std=train_r2_std_svd_2,
    test_r2_std=test_r2_std_svd_2,
    order = 2
)
```

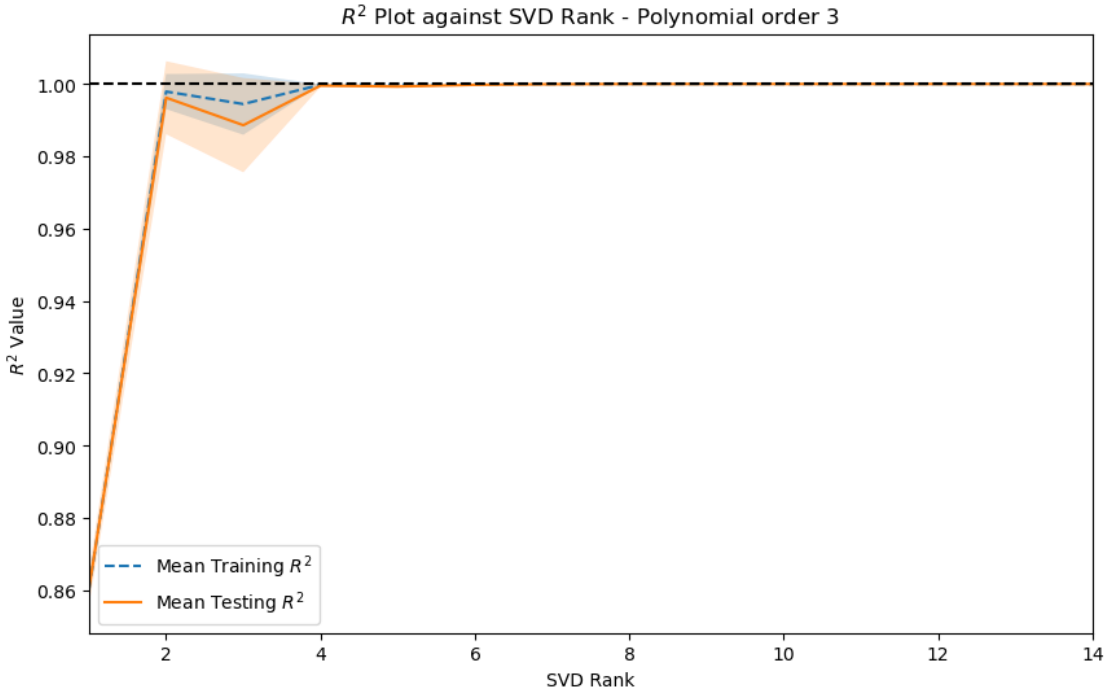




[550]:
```
#Cubic
train_errors_svd_3, test_errors_svd_3, train_errors_std_svd_3,␣
 ↪test_errors_std_svd_3 = svd_rank_eval(
    data_train, data_test,poly_order_svd = 3
)
train_r2_svd_3, test_r2_svd_3, train_r2_std_svd_3, test_r2_std_svd_3 =␣
 ↪svd_rank_eval_r2(
    data_train, data_test, poly_order_svd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

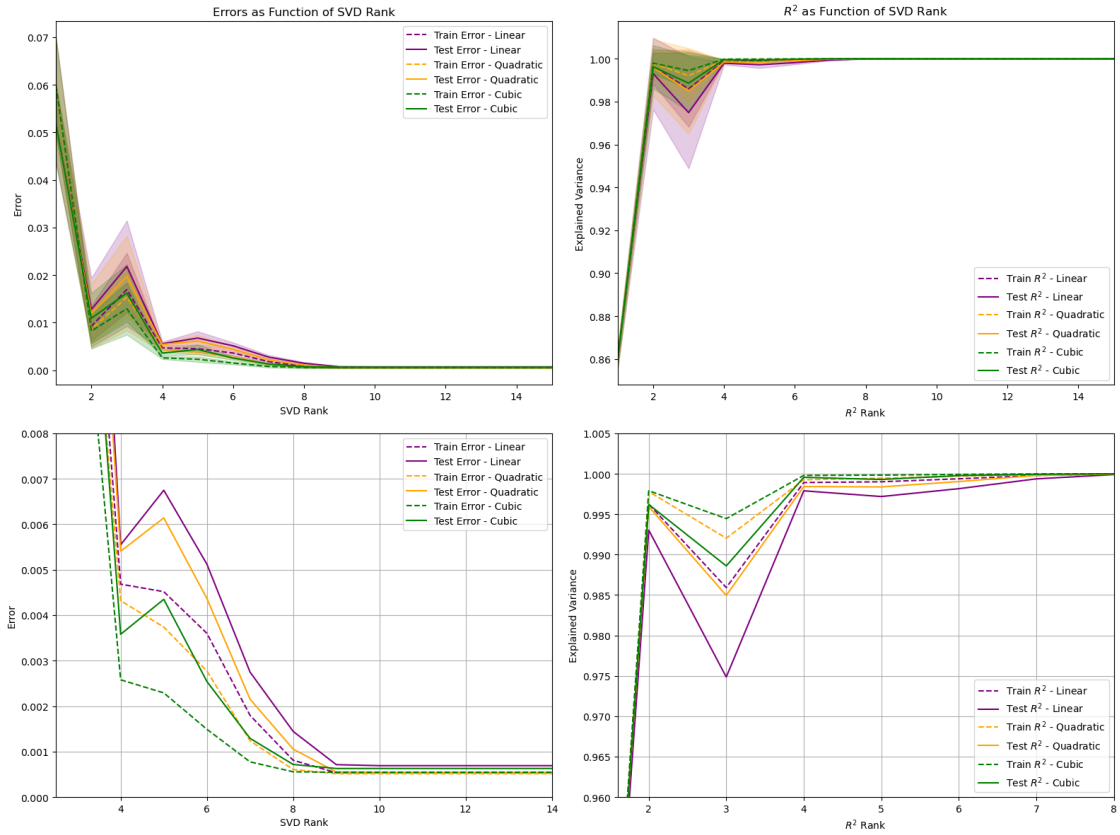
[1078]:
```
fig, axes = plt.subplots(2, 2, figsize=(16, 12))  # 2x2 layout

# 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

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

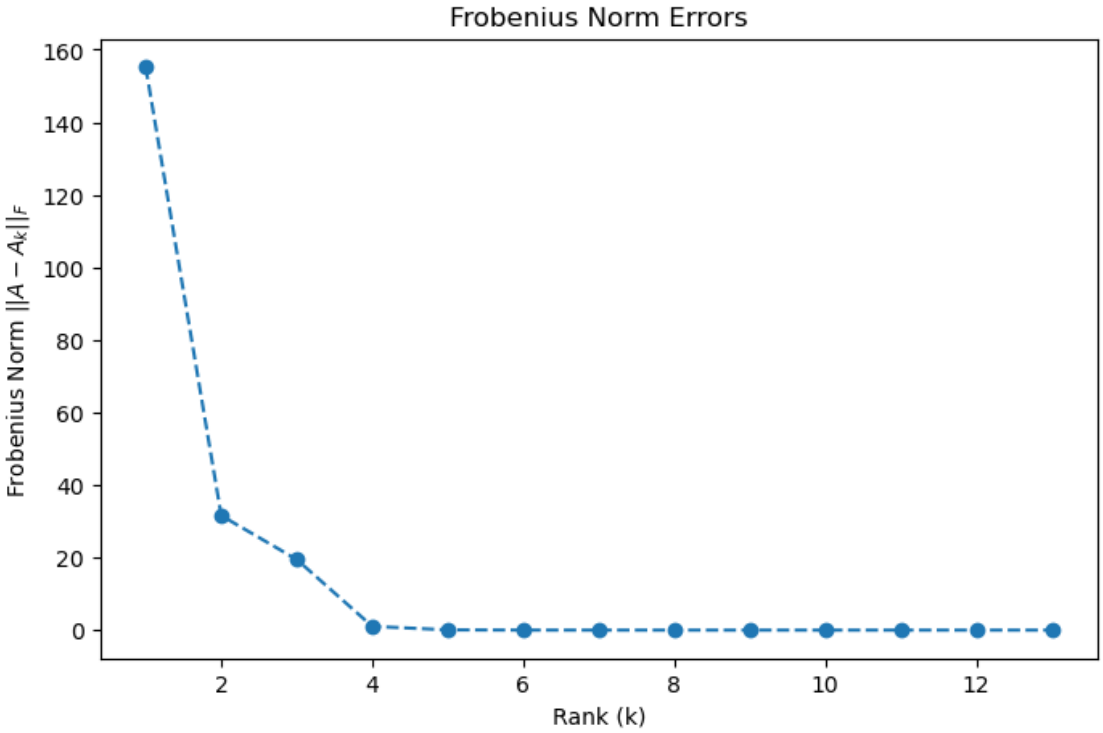
where $||\cdot||_F$ represents the Frobenius norm `frobenius_error()` and A_k is the reconstruction of A from the truncated ranks of its singular matrices. This is the physical application of the Eckhart-Young Theorem[11].

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
```

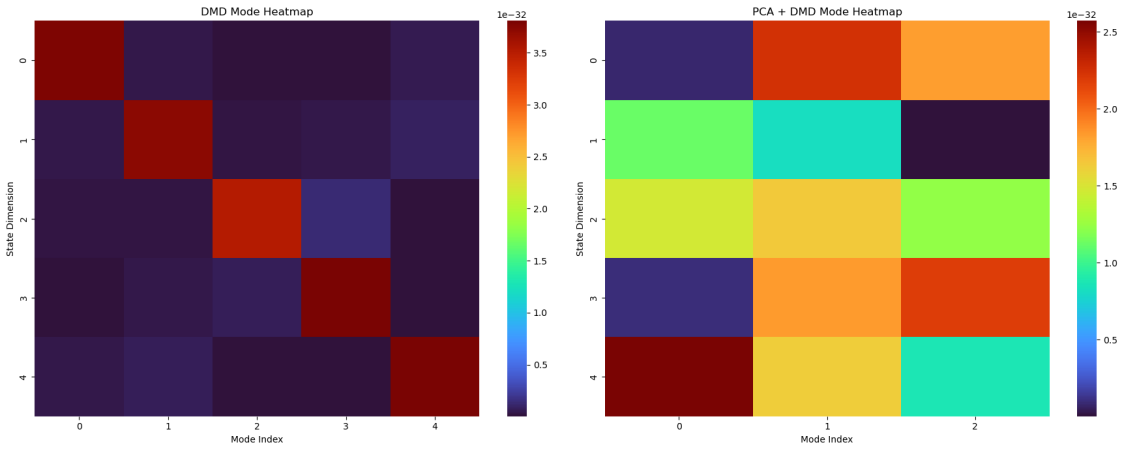
```
DMD_matrix = mode_aggregation(dmd_operators_train, dmd_modes_train,␣
 ↪eigenvalues_train, top_k,weights = weights(WW_train_norm))
DMD_matrix_reduced, components, explained_variance_ratio = pca_dmd(DMD_matrix,␣
 ↪pca_components)

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()
```



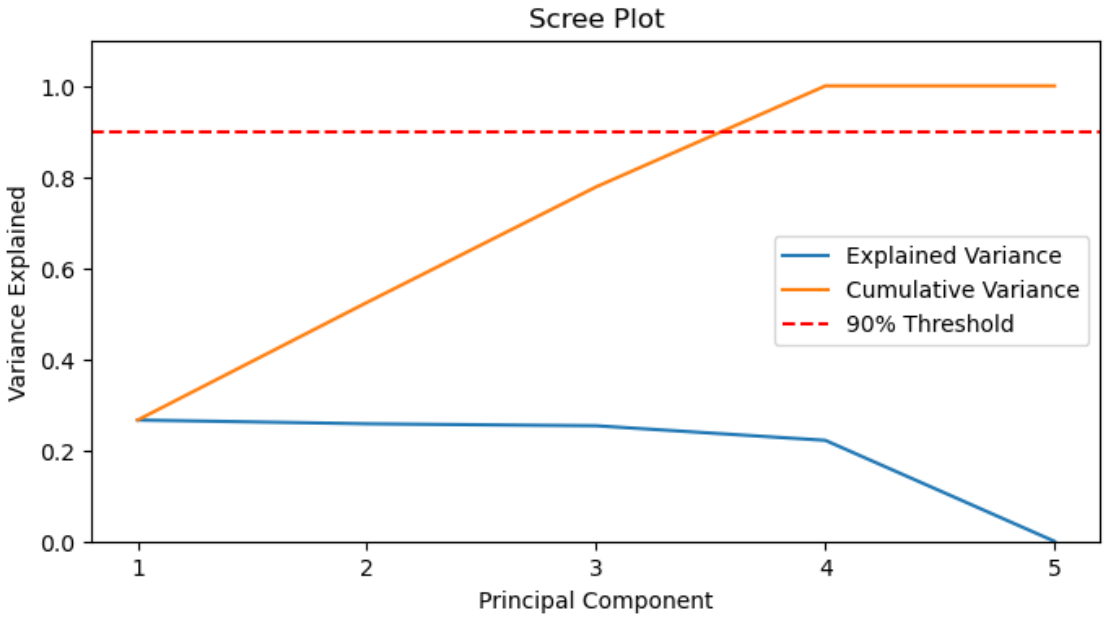
In the DMD heatmap, the most significant modes are likely to be those that show clear, consistent patterns across the state dimensions, however, no clear periodic patterns and sparsely populated dynamics are displayed with a lack of oscillatory components in the original DMD. After applying PCA, the 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.

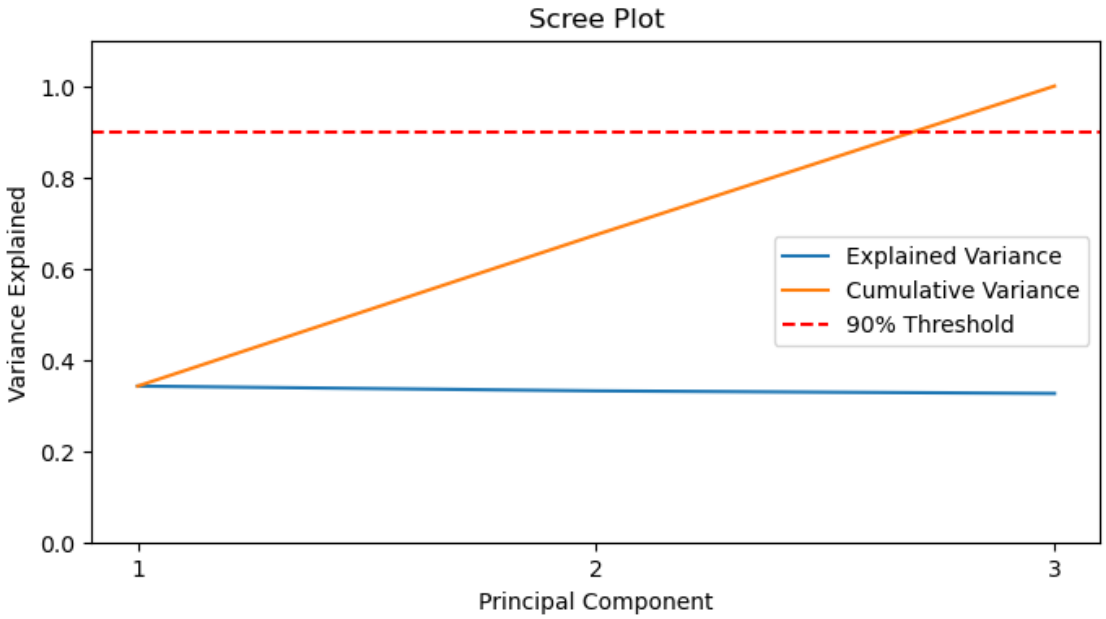
Scree Plot `compute_scree_plot()`

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.

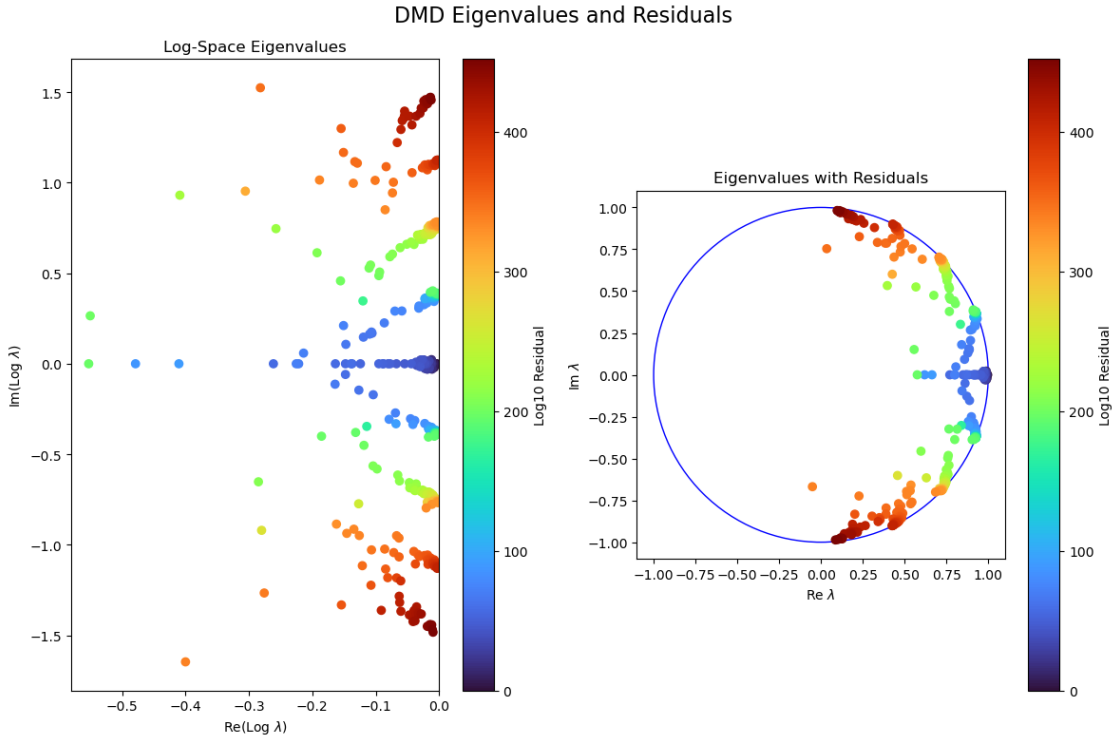
[1504]:
```
#Residuals
res_arr_train,evals_arr_train,evecs_arr_train = residual_dmd_full(data_train,␣
 ↪delay=optimum_delay, svd_rank = svd_rank_dmd, poly_order=1)

# 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;

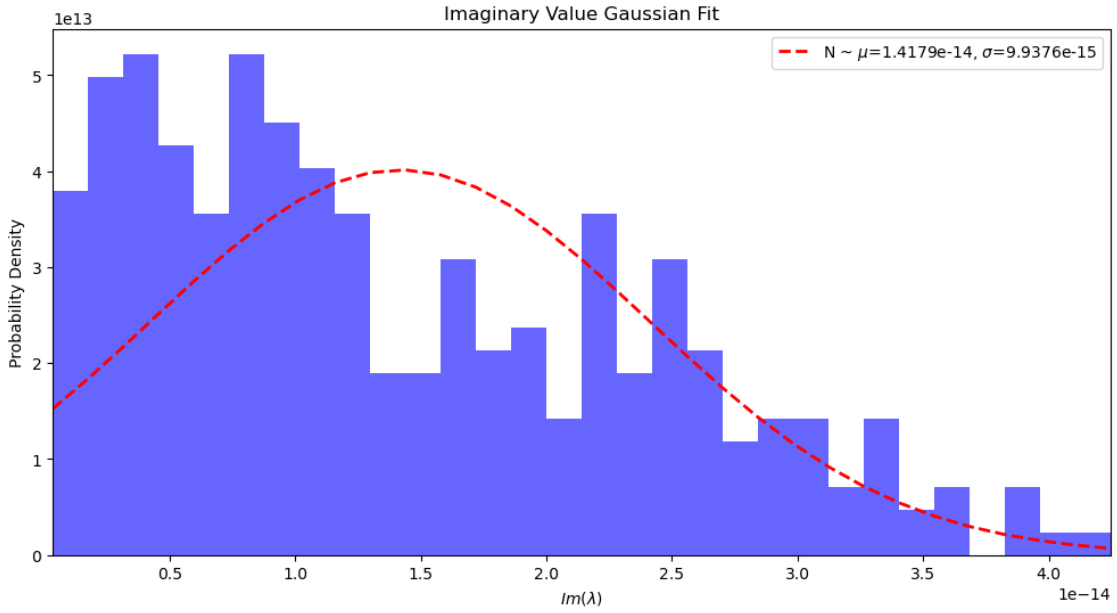
$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{(x-\mu)^2}{2\sigma^2}\right)$$

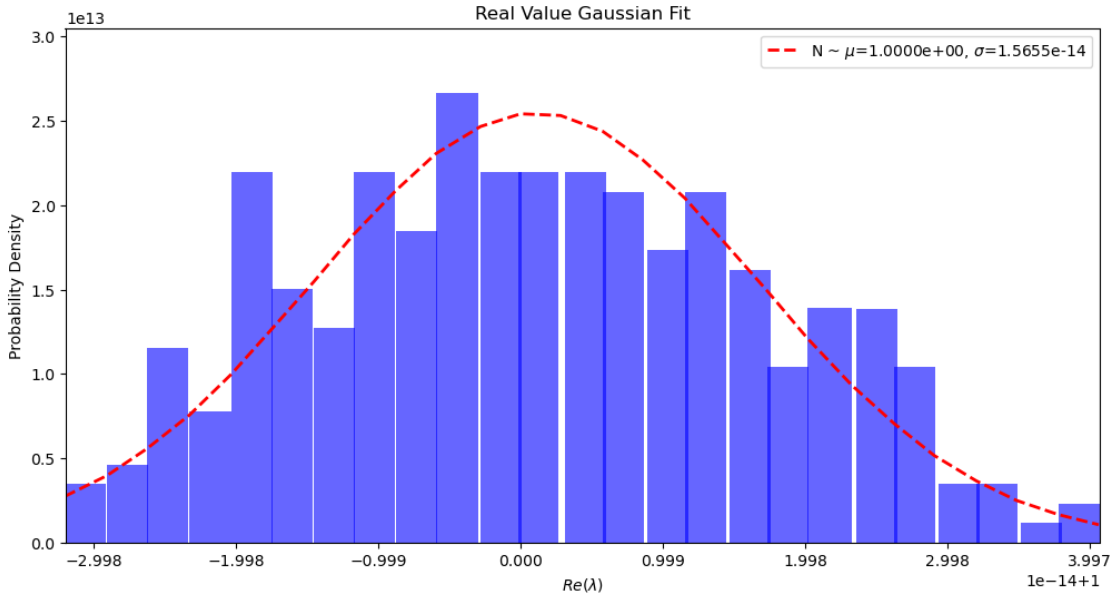
was fitted to the smallest residual eigenvalues for each bagging iteration. The plots are visualised as follows; for each complex basis and an overall contour plot. These visualisations were used to illustrate residual eigenvalue distributions across multiple bagging iterations, providing insight into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

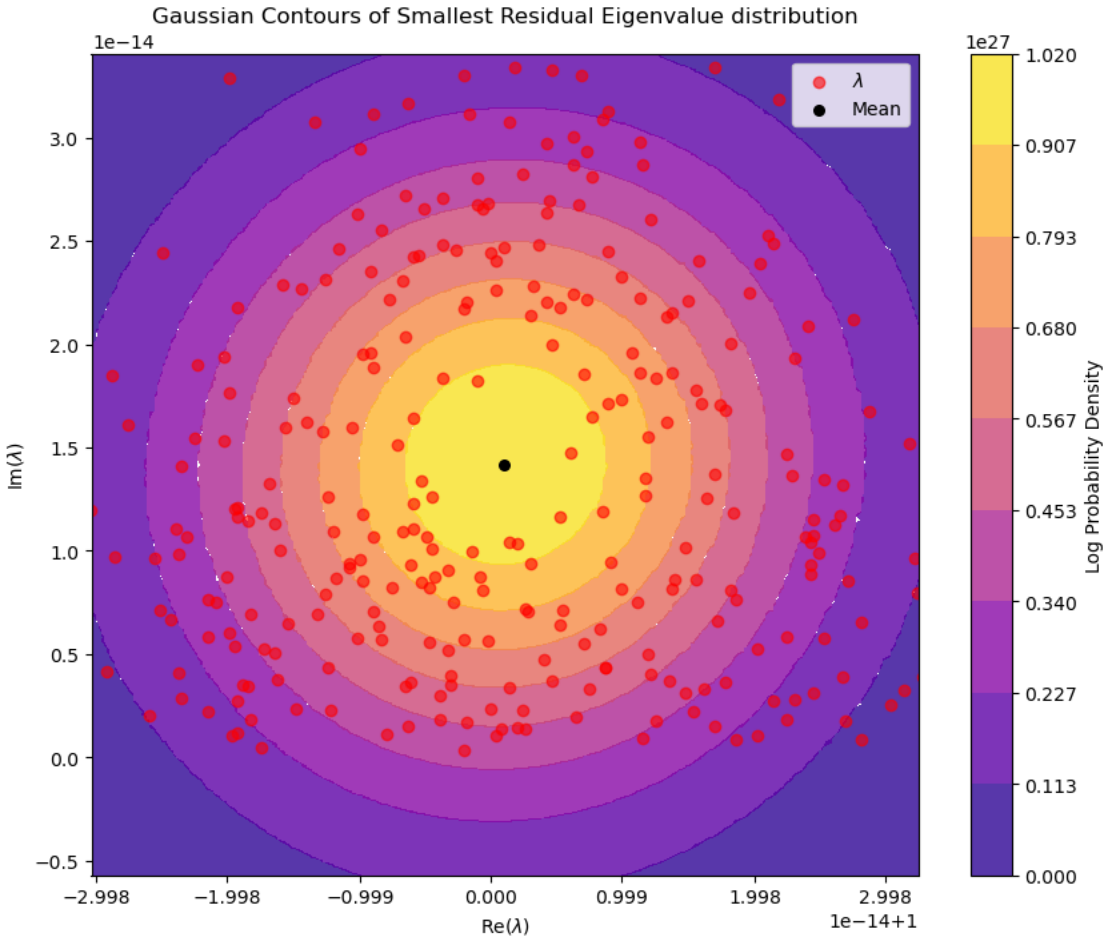
```
Mean Eigenvalue: (1.0000000000000013+1.41785701710148e-14j)
Mean Eigenvector: [1.00000000e+00 1.41785702e-14]
Covariance Matrix:
[[2.45888389e-28 5.00105787e-30]
 [5.00105787e-30 9.90866080e-29]]
```

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

2143062-DDPM-Part2

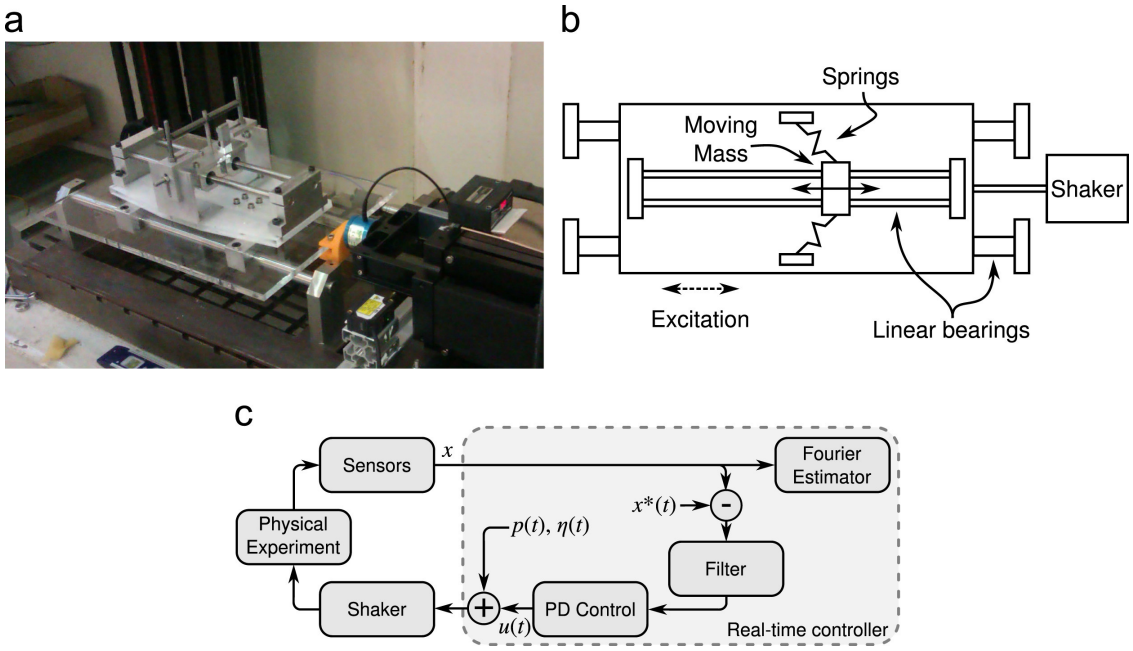
November 28, 2024

1 Data-Driven Physical Modelling (SEMTM0007) Coursework

1.0.1 Wishawin Lertnawapan 2143062

1.1 Table of Contents

1.2 0. Problem Description



> A non-linear mass-spring-damper system mounted on a shaking table. Displacements are measured using a laser displacement sensor and force is measure using a load cell.

System Identification:

The following report explores several artifical neural network (ANN) architectures in training a data-driven physical model. Data is derived from experimental oscillations of a mass-spring-damper

(NTMD) system [1], consisting of a mass supported by low-friction linear bearings, and notably two springs mounted perpendicular to the direction of motion. The mass itself is excited by a linear shaker, creating input excitations through a load cell, creating displacements to a linearly constrained mass.

The significance of the two springs is their introduction of nonlinearity, as opposed to linear mass-spring-damper systems where the spring restoring force aligns with the axis of applied motion. This behaviour is characterised to be geometric, resulting in hardening spring behaviour[2].

This nonlinear stiffness characteristic is to the justified use of ANNs. Due to both intrinsic material hardening and geometric complexities within the restoring force, parameters become nonlinear and thus cannot be represented analytically through conventional regression techniques or classical formulations. To illustrate, the restoring force of a conventional spring in a linear arrangement can be modelled using Hooke's Law [3]

$$F(x) = kx$$

where k is typically a proportionality constant, relating the restoring force F and the spring displacement x. In a perpendicular arrangement, the formulation extends to

$$F(x) = \hat{k}x\left(\frac{x}{s}\hat{i} + \frac{d}{s}\hat{j}\right)$$

where $\hat{k}$ is the nonlinear spring constant, and the terms within the bracket represent directional cosines of the spring force along the $\hat{i}$ and $\hat{j}$ bases respectively. This complexity limits techniques such as delay embedding as these rely on fundamentally linear fitting principles.

Measurement data is noted to be downsampled to 100 Hz, leading to measurement data being a discrete-time representation of a continuous dynamical system. The number of time series recorded is $n = 108$, each with a measurement size of $k = 2000$ sample data points which capture the dynamics. These measurement states can be complied as two matrices, $U(k)$, and $X(k)$ corresponding to the exication inputs and corresponding relative displacement of the mass respectively, for a given time step k.

Output Displacements (Laser Displacement Sensors):

$$X = \begin{bmatrix} x(0)_1 & x(\Delta t)_1 & x(2\Delta t)_1 & \dots & x(k\Delta t)_1 \\ x(0)_2 & x(\Delta t)_2 & x(2\Delta t)_2 & \dots & x(k\Delta t)_2 \\ & & \vdots & & \\ x(0)_n & x(\Delta t)_n & x(2\Delta t)_n & \dots & x(k\Delta t)_n \end{bmatrix}$$

Input Force (Load Cell)

$$U = \begin{bmatrix} u(0)_1 & u(\Delta t)_1 & u(2\Delta t)_1 & \dots & u(k\Delta t)_1 \\ u(0)_2 & u(\Delta t)_2 & u(2\Delta t)_2 & \dots & u(k\Delta t)_2 \\ & & \vdots & & \\ u(0)_n & u(\Delta t)_n & u(2\Delta t)_n & \dots & u(k\Delta t)_n \end{bmatrix}$$

The following ANN architectures were selected, as these architectures are particularly well suited for capturing non-autonomous temporal behavior. ESNs and RNNs retain memory of previous states with feedback mechanisms which allow periodic motion information to be captured. The major

limitation of applying ANNs in the context of the following problem is the relatively low quantity of independent time series, especially as this number decreases with splitting procedures. Assuming a split of 0.7, only 76 time series are available for training. Insufficient data can lead to poor model generalisation [4] and inability to capture nonlinear dynamics which may exhibit large variance from chaotic behaviour. On the testing set, this results in a small number of surrogates available for confident validation.

With this consideration, hyperparameters within the architectures are chosen to mitigate the risk of over-parameterisation and overfitting. Regularisation techniques and several cross-validation techniques have been implemented in the following sections to address this concern.

[1358]:
```
#Importing packages and libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

import scipy.integrate as spi
from scipy.integrate import solve_ivp
from scipy.stats import spearmanr

from sklearn.metrics import mean_squared_error
from sklearn.model_selection import KFold
from sklearn.model_selection import train_test_split

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.utils.data as torchdata

from itertools import product

import optuna
from optuna.visualization import plot_param_importances, plot_slice, plot_contour
optuna.logging.set_verbosity(optuna.logging.ERROR)
```

1.3 1. Exploratory Data Analysis

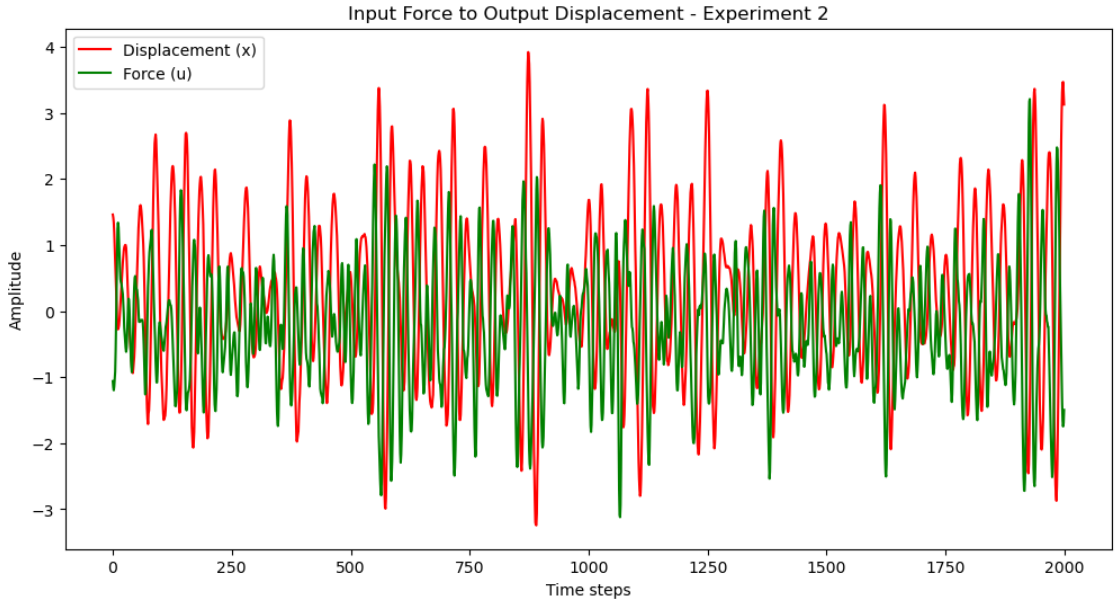
1.3.1 1.0 Time Series Plots

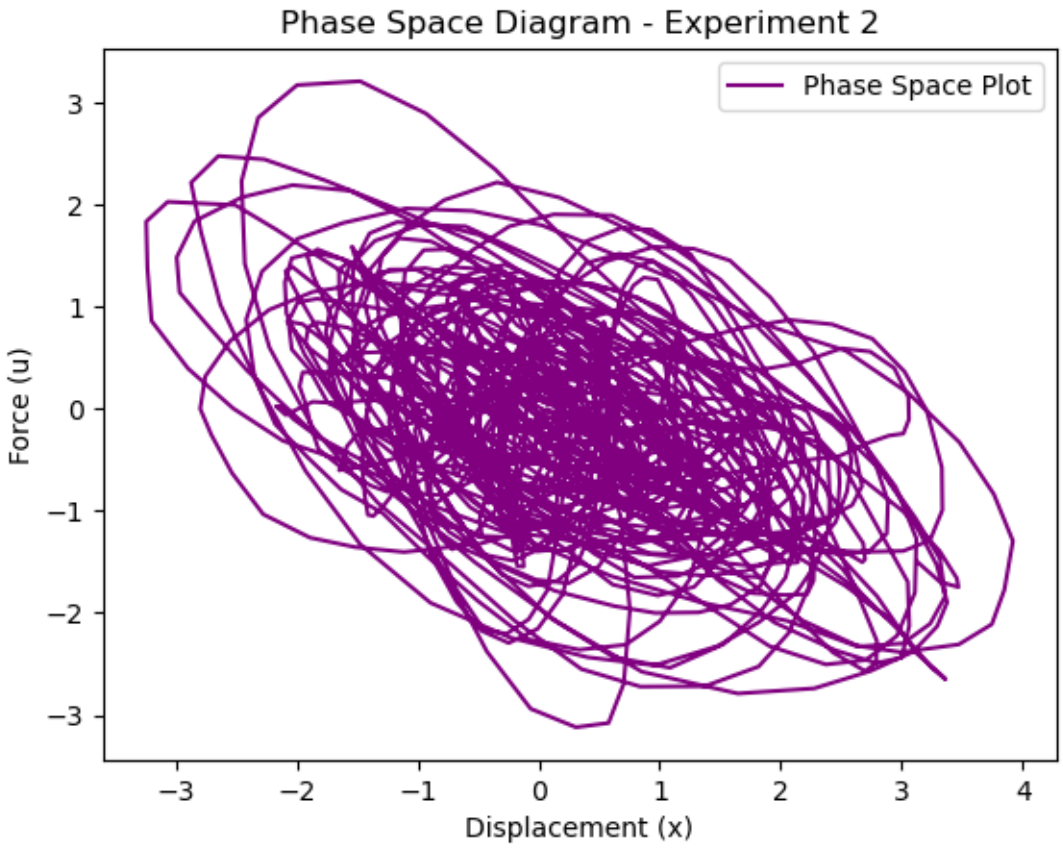
Importing the data for visualisation

[991]:
```
#Loading aata
data = np.load('coursework-2024+25-part2.npz')
x = data['x']
u = data['u']
```

[993]:
```
#-------------------------#
exp = 1 #Experiment
```

```
#--------------------------#

# Visualise data as time series plot
plt.figure(figsize=(12, 6))
plt.plot(np.arange(x.shape[1]), x[exp], label="Displacement (x)",color = "red")
plt.plot(np.arange(u.shape[1]), u[exp], label="Force (u)",color = "green")
plt.xlabel("Time steps")
plt.ylabel("Amplitude")
plt.title(f"Input Force to Output Displacement - Experiment {exp+1}")
plt.legend()
plt.show()

# Visualise data as phase space plot
plt.plot(x[exp], u[exp], label="Phase Space Plot", color='purple')
plt.xlabel("Displacement (x)")
plt.ylabel("Force (u)")
plt.title(f"Phase Space Diagram - Experiment {exp+1}")
plt.legend()
plt.show()
```





Time Series Graph:
The time-series graph exhibits oscillatory behaviour, with a clear correlation between inputs u and outputs x, and a noticeable lag between the input and output due to delayed energy dissipation. These are expected characteristics of a mass-spring-damper system. However, the system's nonlinearities are visible and manifest in the form of amplitude modulation variations.

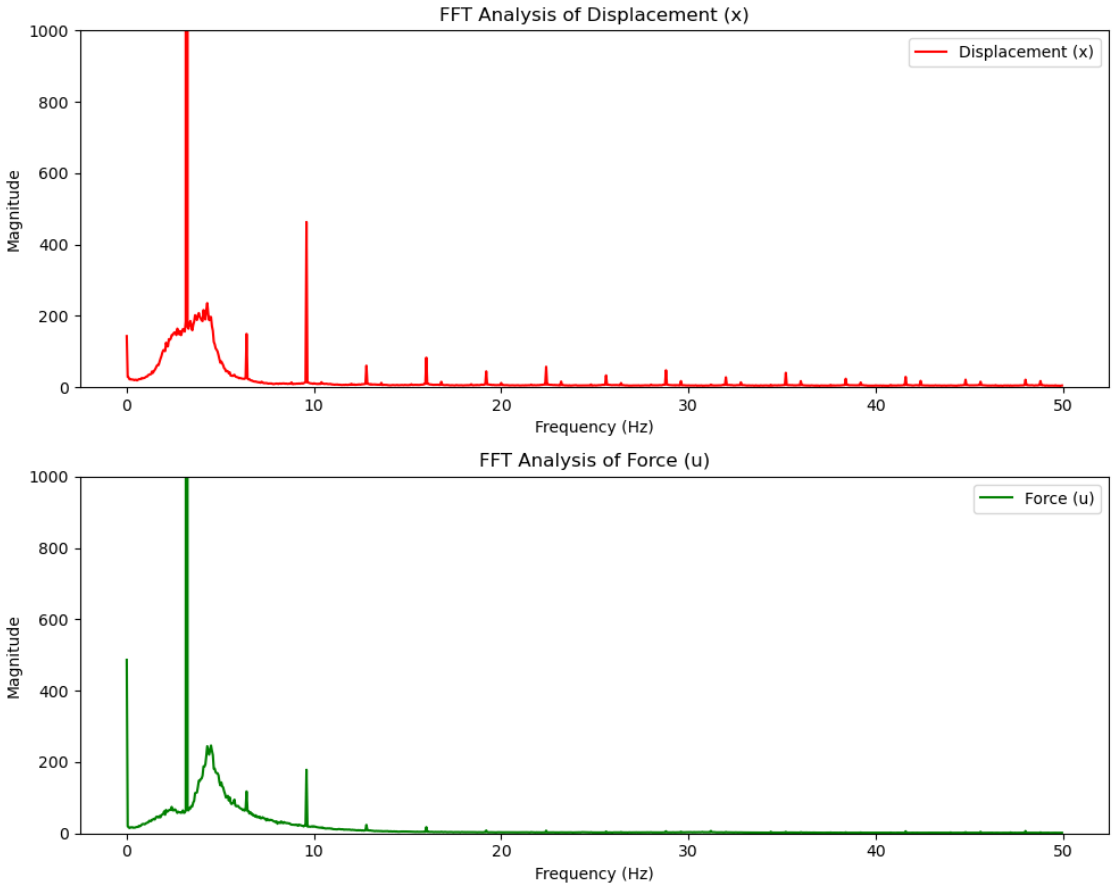
Phase Space Graph:
Phase space highlights the nonlinear and chaotic characteristics of the system, as the elliptic shapes are not fixed, and do not vary in a predictable pattern. The orientation of the semi-major axis further illustrates the phase difference between the two time series. Regions of clustering may indicate the recurrent underlying dynamics, while large variations may be due to the geometric nonlinear behaviour influenced by the spring.

Oscillations also appear affected by the presence of experimental noise, with irregularities from multi-frequency dynamics such as nonlinear energy transfer between modes through coupling. A spectral analysis was performed across all 108 tests to validate and characterise the frequencies.

[996]:
```
#Frequency Analysis function
def fft_analysis(series):
    n = len(series)
    freq = np.fft.fftfreq(n, d=1/100)
```

```
    fft_values = np.fft.fft(series)
    magnitude = np.abs(fft_values)
    return freq[:n//2], magnitude[:n//2]

x_magnitudes = []
u_magnitudes = []
# Average FFT across all time series
for series in x:
    _, magnitude = fft_analysis(series)
    x_magnitudes.append(magnitude)

for series in u:
    _, magnitude = fft_analysis(series)
    u_magnitudes.append(magnitude)

avg_x_magnitude = np.mean(x_magnitudes, axis=0)
avg_u_magnitude = np.mean(u_magnitudes, axis=0)

freq = fft_analysis(x[0])[0]
```

[998]:
```
# Visualising FFT analysis
plt.figure(figsize=(10, 8))

# Subplot x FFT
plt.subplot(2, 1, 1)
plt.plot(freq, avg_x_magnitude, label="Displacement (x)", color="red")
plt.title("FFT Analysis of Displacement (x)")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Magnitude")
plt.ylim(0,1000)
plt.legend()

# Subplot x FFT
plt.subplot(2, 1, 2)
plt.plot(freq, avg_u_magnitude, label="Force (u)", color="green")
plt.title("FFT Analysis of Force (u)")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Magnitude")
plt.ylim(0,1000)
plt.legend()

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



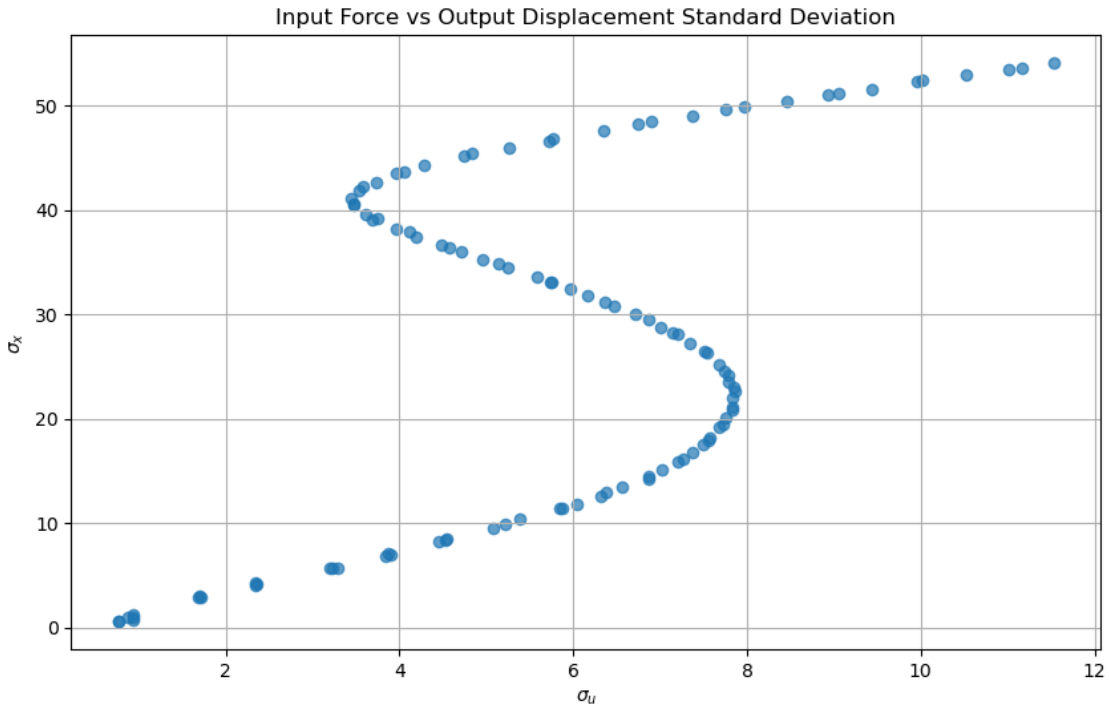
Frequency Analysis `fft_analysis()`

An FFT plot was performed for both input and displacement across all time series before averaging. The key observation is the large peak exhibited in both graphs at 4 Hz which likely corresponds to the dominant natural frequency of oscillation. Interestingly, both graphs show smaller pronounced peaks and an elevated low-frequency response band. The prior phenomena are known as harmonic peaks [5] occuring at multiples of the fundamental frequency and possibly are a consequence of nonlinearities within the system. The latter indicates energy is distributed over a range near the dominant frequency, possibly caused by damping effects within the NTMD. The presence of noise is largely negligible, which justify the lack of filtering preprocessing procedures.

1.3.2 1.1 Standard Deviation

The following step calculates the standard deviation of amplitudes for both input and output matrices. In other terms, for each experiment time series, a measure of its variation across its signal length is computed. This was then plotted to identify the relationship of each signal's amplitude, more specifically the correlation between the system's inputs σ_u, and its direct effect on the response σ_x.

[1003]:
```
x_original = x
u_original = u

# Standard deviation
std_x = np.std(x, axis=1)
std_u = np.std(u, axis=1)

# Visualise standard deviations of inputs vs outputs
plt.figure(figsize=(10, 6))
plt.scatter(std_u, std_x, alpha=0.7)
plt.xlabel("$\\sigma_u$")
plt.ylabel("$\\sigma_x$")
plt.title("Input Force vs Output Displacement Standard Deviation")
plt.grid(True)
plt.show()
```



Standard Deviation Analysis

From the visualisation, a positive correlation is shown, which is expected for coupled dynamics for which higher variability in the input force correlates to outputs with higher variablility. However, the data points do not all follow a strict line, rather an S-shaped curve indicating some level of variation in how the system responds to similar force amplitudes. This could be due to nonlinear effects in the mass-spring-damper system, where other factors influence displacement amplitude.

In context of the physical system, there can be several reasons for this shape. For small values of

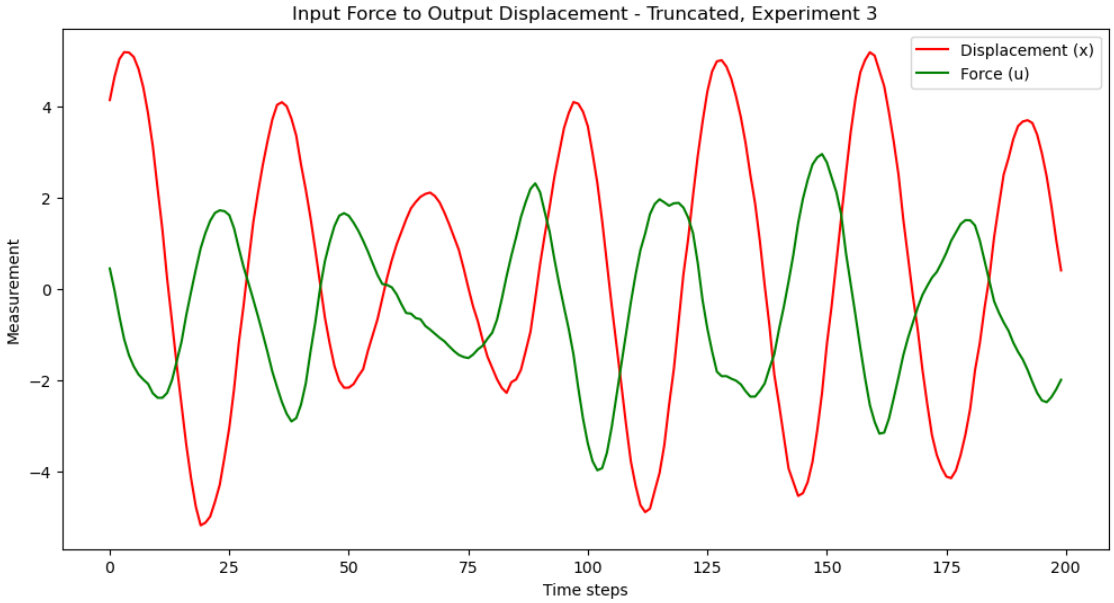
$\sigma_u < 20$, σ_x is less sensitive, which suggest nonlinear damping effects limiting variability. The same effect is also experienced at higher values where $\sigma_u > 40$, showing that the systems response is likely constrained, or limited by the stiffness of the spring being saturated at higher forces. Overall, the plot communicates clear justification for using higher complexity models in subsequent sections.

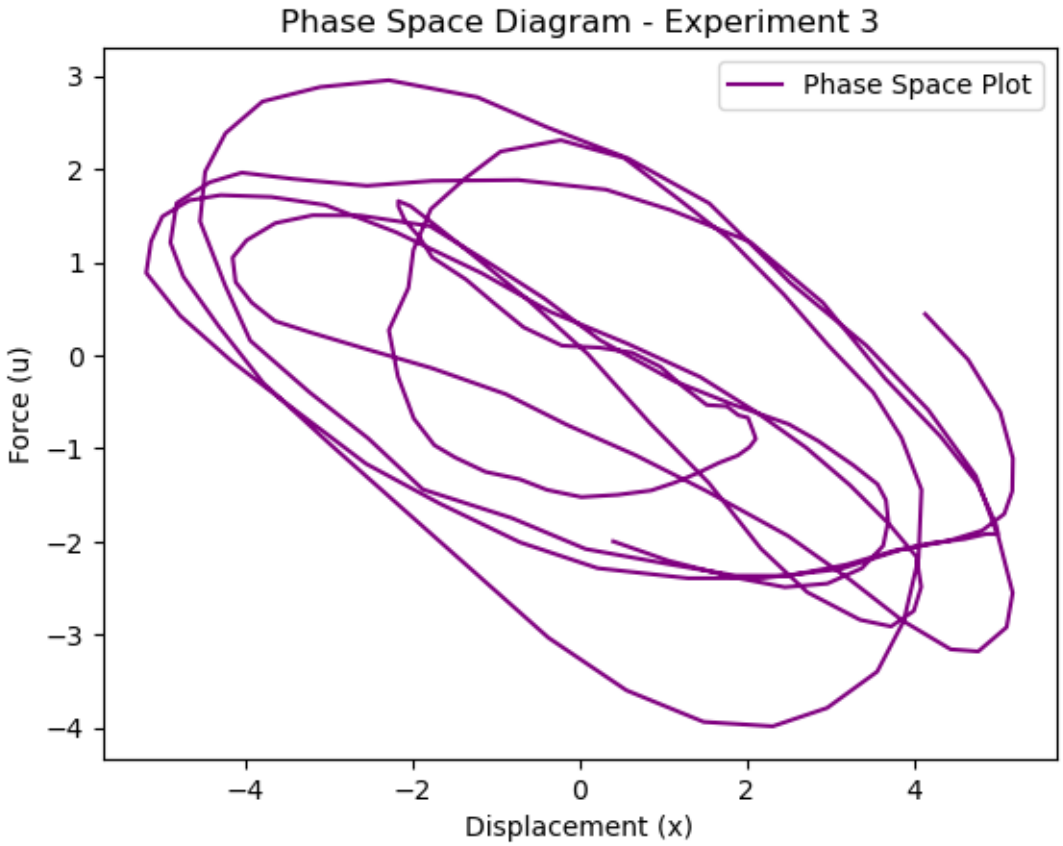
1.3.3 1.2 Truncation

Each time series was truncated to the first 200 index, purely as a means to improve computational speed. However, the report acknowledges the downstream effects of lowering complexity, especially in context of nonlinear models. By truncating to effectively a 10th of the series' original length, truncation has potential to remove major temporal dependencies which formulate the underlying dynamics the model is being trained upon.

For ESNs, RNNs and their advanced architectures, removing data will likely negatively impact learning of long-term dependencies and retention of important temporal patterns. This can lead to risk of overfitting to shorter patterns but was deemed necessary to allow efficient hyperparameter exploration.

[1368]:
```
# Truncating to 200
x = x[:, :200]
u = u[:, :200]

# Visualise truncated time series
plt.figure(figsize=(12, 6))
plt.plot(np.arange(x.shape[1]), x[exp], label="Displacement (x)",color = "red")
plt.plot(np.arange(u.shape[1]), u[exp], label="Force (u)",color = "green")
plt.xlabel("Time steps")
plt.ylabel("Measurement")
plt.title(f"Input Force to Output Displacement - Truncated, Experiment {exp+1}")
plt.legend()
plt.show()

# Visualise truncated data as phase space plot
plt.plot(x[exp], u[exp], label="Phase Space Plot", color='purple')
plt.xlabel("Displacement (x)")
plt.ylabel("Force (u)")
plt.title(f"Phase Space Diagram - Experiment {exp+1}")
plt.legend()
plt.show()
```


Input Force to Output Displacement - Truncated, Experiment 3
Displacement (x)
Force (u)
Measurement
Time steps
4
2
0
−2
−4
0
25
50
75
100
125
150
175
200


Phase Space Diagram - Experiment 3
Phase Space Plot
Force (u)
Displacement (x)
3
2
1
0
−1
−2
−3
−4
−4
−2
0
2
4

1.3.4 1.3 Normalisation

Min-Max Normalisation: `min_max_norm()`
Most ANN architecture operate effectively on appropriately scaled and normalised data. The importance of the following steps ensures experimental data is effectively preprocessed and the scale of data is consistent.

The method of normalisation applied is min-max normalisation using the function below. The function scales the data to a fixed range of (-1,1) by applying the formulation

$$x_n = 2\frac{x - x_{\min}}{x_{\max} - x_{\min}} - 1$$

As with other forms of machine learning, normalisation is an essential preprocessing step especially in the context of ANNs as these models rely on gradient-based optimisation such as ADAM. Large variations in the features can lead to numerical instability in phenomena such as exploding or vanishing gradients. Additionally, the step ensures features remain comparable regardless of scale.

The choice of min-max scaling technique over other forms of normalisation for ANNs ensures preservation of specific properties [6]. Though not universally better than other methods, it is fairly simple to implement and performs the necessary adjustments to the data range. By adhering to this range, certain properties can be satisfied; including ensuring data remains within the effective range of nonlinear activation functions, and the echo state property of ESNs.

[1370]:
```
# Min-Max Normalisation function
def min_max_norm(x):
    min_x = np.min(x)
    max_x = np.max(x)
    x_norm = 2 * (x - min_x) / (max_x - min_x) - 1
    return x_norm

x_norm = min_max_norm(x)
u_norm = min_max_norm(u)

#Print range
print(f"x Max-Min values: {np.max(x_norm)}, {np.min(x_norm)}")
print(f"u Max-Min values: {np.max(u_norm)}, {np.min(u_norm)}")
```

```
x Max-Min values: 1.0, -1.0
u Max-Min values: 1.0, -1.0
```

1.4 2. Echo State Neural Networks

Echo State Networks (ESNs) are a simple neural network architecture designed to process sequential data. The underlying principle lies in the concept of a fixed, hidden connected reservoir layer of neurones, which transforms input sequences into a high-dimensional dynamic state space. They are a considered a simplified formulation of more typical Recurrent Neural Networks (RNNs), as the reservoir remains untrained, and only the output weights W_o are optimised - typically through linear optimisation techniques. ESNs are considered appropriate for capturing the dynamical system, which leverages identification of temporal dependencies and nonlinear dynamics. The advantage of

the reservoir's stable dynamics enable the network to capture complex patterns without the need for iterative backpropagation through time [7].

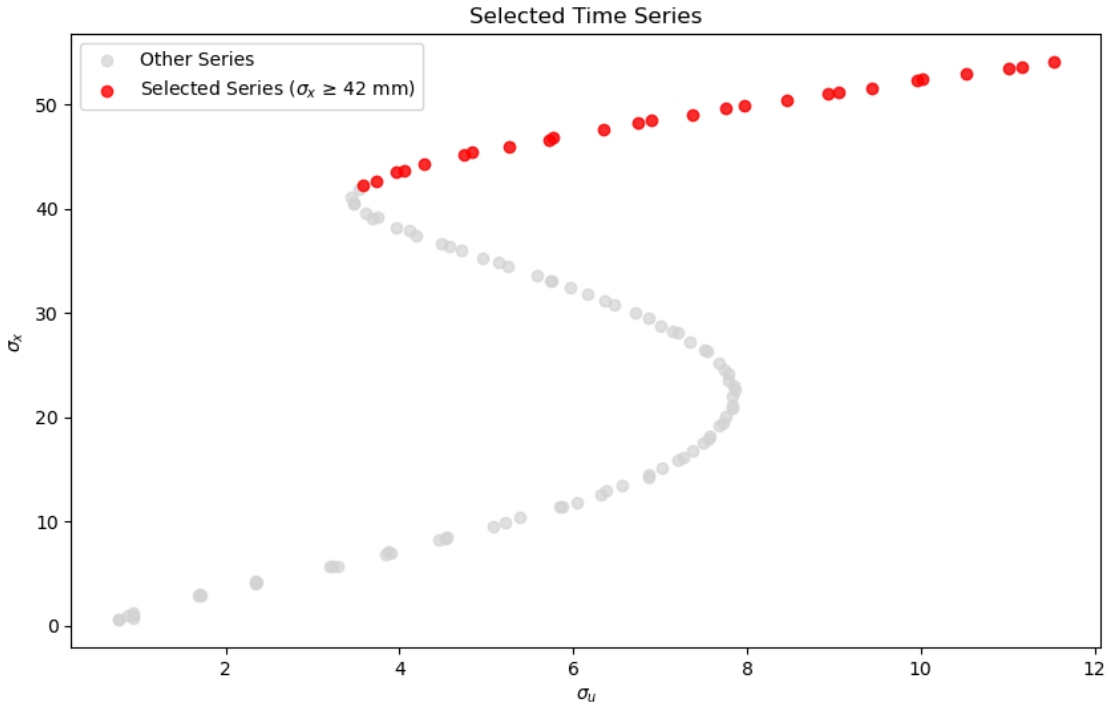
The use of ESN allows preliminary predictive modelling to be applied. Unlike standard RNNs, which require computationally expensive gradient-based training, and more hyperparameters to optimise, ESNs train only the output weights owing to their computational efficiency and less risk of vanishing/exploding gradient problems.

1.4.1 2.1 Data Separation

For exploratory processes, the ESN is applied for a select subset of time series where the standard deviation of the input signal σ_u is greater than a threshold value of 42mm.

[1450]:
```
# Criterion
selectidx = np.where(np.std(x_original, axis=1) >= 42)[0]

x_selected = x_norm[selectidx]
u_selected = u_norm[selectidx]

print("Selected series: ", len(selectidx))

# Visualise selected indices
plt.figure(figsize=(10, 6))
plt.scatter(std_u, std_x, color="lightgray", label="Other Series", alpha=0.7)
plt.scatter(std_u[selectidx], std_x[selectidx], color="red", label="Selected␣
↪Series ($\\sigma_x$  42 mm)", alpha=0.8)

plt.xlabel("$\\sigma_u$")
plt.ylabel("$\\sigma_x$")
plt.title("Selected Time Series")
plt.legend()
plt.show()
```

```
Selected series:  26
```



[1374]:
```
#Training and Testing Split
#--------------------------#
esn_split = 0.7
#--------------------------#

x_train, x_test, u_train, u_test = train_test_split(x_selected, u_selected,␣
 ↪test_size=1-esn_split)

print(f"x training data shape: {x_train.shape} ")
print(f"x testing data shape: {x_test.shape} ")
print(f"u training data shape: {u_train.shape} ")
print(f"u testing data shape: {u_test.shape} ")
```

```
x training data shape: (18, 200)
x testing data shape: (8, 200)
u training data shape: (18, 200)
u testing data shape: (8, 200)
```

1.4.2 2.2 ESN Formulation

Reservoir Creation `create_reservoir()`
Inputs: - `size`: The reservoir dimension N, which defines the number of neurones inside the hidden layer - `in_size`: Dimensionality of the input signal - `spectral radius`: The spectral radius of the reservoir weight matrix - `sparsity`: Proportion of zero elements in the reservoir

The ESN reservoir of a determined size $W \in \mathbb{R}^{N \times N}$ was initialised using this function. Within it, its weights can be randomly initialised with a uniform distribution between the ranges (-0.5, 0.5). An additional `sparsity` mask was applied to control the fraction of zero elements was an architectural choice as a form of regularisation, mimicking the sparsity-promoting action of conventional regularisation by limiting the number of possible connections. This deliberate design choice was made to prevent overfitting, especially as the data set is relatively small.

Additionally, the weights W are scaled to ensure its largest absolute eigenvalue $\lambda_{\max}(W)$ is mapped to the value of the defined spectral radius ρ to ensure stable dynamics [8] in the reservoir dynamics.

$$W \leftarrow W \cdot \frac{\rho}{\lambda_{\max}(W)}$$

For the Echo State Property to hold true, $\rho < 1$.

[1308]:
```
def create_reservoir(size, in_size=1, spectral_radius=0.9,sparsity = 0.2):
    # Reservoir masking
    W = np.random.rand(size, size) - 0.5
    mask = np.random.rand(size, size) < sparsity
    W *= mask

    # Scaling for spectral radius
    max_eig = np.max(np.abs(np.linalg.eigvals(W)))
    if max_eig > 0:
        W *= spectral_radius / max_eig

    # Step 4: Initialize the input weight matrix Win
    Win = np.random.rand(size, in_size) - 0.5

    return W, Win
```

State Evolution `run_reservoir()`
Inputs: - `W`: The reservoir weight matrix - `Win`: The input weight matrix - `u`: Input time series - `x`: Initial state vector of the reservoir

The state of the reservoir at time $t+1$ is evolved using this function, transforming the input sequence x to a high-dimensional state space x_t to capture temporal dependencies. Initially, this state is set to 0, but gradually modified as it is fed back into iterations and the model receives new information. Mathematically, evolution can be expressed as

$$x_{t+1} = f(W_{in}u_t + Wx_t)$$

where f is a nonlinear activation function mimicking the concept of neurones being in inactive or active states. Several logistic functions are explored in later sections. However, for the purposes of simplicity, and as the range of scaled inputs is (-1,1), the tanh function was used.

[1311]:
```
def run_reservoir(W, Win, u, x = None):
    if x is None:
        x = np.zeros(W.shape[0])

    all_x = np.zeros((u.shape[0], W.shape[0]))
```

```
    for i in range(u.shape[0]):
        x = np.tanh(np.dot(W, x) + np.dot(Win, u[i, :])) #nonlinear activation␣
↪function
        all_x[i, :] = x
    return all_x
```

Output Weight Training `train_reservoir()` - `all_x`: - `y`: Target output matrix - `ridge_alpha`:

Trains the output weights of the reservoir using linear regression methods, which aims to minimise the difference between the predicted outputs y and the updated reservoir outputs x_t

$$W_{out} = \text{argmin}_W ||y - W_x||_F^2$$

where $||\cdot||_F$ is the Frobenius norm. The additional ridge parameter α introduces a regularisation term, which is another measure to mitigate the risk of the ESN overfitting. This parameter modifies the optimisation problem into a ridge regression with closed form solution:

$$W_{out} = \text{argmin}_W ||y - W_x||_F^2 + \alpha||W||_F^2$$

using matrix notation, the code form is expressed as

$$W_{out} = (X^T X + \alpha I)^{-1} X^T y$$

[1314]:
```
def train_reservoir(all_x, y,ridge_alpha=1e-6):
    I = np.identity(all_x.shape[1])
    return np.linalg.lstsq(all_x.T @ all_x + ridge_alpha * I, all_x.T @ y,␣
↪rcond=None)[0]
```

Predicting Output `predict_reservoir()`
Inputs: - `Wout`: The output weight matrix. - `W` : The reservoir weight matrix. - `Win`:he input weight matrix. - `u`: The input time series

Used to predict the output signal y_t for the given input sequence u_t using the trained output weights W_{out} by linear combination

$$y_t = W_{out}^T x_t$$

where it is used to evaluate the ESN's ability to generalise and predict output signals for new testing input sequences

[1317]:
```
def predict_reservoir(Wout, W, Win, u):
    x = np.zeros(W.shape[0])
    y = np.zeros((u.shape[0], Wout.shape[1]))

    # Loop for all time  series
    for i in range(u.shape[0]):
        # Evaluate the reservoir at the current time step
        x = np.tanh(np.dot(W, x) + np.dot(Win, u[i, :]))
        y[i, :] = np.dot(Wout.T, x)
    return y
```

1.4.3 2.3 ESN Training

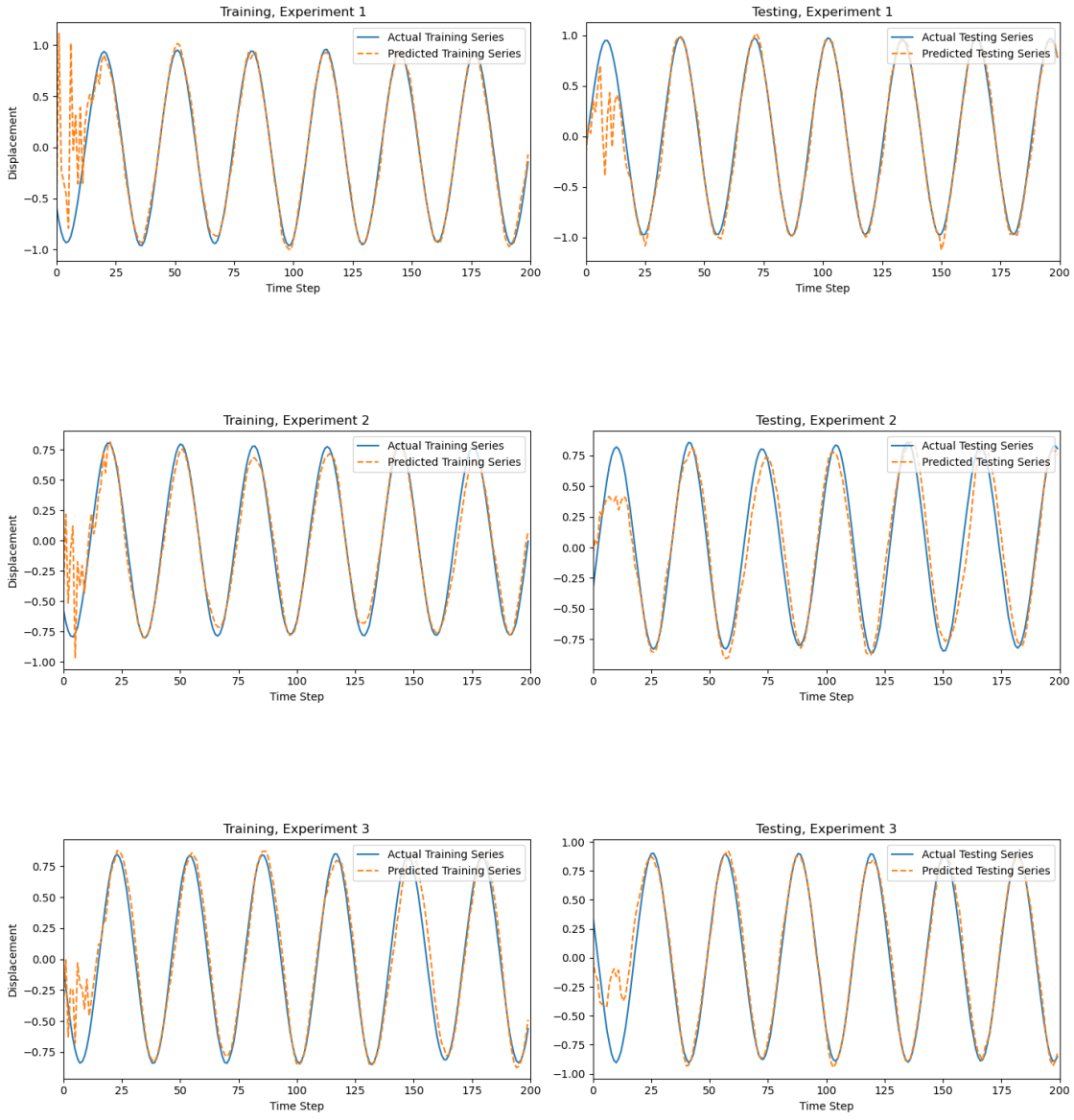
In the following seegment, the initial trained ESN model was trained to evaluate the weight matrices W_{in}, W_{out}, W. This trained ESN was then applied and the model's performance was evaluated by comparing its predicted output to the actual time series for both (a) a training set and (b) a testing set. Both the predicted and actual time series were plotted to assess the ESN's ability to capture the dynamics of the system for unseen data.

[1376]:
```
# ESN Hyperparameters
#--------------------------#
reservoir_size = 100    # Number of neurons
input_size = 1          # DImension of time series input
spectral_radius = 0.9   # Spectral Radius
washout_period = 20     # Indices to ignore for ESN transience
ridge_alpha = 1e-6      # Regularisation parameter
sparsity = 0.2          # Reservoir sparsity
#--------------------------#
```

[1378]:
```
# 1. Creating reservoir
W, Win = create_reservoir(size=reservoir_size, in_size=input_size,␣
 ↪spectral_radius=spectral_radius,sparsity = sparsity)

# 2. Training loop for all u series
train_states = []
for i in range(u_train.shape[0]):
    states = run_reservoir(W, Win, u_train[i].reshape(-1, 1))
    train_states.append(states[washout_period:])

all_x_train = np.vstack(train_states)
y_train = np.vstack([x[washout_period:].reshape(-1, 1) for x in x_train])

# 3. Run ESN Training
Wout = train_reservoir(all_x_train, y_train,ridge_alpha=ridge_alpha)
```

[1380]:
```
#Predictions for a set of ESN experiments
for exp in range(0,3):
    train_pred = predict_reservoir(Wout, W, Win, u_train[exp].reshape(-1, 1))
    test_pred = predict_reservoir(Wout, W, Win, u_test[exp].reshape(-1, 1))

    # Visualise training predictions
    plt.figure(figsize=(14, 4))
    plt.subplot(1, 2, 1)
    plt.plot(x_train[exp], label="Actual Training Series")
    plt.plot(train_pred, label="Predicted Training Series", linestyle='--')
    plt.xlabel("Time Step")
    plt.ylabel("Displacement")
    plt.xlim(0,200)
    plt.title(f"Training, Experiment {exp+1}")
```

```
    plt.legend(loc='upper right')

    #Testing
    plt.subplot(1, 2, 2)
    plt.plot(x_test[exp], label="Actual Testing Series")
    plt.plot(test_pred, label="Predicted Testing Series", linestyle='--')
    plt.xlabel("Time Step")
    plt.xlim(0,200)
    plt.title(f"Testing, Experiment {exp+1}")
    plt.legend(loc='upper right')

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



The plots above illustrate the performance of the ESN model, compared against time series for both training and testing datasets for sample time series. The training plots (left column) show a confident fit, with the predicted series closely aligning with the actual data, indicating strong learning capability. In the testing plots (right column), while the model generalises well, minor deviations occur at the start of the time series. This region is explained by the ESN being in a transient stage, for which the reservoir has not fully stabilised. Overall, the ESN demonstrates robust predictive accuracy across multiple experiments, maintaining phase and amplitude consistency in both datasets.

Evaluating the performance of the ESN was done was calculating the Mean Squared Error (MSE) for all training and testing sets separately. The MSE is formulated as [9]

$$\text{MSE} = \frac{1}{T-k}\sum_{t=k}^{T-1}(x_t - \hat{x}_t)^2$$

for each time series, where k is the washout period, subtracted for the total entries in a time series T. The MSE acts as a quantitive measure of performance and generalisability. Note the use of the `washout_period` parameter, which is the index upto which calculation of the error is excluded to account for the transient initiation phase.

[1382]:
```
# MSE over training set
train_mse_list = []
for i in range(u_train.shape[0]):
    train_pred_series = predict_reservoir(Wout, W, Win, u_train[i].reshape(-1,␣
 ↪1))
    train_mse_list.append(mean_squared_error(x_train[i][washout_period:],␣
 ↪train_pred_series[washout_period:]))

train_mse = np.mean(train_mse_list)
train_mse_std = np.std(train_mse_list)


# MSE over testing set
test_mse_list = []
for i in range(u_test.shape[0]):
    test_pred_series = predict_reservoir(Wout, W, Win, u_test[i].reshape(-1, 1))
    test_mse_list.append(mean_squared_error(x_test[i][washout_period:],␣
 ↪test_pred_series[washout_period:]))

test_mse = np.mean(test_mse_list)
test_mse_std = np.std(test_mse_list)

print(f"Average Training series MSE: {train_mse},Standard Deviation:␣
 ↪{train_mse_std}")
print(f"Average Testing series MSE: {test_mse}, Standard Deviation:␣
 ↪{test_mse_std}")
```

```
Average Training series MSE: 0.007058611022131908,Standard Deviation:
0.005774009066660622
Average Testing series MSE: 0.006241305055424697, Standard Deviation:
0.005521492013747539
```

The similar average MSE values between training and testing data implies that the current ESN configuration generalises well to the unseen test data and suggests that the model is able to effectively predict unseen sequences.

The low MSE values indicate that the ESN captures the underlying dynamics of the system accurately, and despite the nonlinear nature of the NTMD, the model can represent these dynamics accurately, with the current ESN configuration. The relatively low reservoir size and introduction of regularisation have likely contributed to the lack of overfitting, despite the limited quantity of data.

1.4.4 2.4 ESN Validation

One method to validate the weights generated by the ESN and its ability to generalise is to calculate the MSE across different subsets of the data set, in a way that parallels bootstrapping techniques. This is known as K-fold validation, for which the procedure is as follows [10]

1. The data is divided into k number of fold for training, for which the remainder is used for validation purposes
2. For each training fold, reservoir creation and ESN training is performed and the MSE is computed
3. Predictions made by this model are used to compute an MSE value for the validation (testing) fold to assess generalisability

For nonlinear systems, this validation serves to rigorously assess quantitatively the ESN's robustness to variations between u and x.

[1387]:
```
# K-Fold Cross Validation
#--------------------------#
folds = 5  # Number of folds
#--------------------------#

kf = KFold(n_splits=folds, shuffle=True, random_state=2143062)

train_mse_folds = []
val_mse_folds = []

# Cross validation iteration
for train_index, val_index in kf.split(u_train):
    # Split data into training and validation sets for this fold
    u_train_fold, u_val_fold = u_train[train_index], u_train[val_index]
    x_train_fold, x_val_fold = x_train[train_index], x_train[val_index]

    # Create reservoir with current parameters
    W, Win = create_reservoir(size=reservoir_size, in_size=1,␣
 ↪spectral_radius=spectral_radius)
```

```
    # Train the ESN on the training fold
    train_states = []
    for i in range(u_train_fold.shape[0]):
        states = run_reservoir(W, Win, u_train_fold[i].reshape(-1, 1))
        train_states.append(states[washout_period:])  # Apply washout
    all_x_train = np.vstack(train_states)
    y_train = np.vstack([x[washout_period:].reshape(-1, 1) for x in␣
↪x_train_fold])

    # Train output weights
    Wout = train_reservoir(all_x_train, y_train)

    # Training MSE for fold
    train_pred = np.vstack([predict_reservoir(Wout, W, Win, u.reshape(-1,␣
↪1))[washout_period:] for u in u_train_fold])
    train_mse = mean_squared_error(y_train, train_pred)
    train_mse_folds.append(train_mse)

    # Testing MSE for fold to validate
    val_pred = np.vstack([predict_reservoir(Wout, W, Win, u.reshape(-1,␣
↪1))[washout_period:] for u in u_val_fold])
    y_val = np.vstack([x[washout_period:].reshape(-1, 1) for x in x_val_fold])
    val_mse = mean_squared_error(y_val, val_pred)
    val_mse_folds.append(val_mse)

# Calculate average training and validation MSE across all folds
avg_train_mse = np.mean(train_mse_folds)
std_train_mse = np.std(train_mse_folds)
avg_val_mse = np.mean(val_mse_folds)
std_val_mse = np.std(val_mse_folds)

print(f"Average Training MSE: {avg_train_mse} (Std Dev: {std_train_mse})")
print(f"Average Validation MSE: {avg_val_mse} (Std Dev: {std_val_mse})")
```

```
Average Training MSE: 0.005927219976210901 (Std Dev: 0.0014716041982470746)
Average Validation MSE: 0.008378108074088158 (Std Dev: 0.002498474566999234)
```

Training MSE reflects how well the model fits the data it was trained on in each fold, while validation MSE reflects how well the model performs on unseen data within each fold. The relatively small values indicate the ESN effectively captures the NTMD system's nonlinear dynamics with low training error, while the slightly higher validation error suggests good but not perfect generalisation.

1.5 3. Hyperparameter Tuning

1.5.1 3.1 Hyperparameters for ESNs

Hyperparameter tuning is critical for optimising the performance of any ANN, as their flexible architecture increases its dependence on proper parameter selection as they govern the behaviour

of models which are specific to the characteristics of the system they are intended to predict. For more complex networks, a large number of parameters must be optimised, in both the architecture and within training (as explicitly shown in Section 4). For illustration purposes and demonstration of proven techniques, a comparison of four of the following ESN parameters was assessed:

`reservoir_size` (N): Dimension of neurones in the hidden layer. - Low: Fewer neurones in the reservoir, leading to reduced capacity to capture complex nonlinear dynamics, potentially underfitting the data - High: Larger reservoir enables greater expressive power and capacity to model such patterns but increases computational cost and risk of overfitting, especially with limited training data

`spectral_radius` (ρ): Largest eigenvalue of the reservoir weight matrix - Low: Reservoir dynamics fade quickly, reducing memory and the ability to capture long-term dependencies - High: Reservoir may become unstable, leading to chaotic behaviour and loss of predictive accuracy

`ridge_alpha` (α): Regularisation parameter for the ridge regression used in training the output weights - Low: Minimal regularisation, potentially overfitting the training data, especially with noisy datasets or small training sets - High: Strong regularisation, reducing overfitting but potentially underfitting if the model is overly constrained

`sparsity` (S: Fraction of nonzero connections in the reservoir weight matrix. - Low: Fewer connections, leading to a simpler reservoir with reduced capacity to capture complex relationships - High: Dense reservoir, increasing capacity but risking redundancy and higher computational costs

1.5.2 3.2 Optimisation

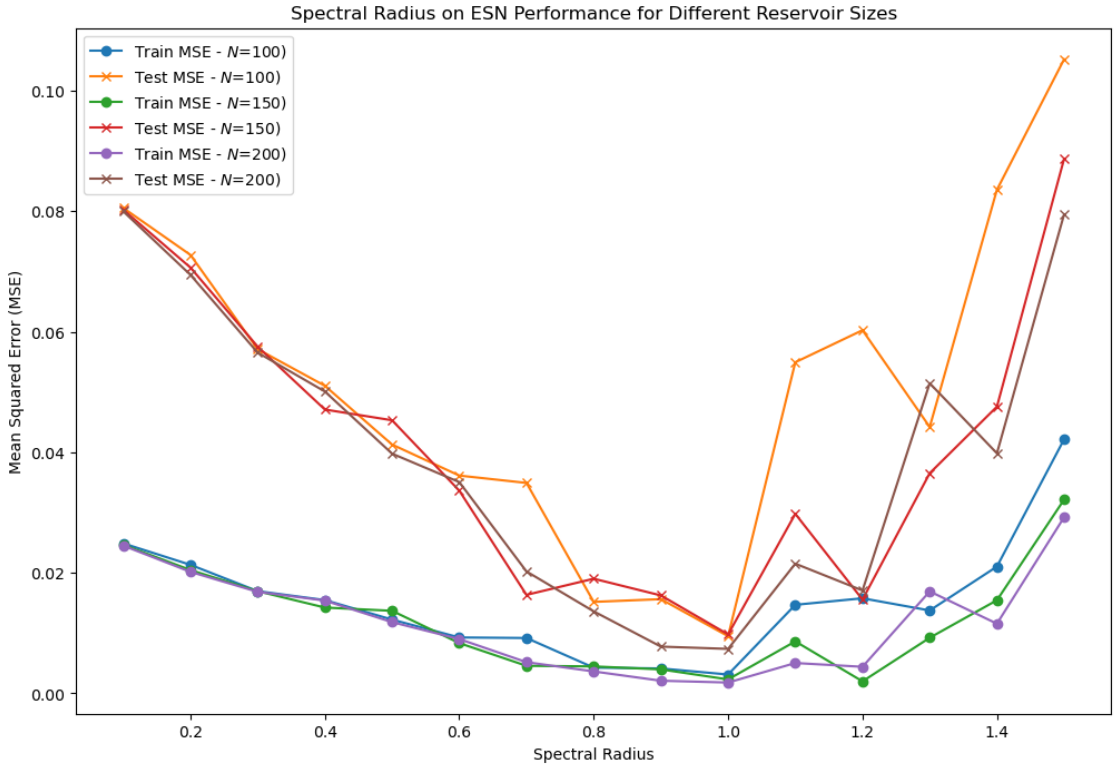
As a preliminary investigation, the impacts of `reservoir_size` and `spectral_radius` were compared. These two were chosen due to their large impact on model performance [11] . The visual below was formed using conventional manual search training and testing iterations over a range of specified parameter lists.

[1336]:
```
# Optimisation of Reservoir sizes against Spectral Radius
#-------------------------#
reservoir_sizes_list = [ 100, 150,200]
spectral_radius_list =  np.linspace(0.1,1.5,15)
#-------------------------#

train_mse_list_size = []
test_mse_list_size = []

# Iterate over reservoir sizes
for reservoir_size in reservoir_sizes_list:
    train_mse_list_res = []
    test_mse_list_res = []


    for spectral_radius in spectral_radius_list:
        # Create reservoir
        W_res, Win_res = create_reservoir(size=reservoir_size,␣
↪in_size=input_size, spectral_radius=spectral_radius, sparsity=sparsity)
```

```
        # Train
        train_states_res = []
        for i in range(len(u_train)):
            states_res = run_reservoir(W_res, Win_res, u_train[i].reshape(-1, 1))
            train_states_res.append(states_res[washout_period:])
        all_x_train_res = np.vstack(train_states_res)
        y_train_res = np.vstack([x[washout_period:].reshape(-1, 1) for x in␣
↪x_train])

        # Train reservoir output weights
        Wout_res = train_reservoir(all_x_train_res, y_train_res,␣
↪ridge_alpha=ridge_alpha)

        # Evaluate on training data
        train_mse_res = []
        for i in range(len(u_train)):
            train_pred_series_res = predict_reservoir(Wout_res, W_res, Win_res,␣
↪u_train[i].reshape(-1, 1))
            train_mse_res.append(mean_squared_error(x_train[i][washout_period:],␣
↪train_pred_series_res[washout_period:]))
        train_mse_list_res.append(np.mean(train_mse_res))

        # Evaluate on testing data
        test_mse_res = []
        for i in range(len(u_test)):
            test_pred_series_res = predict_reservoir(Wout_res, W_res, Win_res,␣
↪u_test[i].reshape(-1, 1))
            test_mse_res.append(mean_squared_error(x_test[i][washout_period:],␣
↪test_pred_series_res[washout_period:]))
        test_mse_list_res.append(np.mean(test_mse_res))

    # Store results for the current reservoir size
    train_mse_list_size.append(train_mse_list_res)
    test_mse_list_size.append(test_mse_list_res)
```

[1337]:
```
# Visualisation of MSE performance for reservoir sizes and spectral radii
plt.figure(figsize=(12, 8))
for i, reservoir_size in enumerate(reservoir_sizes_list):
    plt.plot(spectral_radius_list, train_mse_list_size[i], label=f"Train MSE -␣
↪$N$={reservoir_size})", marker='o')
    plt.plot(spectral_radius_list, test_mse_list_size[i], label=f"Test MSE -␣
↪$N$={reservoir_size})", marker='x')

plt.xlabel("Spectral Radius")
plt.ylabel("Mean Squared Error (MSE)")
```

```
plt.title("Spectral Radius on ESN Performance for Different Reservoir Sizes")
plt.legend()
plt.show()
```



The relationship of the MSE against spectral radius ρ is consistent and evident across all reservoir sizes N. As the radius increases, the error decreases initially as the model performance improves, reaching an optimal at $0.8 < \rho < 1$. Beyond 1.0, performance quickly decreases as expected due to loss of the Echo state property, resulting in chaotic reservoir dynamics and instability.
Comparing reservoir sizes show optimal performance with higher reservoir sizes of $N = 200$, which shows an increase leads to the ESN generally achieving a lower MSE.

Grid Search

Manual hyperparameter tuning is time-consuming and prone to human error, and often suboptimal due to its inability to systematically explore the parameter space. The following section applies an automated search process known as grid search [12], evaluating the performance over a specified range of parameters systematically.

The `product()` function was applied, generating the Cartesian product of the specified parameter values below. The ESN model was then trained for each combination of hyperparameters and the MSE for both training and testing sets was evaluated for comparison.

[1353]:
```
# Parameter List
#--------------------------#
```

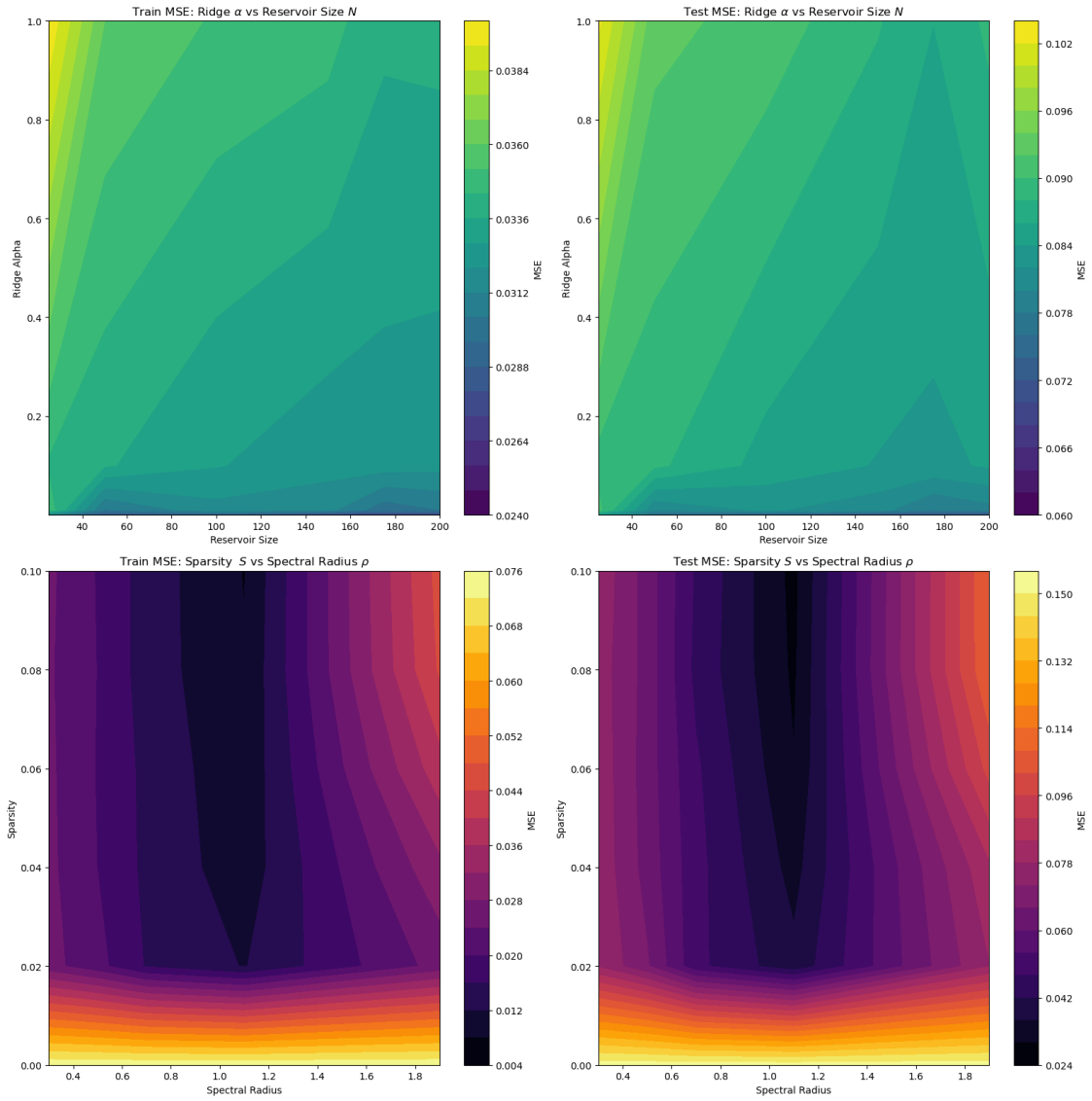
```
reservoir_sizes = [25,50, 100, 150,175,200]
spectral_radii = [0.3,0.5,0.7, 1.9, 1.1]
ridge_alphas = [1e-8,1e-6, 1e-4, 1e-2,1e-1,1e0]
sparsities = [0,0.02, 0.04, 0.06, 0.08,0.1]
#--------------------------#
```

[1342]:
```
# Grid Search for hyperparameter optimisation
optimisation_results = []

for size, radius, alpha, sparsity in product(reservoir_sizes, spectral_radii,␣
↪ridge_alphas, sparsities):
    # Create reservoir
    W_op, Win_op = create_reservoir(size=size, in_size=input_size,␣
↪spectral_radius=radius, sparsity=sparsity)

    # Training ESN
    train_states_op = []
    for i in range(len(u_train)):
        states_op = run_reservoir(W_op, Win_op, u_train[i].reshape(-1, 1))
        train_states_op.append(states_op[washout_period:])
    all_x_train_op = np.vstack(train_states_op)
    y_train_op = np.vstack([x[washout_period:].reshape(-1, 1) for x in x_train])

    Wout_op = train_reservoir(all_x_train_op, y_train_op, ridge_alpha=alpha)

    # MSE Train
    train_mse_list_op = []
    for i in range(len(u_train)):
        train_pred_series_op = predict_reservoir(Wout_op, W_op,␣
↪Win_op,u_train[i].reshape(-1, 1))
        train_mse_list_op.append(mean_squared_error(x_train[i][washout_period:],␣
↪train_pred_series_op[washout_period:]))
    train_mse_op = np.mean(train_mse_list_op)

    # MSE Test
    test_mse_list_op = []
    for i in range(len(u_test)):
        test_pred_series_op = predict_reservoir(Wout_op, W_op, Win_op, u_test[i].
↪reshape(-1, 1))
        test_mse_list_op.append(mean_squared_error(x_test[i][washout_period:],␣
↪test_pred_series_op[washout_period:]))
    test_mse_op = np.mean(test_mse_list_op)

    # Record results
    optimisation_results.append({
        "reservoir_size": size,
        "spectral_radius": radius,
```

```
        "ridge_alpha": alpha,
        "sparsity": sparsity,
        "train_mse": train_mse_op,
        "test_mse": test_mse_op
    })

# Convert to pd dataframe
results_df = pd.DataFrame(optimisation_results)
```

[1419]:
```
# Visualisation of heatmaps for various parameters
fig, axs = plt.subplots(2, 2, figsize=(16, 16))

# Train MSE - ridge_alpha vs reservoir_size
pivot_table_train_1 = results_df.pivot_table(index="ridge_alpha",␣
 ↪columns="reservoir_size", values="train_mse")
train_contour_1 = axs[0, 0].contourf(
    pivot_table_train_1.columns,
    pivot_table_train_1.index,
    pivot_table_train_1.values,
    cmap="viridis",
    levels=20
)
cbar_train_1 = fig.colorbar(train_contour_1, ax=axs[0, 0])
cbar_train_1.set_label("MSE")
axs[0, 0].set_title("Train MSE: Ridge $\\alpha$ vs Reservoir Size $N$")
axs[0, 0].set_xlabel("Reservoir Size")
axs[0, 0].set_ylabel("Ridge Alpha")

# Test MSE - ridge_alpha vs reservoir_size
pivot_table_test_1 = results_df.pivot_table(index="ridge_alpha",␣
 ↪columns="reservoir_size", values="test_mse")
test_contour_1 = axs[0, 1].contourf(
    pivot_table_test_1.columns,
    pivot_table_test_1.index,
    pivot_table_test_1.values,
    cmap="viridis",
    levels=20
)
cbar_test_1 = fig.colorbar(test_contour_1, ax=axs[0, 1])
cbar_test_1.set_label("MSE")
axs[0, 1].set_title("Test MSE: Ridge $\\alpha$ vs Reservoir Size $N$")
axs[0, 1].set_xlabel("Reservoir Size")
axs[0, 1].set_ylabel("Ridge Alpha")

# Train MSE - sparsity vs spectral_radius
pivot_table_train_2 = results_df.pivot_table(index="sparsity",␣
 ↪columns="spectral_radius", values="train_mse")
```

```
train_contour_2 = axs[1, 0].contourf(
    pivot_table_train_2.columns,
    pivot_table_train_2.index,
    pivot_table_train_2.values,
    cmap="inferno",
    levels=20
)
cbar_train_2 = fig.colorbar(train_contour_2, ax=axs[1, 0])
cbar_train_2.set_label("MSE")
axs[1, 0].set_title("Train MSE: Sparsity  $S$ vs Spectral Radius $\\rho$")
axs[1, 0].set_xlabel("Spectral Radius")
axs[1, 0].set_ylabel("Sparsity")

# Test MSE - sparsity vs spectral_radius
pivot_table_test_2 = results_df.pivot_table(index="sparsity",␣
 ↪columns="spectral_radius", values="test_mse")
test_contour_2 = axs[1, 1].contourf(
    pivot_table_test_2.columns,
    pivot_table_test_2.index,
    pivot_table_test_2.values,
    cmap="inferno",
    levels=20
)
cbar_test_2 = fig.colorbar(test_contour_2, ax=axs[1, 1])
cbar_test_2.set_label("MSE")
axs[1, 1].set_title("Test MSE: Sparsity $S$ vs Spectral Radius $\\rho$")
axs[1, 1].set_xlabel("Spectral Radius")
axs[1, 1].set_ylabel("Sparsity")

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



The heatmaps above illustrate the influence of hyperparameters on the training and testing MSE performance for the ESN. Increasing the reservoir size generally reduces MSE, behaviour consistent with one demonstrated in the earlier plot with diminishing returns after a certain size due to overfitting on the training set. Similarly, lower values of $\alpha < 0.2$ improve performance, indicating minimal regularisation is optimal for this setup. The relationship between sparsity and spectral radius is interesting, and highlights that low $S < 0.05$ combined with a $\rho \approx 1$ yields the best results, aligning with the Echo State Property. However, very high $\rho > 1.2$ values degrade performance due to instability.

Bayesian Optimisation

Though grid search is useful for evaluating large ranges, specific evaluation can become increasingly computationally expensive even for the ESN model. As an exploratory method to explore the effects of hyperparameters, it is a good visualisation tool, however the grid will exhaustively search and

ignore redundant combinations - combining explorative and exploitative searching.

In contrast, Bayesian Optimisation methods utilise probabilistic models to estimate the objective function $f(x)$ to search the parameter space adaptively [13] which reduces computational overhead. This was implemented using the `Optuna` framework. First, an objective function `objective()` was defined using suggested trial values within defined ranges for each parameter. The ESN is constructed using sampled hyperparameters as before, with the aim of minimising the MSE. Over `n_trials`, the objective function was run to return the optimal hyperparameters found.

[1389]:
```
# Optuna objective function
def objective(trial):
    # Suggest values for each hyperparameter
    reservoir_size = trial.suggest_int("reservoir_size", 25, 300)
    spectral_radius = trial.suggest_float("spectral_radius", 0.1, 1.5)
    ridge_alpha = trial.suggest_float("ridge_alpha", 1e-12, 1e0, log=True)
    sparsity = trial.suggest_float("sparsity", 0.0, 0.4)

    # Create reservoir
    W, Win = create_reservoir(size=reservoir_size, in_size=input_size,␣
 ↪spectral_radius=spectral_radius, sparsity=sparsity)

    # Train
    train_states = []
    for i in range(len(u_train)):
        states = run_reservoir(W, Win, u_train[i].reshape(-1, 1))
        train_states.append(states[washout_period:])
    all_x_train = np.vstack(train_states)
    y_train = np.vstack([x[washout_period:].reshape(-1, 1) for x in x_train])

    # Train weights
    Wout = train_reservoir(all_x_train, y_train, ridge_alpha=ridge_alpha)

    # MSE Test
    test_mse = []
    for i in range(len(u_test)):
        test_pred_series = predict_reservoir(Wout, W, Win, u_test[i].reshape(-1,␣
 ↪1))
        test_mse.append(mean_squared_error(x_test[i][washout_period:],␣
 ↪test_pred_series[washout_period:]))

    return np.mean(test_mse)

# Create study
study = optuna.create_study(direction="minimize")
study.optimize(objective, n_trials=50, timeout=600)
best_params = study.best_params
best_value = study.best_value
```

```
print("Optimal Hyperparameters:")
for param, value in best_params.items():
    print(f"{param}: {value}")
print(f"Optimal Test MSE: {best_value}")
```

```
Optimal Hyperparameters:
reservoir_size: 277
spectral_radius: 1.0712450776351856
ridge_alpha: 1.6821893517096605e-06
sparsity: 0.12894353169268413
Optimal Test MSE: 0.0018412699440084927
```

Generalisation Approaches

In addition to optimal hyperparameter tuning, other methods to imrpove the generalisability of the ESN at the higher level include:

Input noise augmentation:
Small Gaussian noise can be added to the input x during training, to simulate experimental variability and aid the ESN in recognising underlying patterns consistent over data sets. This approach parallels dropout regularisation, as both apply the concept of stochastic regularisation, which forces the ESN to focus on identifying the underlying structure of the data as opposed to on noise-free inputs.

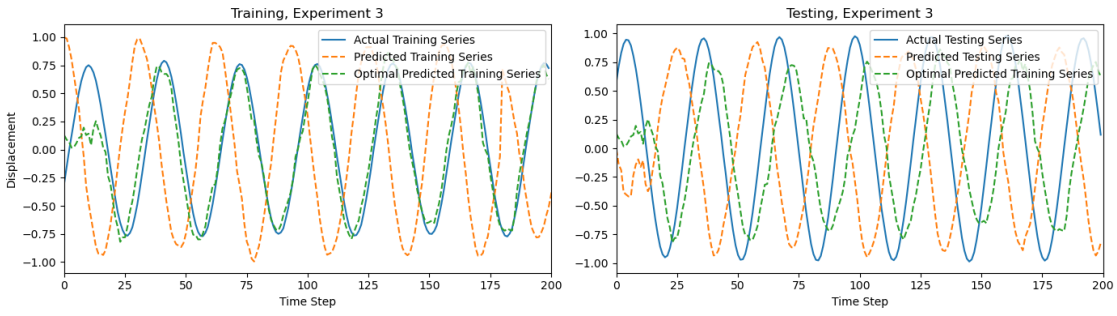
Ensemble ESNs:
Training an ensemble of ESNs involves creating multiple networks with slightly varied initialisations or parameter settings and averaging their predictions. This method can be applied to reduce intrinsic variance in relying on a single ESN model, and leverages the learning patterns of multiple models, making the overall prediction more robust and less prone to overfitting caused by individual ESN configurations. This can also extend to training separate ESNs or tailoring reservoirs for high- and low-amplitude series, before aggregating their predictions based on the characteristics of the input data.

1.5.3 3.3 Performance

These optimised hyperparameters were tested for both sample prediction test sequences, and for the full normalised data set.

[1421]:
```
exp_op = 6
# Training with optimised parameters
W_x, Win_x = create_reservoir(size=best_params["reservoir_size"], in_size=1,␣
 ↪spectral_radius=best_params["spectral_radius"],␣
 ↪sparsity=best_params["sparsity"])
train_states_x = []
for i in range(u_train.shape[0]):
    states_x = run_reservoir(W_x, Win_x, u_train[i].reshape(-1, 1))
    train_states_x.append(states_x[washout_period:])

all_x_train_x = np.vstack(train_states_x)
```

```
y_train_x = np.vstack([x[washout_period:].reshape(-1, 1) for x in x_train])

Wout_x = train_reservoir(all_x_train_x,␣
↪y_train_x,ridge_alpha=best_params["ridge_alpha"])

train_pred_x = predict_reservoir(Wout_x, W_x, Win_x, u_train[exp_op].reshape(-1,␣
↪1))
test_pred_x = predict_reservoir(Wout_x, W_x, Win_x, u_test[exp_op].reshape(-1,␣
↪1))

# Visualise training predictions
plt.figure(figsize=(14, 4))
plt.subplot(1, 2, 1)
plt.plot(x_train[exp_op], label="Actual Training Series")
plt.plot(train_pred, label="Predicted Training Series", linestyle='--')
plt.plot(train_pred_x, label="Optimal Predicted Training Series", linestyle='--')
plt.xlabel("Time Step")
plt.ylabel("Displacement")
plt.xlim(0,200)
plt.title(f"Training, Experiment {exp+1}")
plt.legend(loc = "upper right")

#Testing
plt.subplot(1, 2, 2)
plt.plot(x_test[exp_op], label="Actual Testing Series")
plt.plot(test_pred, label="Predicted Testing Series", linestyle='--')
plt.plot(train_pred_x, label="Optimal Predicted Training Series", linestyle='--')
plt.xlabel("Time Step")
plt.xlim(0,200)
plt.title(f"Testing, Experiment {exp+1}")
plt.legend(loc = "upper right")

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



[1425]:
```
# Training on entire data set
full_W, full_Win = create_reservoir(size=best_params["reservoir_size"],␣
↪in_size=input_size, spectral_radius=best_params["spectral_radius"],sparsity =␣
↪best_params["sparsity"])

all_states_full = []
for i in range(u_norm.shape[0]):
    states_full = run_reservoir(full_W, full_Win, u_norm[i].reshape(-1, 1))
    all_states_full.append(states_full[washout_period:])

all_x_full = np.vstack(all_states_full)
y_full = np.vstack([x[i][washout_period:].reshape(-1, 1) for i in range(x_norm.
↪shape[0])])

full_Wout = train_reservoir(all_x_full,␣
↪y_full,ridge_alpha=best_params["ridge_alpha"])

mse_list_full = []
for i in range(u.shape[0]):
    pred_series_full = predict_reservoir(full_Wout, full_W, full_Win, u[i].
↪reshape(-1, 1))
    mse_list_full.append(mean_squared_error(x_norm[i][washout_period:],␣
↪pred_series_full[washout_period:]))
overall_mse_full = np.mean(mse_list_full)

print(f"Average MSE: {overall_mse_full}")
```

```
Average MSE: 3604652.1501764115
```

Performance Evaluation

On the full data set, the average MSE is substantially higher indicating the model is unable to generalise to the entire data set. There are some possible explanations for this, stemming for the nonlinear behaviours exhibited by the data set variances explored in the data from Section 1.

Fundamentally, the NTMD system is nonlinear in nature, leading to different beheaviours across different inputs and initial conditions which have been identified into several regimes. The ESN was trained on exclusively high variance data of $\sigma_u > 42$, the model was only exposed to specific high-energy dynamics - areas exhibiting large oscillations and, as identified, spring saturation.

As ESNs are sensitive to scale and variance of input signals even after normlisation, this tailors the model to only capture behaviour within the scope. When trained on low variance data which are likely dominated by damping effects, which are governed by different dynamics and thus map x to u through different interactions. When the ESN is trained on the entire dataset dynamics differ significantly from those learned during training on high-variance data appearing as "outliers" or less relevant states to the reservoir. This likely causes dilution in the reservoir's pattern recognitiion, adapting to all noninear regimes resulting in an overall poor performance.

1.6 4. Recurrent Neural Networks

1.6.1 4.1 Data Separation

[1054]:
```
#Training and Testing Split
#-------------------------#
rnn_split = 0.7
#-------------------------#

x_train, x_test, u_train, u_test = train_test_split(x_norm, u_norm, test_size=1␣
 ↪- rnn_split, random_state=42)

print(f"x training data shape: {x_train.shape} ")
print(f"x testing data shape: {x_test.shape} ")
print(f"u training data shape: {u_train.shape} ")
print(f"u testing data shape: {u_test.shape} ")
```

```
x training data shape: (75, 200)
x testing data shape: (33, 200)
u training data shape: (75, 200)
u testing data shape: (33, 200)
```

Tensor Conversion

Post train and test splitting, the matrices X and U are formatted into a single input state to be processed by an RNN. The concatenation step ensures the model is able to learn the coupled dynamics of the system. By converting to PyTorch tensors from NumPy arrays, the transformed structure is compatible with gradient-based optimisation algorithms and tensor operations which require an additional feature dimension for each time step. The operation `unsqueeze()` is used to create an additional feature dimension for both matrices.

[1060]:
```
#Tensor Conversion
#Train
x_train_tensor = torch.tensor(x_train, dtype=torch.float32).unsqueeze(-1)
u_train_tensor = torch.tensor(u_train, dtype=torch.float32).unsqueeze(-1)
input_data_train = torch.cat((x_train_tensor, u_train_tensor), dim=-1)
output_data_train = torch.tensor(x_train, dtype=torch.float32).unsqueeze(-1)

#Test
x_test_tensor = torch.tensor(x_test, dtype=torch.float32).unsqueeze(-1)
u_test_tensor = torch.tensor(u_test, dtype=torch.float32).unsqueeze(-1)

#Combining u and x into an input
input_data_test = torch.cat((x_test_tensor, u_test_tensor), dim=-1)
output_data_test = torch.tensor(x_test, dtype=torch.float32).unsqueeze(-1)

print("Training input data shape:", input_data_train.shape)
print("Testing input data shape:", input_data_test.shape)
```

```
Training input data shape: torch.Size([75, 200, 2])
```

```
Testing input data shape: torch.Size([33, 200, 2])
```

1.6.2 4.2 RNN Formulation

Recurrent Neural Networks (RNNs) are a class of ANNs which, in similar design to ESNs are used to model sequential data by maintaining a memory state which evolves in time. This hidden layer is similar to ESNs in which both possess a large number of neurones dynamic to the input states, and each containing recurrent neural connections, allowing information to persist. More generally, RNNs extend concepts found in ESNs by learning the recurrent connections through backpropagation through time (BPTT), rather than fixing the internal reservoir.

As RNNs are a more general form of architecture, this flexibility is encapsulated through the use of a class `RNN()`, with the following methods

Initialisation `__init__()` The RNN class is initialised with three main components: - Input Layer, which accepts the input states (u, x) for each subsequent time step - The recurrent layer models hidden state evolution, taking the input and previous hidden state to produce the current iteration, determined by

$$h_t = \sigma(W_{ih}x_t + W_{hh}h_{t-1} + b_h)$$

where σ is a nonlinear activation function, W_{hh} are recurrent connection weights (hidden-hidden), W_{ih} are input-hidden weights, and b_h contains bias values for the hidden layer. - The output layer or prediction layer, which linearly maps the hidden state to the output to calculate predictions

Forward Propagation `forward()`
This function dictates the data flow through the network over the input sequence. It is then passed to a nonlinear dropout function, before being processed by the output layer - mapped by the function

$$y_t = W_{ho}h_t + b_o$$

for W_{ho} being hidden-output weights, to compute predictions y_t.

Hidden State Initialisation `init_hidden()`
The hidden state initialisation method enhances the flexibility of the `RNN()` class, allowing initialisation of either a single hidden state zero tensor based on the batch size used in RNN and GRU architectures

$$h_0 = 0$$

, or a multiple cell and hidden state initialisation for LSTMs.

$$(h_0, c_0) = (0, 0)$$

For this architecture, the report chose to also explore nonlinear methods to improve complex temporal relationship learning, notably through activation functions σ in the forward method. The one applied here is a common function known as the Rectified Linear Unit (ReLU)

$$\text{ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

for sparse activation and improved gradient stability [14].
A dropout regulariser was also employed, which led to increased model generalisability through prior testing. The dropout layer is parameterised by a weight p, which acts to randomly deactivate a proportion of hidden states. This introduces stochastic variability and is a form of regularisation.

[1097]:
```
# RNN Setup
class RNN(nn.Module):
    # Initialisation
    def __init__(self, input_size=2, hidden_size=50, output_size=1,␣
 ↪rnn_layers=1,dropout_p = 0.3):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.rnn_layers = rnn_layers
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True,␣
 ↪num_layers=rnn_layers)
        self.fc = nn.Linear(hidden_size, output_size)
        self.activation = nn.ReLU()
        self.dropout = nn.Dropout(p=dropout_p)

    # Forward method
    def forward(self, input_data, hidden):
        out, hidden = self.rnn(input_data, hidden)
        out = self.dropout(out)
        out = self.fc(out)
        #out = self.activation(out)
        return out, hidden

    # Hidden state initialisation
    def init_hidden(self, batch_size):
        if isinstance(self.rnn, nn.LSTM):
            # LSTM
            return (torch.zeros(self.rnn_layers, batch_size, self.hidden_size),
                    torch.zeros(self.rnn_layers, batch_size, self.hidden_size))
        else:
            # RNN and GRU
            return torch.zeros(self.rnn_layers, batch_size, self.hidden_size)
```

[1164]:
```
# Model Hyperparameters
#--------------------------#
input_size = 2                          # Dimension of the concatenated inputs␣
 ↪(u,x)
hidden_size = 64                        # Number of neurons in the hidden layer
output_size = 1                         # Output size - predicting a single␣
 ↪displacement value at each step
rnn_layers = 1                          # Number of stacked RNN layers
dropout_p = 0.3                         # Dropout weight
batch_size = input_data_train.shape[0] # Number of sequences processed␣
 ↪simultaneously
#--------------------------#

# RNN Model
```

```
model = RNN(input_size=input_size, hidden_size=hidden_size,␣
 ↪output_size=output_size, rnn_layers=rnn_layers)

# Hidden state initialisation
hidden = model.init_hidden(batch_size)
```

1.6.3 4.3 RNN Training

Hyperparameters

Hyperparameters are crucial or any ANN, as they directly influence the model's ability to learn and generalise from sequential data. Unlike ESNs, which rely on fixed internal dynamics and only train the output weight W_o, RNNs are fully flexible, making hyperparameter tuning even more critical.

Key hyperparameters determine the network's capacity, convergence rate, and generalisability. For instance, increasing the hidden size enhances the model's expressiveness but also raises the risk of overfitting and raises computational overhead. Similarly, a small learning rate ensures stable convergence but can slow down training, while higher rates may lead to instability. The flexible nature of RNNs makes them highly adaptable to various tasks, but also necessitates careful balancing of hyperparameters to prevent underfitting, overfitting, or inefficiencies during training. While this report explores the significance of these hyperparameters and their roles, excessive optimisation was not performed due to computational limitations, though theoretically can be executed in a similar manner as outlined Section 3.

[1105]:
```
# Training Hyperparameters
#--------------------------#
learning_rate = 0.0005                                                    ␣
 ↪          # Step size for updating model weights
n_epochs = 50                                                             ␣
 ↪          # Number of full passes over training set
criterion = nn.MSELoss()                                                  ␣
 ↪          # Loss function used - MSE
optimizer = optim.Adam(model.parameters(), lr=learning_rate)              ␣
 ↪          # Adam optimiser to update model weights
step_size = 30                                                            ␣
 ↪          # Scheduler steps before reducing learning rate
gamma = 0.25                                                              ␣
 ↪          # Decay factor for reducing learning rate
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=step_size,␣
 ↪gamma=gamma)  # Dynamically adjusts learning rate
lag_weight = 0.9                                                          ␣
 ↪          # Lag weight to smoothen dynamics and phase
#--------------------------#
```

Training Loop

The training loop implements an iterative process to optimise the RNN parameters for predicting sequential data. As follows;

1. The model is set to training mode using `model.train()`, and the hidden state is initialised at the start of each epoch to $h_0 = 0$ For each epoch up to `n_epochs`;
2. Each time series in the data set is analysed. The RNN processes the input, using this to update the hidden state using
$$h_t, y_t = RNN(x_t, h_{t-1})$$
where y_t is the predicted output and h_t the updated hidden state at time t
3. The MSE is calculated between the predicted and target sequences as the primary loss function, using least squares as shown below
$$\mathcal{L}_{\text{MSE}} = \frac{1}{T}\sum_{t=1}^{T} ||y_t - \hat{y}_t||_F^2$$
4. As a phase discrepancy was consistently apparent in training iterations, a lag penalty was introduced, using a regularisation formulation calculated as [15]
$$\mathcal{L}_{\text{lag}} = \frac{1}{T-1}\sum_{t=1}^{T-1} ||y_t - \hat{y}_{t+1}||_F^2$$
such that this is combined with the lag penalty weight, λ. Thus,
$$\mathcal{L} = \mathcal{L}_{\text{MSE}} + \lambda\mathcal{L}_{\text{lag}}$$
5. Gradients are computed for all model parameters $\nabla\mathcal{L}$ by backpropagation. These values are then clipped as per specification, to a maximum norm of 1:
$$||\hat{\nabla}_p|| = \min(||\nabla_p||, 1)$$
where p represents the gradients of parameter p
6. Parameters are then updated using the chosen optimiser - for the purposes of this report, the Adam optimiser was used [16]. A learning rate scheduler was also implemented to adjust the `learning_rate` by a specified `step_size` by the parameter γ.
7. The average gradient norm is calculated, and the total loss for the epoch is accumulated and returned - which is a measure of the RNN convergence over consecutive epochs

To monitor gradient clipping during training, average gradient norms `AGN` values were computed before and after clipping to identify the significance of the gradient being clipped. This also provided insight into the training stability and convergence while validating the model. Gradient norms were calculated using the formula
$$||\nabla|| = \sqrt{\sum_p ||\nabla_p||_F^2}$$

[1108]:
```
#Training RNN
def train_rnn(model, input_data, target_data, n_epochs=n_epochs,␣
↪learning_rate=learning_rate,criterion=criterion,optimizer=optimizer,scheduler=scheduler,␣
↪lag_weight=lag_weight):
```

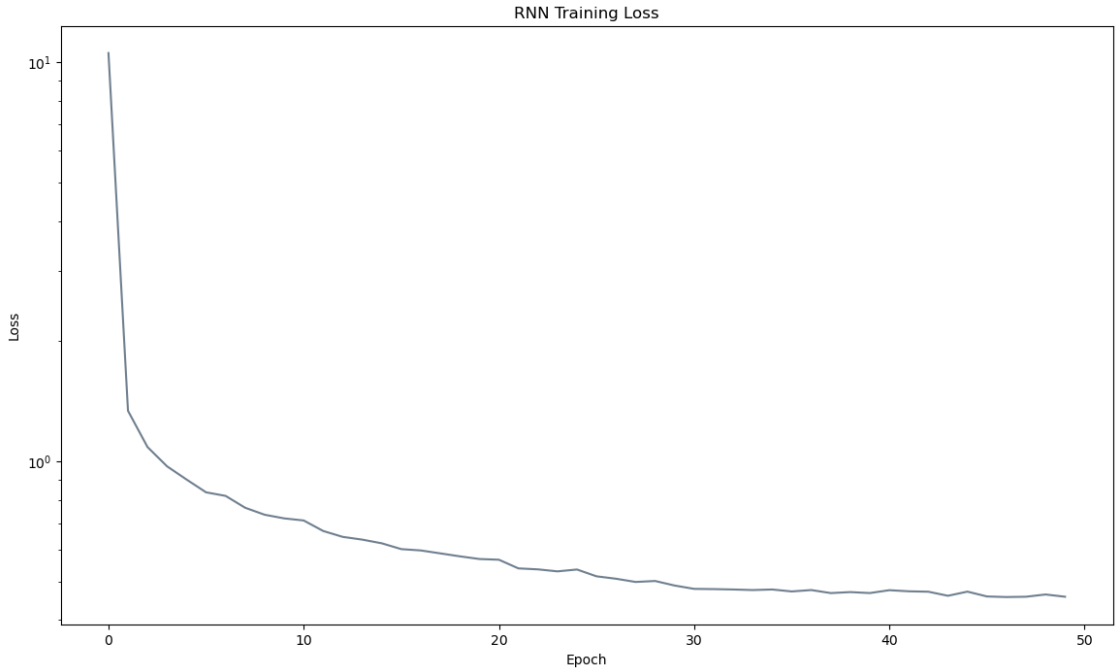
```
    # Track losses over epochs
    losses = []
    GN_before = []
    GN_after = []

    # Training loop
    for epoch in range(n_epochs):
        model.train()  # Set the model to training mode
        hidden = model.init_hidden(input_data.size(0))  # Initialize hidden␣
↪state for each epoch
        epoch_loss = 0.0

        for i in range(input_data.shape[0]):

            input_sequence = input_data[i].unsqueeze(0)
            target_sequence = target_data[i].unsqueeze(0)

            if i == 0:
                hidden = model.init_hidden(batch_size=1)

            # Forward pass
            output_pred, hidden = model(input_sequence, hidden)
            loss = criterion(output_pred, target_sequence)

            # Lag penalty
            if target_sequence.shape[1] > 1:
                lag_penalty = torch.mean((output_pred[:, :-1, :] -␣
↪target_sequence[:, 1:, :]) ** 2)
                total_loss = loss + lag_weight * lag_penalty
            else:
                total_loss = loss

            # Detach hidden state
            if isinstance(hidden, tuple):
                hidden = tuple(h.detach() for h in hidden)
            else:
                hidden = hidden.detach()

            # Backward pass
            optimizer.zero_grad()
            total_loss.backward()

            total_GN_before = torch.norm(torch.stack([torch.norm(p.grad.
↪detach()) for p in model.parameters() if p.grad is not None]))
            GN_before.append(total_GN_before.item())

            # Apply gradient clipping
```

```
            nn.utils.clip_grad_norm_(model.parameters(), max_norm=1)

            total_GN_after = torch.norm(torch.stack([torch.norm(p.grad.detach())␣
↪for p in model.parameters() if p.grad is not None]))
            GN_after.append(total_GN_after.item())

            # Update weights
            optimizer.step()

            # Accumulate loss for this sequence
            epoch_loss += total_loss.item()

        # Step the learning rate scheduler
        scheduler.step()

        losses.append(epoch_loss)

        AGN_before = sum(GN_before)/len(GN_before)
        AGN_after = sum(GN_after)/len(GN_after)

        print(f"Epoch {epoch+1}/{n_epochs}, Loss: {epoch_loss:.4f}, AGN Before:␣
↪{AGN_before:.4f}, AGN After: {AGN_after:.4f}")

        GN_before.clear()
        GN_after.clear()

    return losses

# Train RNN
losses = train_rnn(model, input_data_train,␣
↪output_data_train,learning_rate=learning_rate,criterion=criterion,optimizer=optimizer,schedul
```

```
Epoch 1/50, Loss: 10.5353, AGN Before: 0.6179, AGN After: 0.4746
Epoch 2/50, Loss: 1.3374, AGN Before: 0.2181, AGN After: 0.2181
Epoch 3/50, Loss: 1.0851, AGN Before: 0.1495, AGN After: 0.1495
Epoch 4/50, Loss: 0.9709, AGN Before: 0.1307, AGN After: 0.1307
Epoch 5/50, Loss: 0.8996, AGN Before: 0.1076, AGN After: 0.1076
Epoch 6/50, Loss: 0.8355, AGN Before: 0.1077, AGN After: 0.1077
Epoch 7/50, Loss: 0.8186, AGN Before: 0.1250, AGN After: 0.1250
Epoch 8/50, Loss: 0.7645, AGN Before: 0.1061, AGN After: 0.1061
Epoch 9/50, Loss: 0.7342, AGN Before: 0.0965, AGN After: 0.0965
Epoch 10/50, Loss: 0.7190, AGN Before: 0.0925, AGN After: 0.0925
Epoch 11/50, Loss: 0.7103, AGN Before: 0.1187, AGN After: 0.1187
Epoch 12/50, Loss: 0.6685, AGN Before: 0.0764, AGN After: 0.0764
Epoch 13/50, Loss: 0.6462, AGN Before: 0.0961, AGN After: 0.0961
Epoch 14/50, Loss: 0.6360, AGN Before: 0.0736, AGN After: 0.0736
Epoch 15/50, Loss: 0.6227, AGN Before: 0.0820, AGN After: 0.0820
```

```
Epoch 16/50, Loss: 0.6021, AGN Before: 0.0855, AGN After: 0.0855
Epoch 17/50, Loss: 0.5975, AGN Before: 0.0751, AGN After: 0.0751
Epoch 18/50, Loss: 0.5875, AGN Before: 0.0653, AGN After: 0.0653
Epoch 19/50, Loss: 0.5776, AGN Before: 0.0816, AGN After: 0.0816
Epoch 20/50, Loss: 0.5687, AGN Before: 0.0633, AGN After: 0.0633
Epoch 21/50, Loss: 0.5664, AGN Before: 0.0649, AGN After: 0.0649
Epoch 22/50, Loss: 0.5387, AGN Before: 0.0575, AGN After: 0.0575
Epoch 23/50, Loss: 0.5357, AGN Before: 0.0605, AGN After: 0.0605
Epoch 24/50, Loss: 0.5296, AGN Before: 0.0721, AGN After: 0.0721
Epoch 25/50, Loss: 0.5353, AGN Before: 0.0717, AGN After: 0.0717
Epoch 26/50, Loss: 0.5146, AGN Before: 0.0549, AGN After: 0.0549
Epoch 27/50, Loss: 0.5075, AGN Before: 0.0601, AGN After: 0.0601
Epoch 28/50, Loss: 0.4982, AGN Before: 0.0501, AGN After: 0.0501
Epoch 29/50, Loss: 0.5010, AGN Before: 0.0490, AGN After: 0.0490
Epoch 30/50, Loss: 0.4880, AGN Before: 0.0508, AGN After: 0.0508
Epoch 31/50, Loss: 0.4787, AGN Before: 0.0412, AGN After: 0.0412
Epoch 32/50, Loss: 0.4781, AGN Before: 0.0413, AGN After: 0.0413
Epoch 33/50, Loss: 0.4770, AGN Before: 0.0404, AGN After: 0.0404
Epoch 34/50, Loss: 0.4755, AGN Before: 0.0412, AGN After: 0.0412
Epoch 35/50, Loss: 0.4770, AGN Before: 0.0385, AGN After: 0.0385
Epoch 36/50, Loss: 0.4716, AGN Before: 0.0395, AGN After: 0.0395
Epoch 37/50, Loss: 0.4755, AGN Before: 0.0396, AGN After: 0.0396
Epoch 38/50, Loss: 0.4672, AGN Before: 0.0400, AGN After: 0.0400
Epoch 39/50, Loss: 0.4699, AGN Before: 0.0367, AGN After: 0.0367
Epoch 40/50, Loss: 0.4673, AGN Before: 0.0406, AGN After: 0.0406
Epoch 41/50, Loss: 0.4751, AGN Before: 0.0364, AGN After: 0.0364
Epoch 42/50, Loss: 0.4720, AGN Before: 0.0384, AGN After: 0.0384
Epoch 43/50, Loss: 0.4709, AGN Before: 0.0410, AGN After: 0.0410
Epoch 44/50, Loss: 0.4600, AGN Before: 0.0362, AGN After: 0.0362
Epoch 45/50, Loss: 0.4711, AGN Before: 0.0433, AGN After: 0.0433
Epoch 46/50, Loss: 0.4582, AGN Before: 0.0389, AGN After: 0.0389
Epoch 47/50, Loss: 0.4566, AGN Before: 0.0362, AGN After: 0.0362
Epoch 48/50, Loss: 0.4575, AGN Before: 0.0395, AGN After: 0.0395
Epoch 49/50, Loss: 0.4634, AGN Before: 0.0380, AGN After: 0.0380
Epoch 50/50, Loss: 0.4575, AGN Before: 0.0373, AGN After: 0.0373
```

[1149]:
```python
#Visualisation of Loss against Epochs - RNN
plt.figure(figsize=(14, 8))
plt.axes().set_yscale('log')
plt.plot(range(n_epochs),losses, linestyle='-',color='slategrey')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("RNN Training Loss")
plt.show()
```



The plot of MSE loss against the number of epochs for an RNN is shown above. The loss decreases rapidly in the initial 2-3 epochs, indicating efficient learning of the underlying dynamics. This is followed by a gradual reduction and indication of stabilisation, suggesting convergence to an optimal solution. The absence of fluctuations in the later epochs implies consistent stable learning and minimal overfitting.

1.6.4 4.4 RNN Testing

The RNN performance was be tested by repeatedly refeeding output trajectories into models as inputs - referred to as the process of "rolling out" the model, while retaining memory of states in previous iterations [17]. These predicted trajectories can be compared against the original model and its loss can be validated.

Testing Hyperparameters
Two testing hyperparameters were specified for the unrolling procedure;

- `n_warmup` dictates the number of initial time steps to process known data. As the hidden state is initialised as zero tensors, this can negatively affect the outputs. As a solution, the RNN model hidden states are initialised using known states using the known test sequence data before transitioning to autoregressive means
- `alpha` is a weight factor, added for this to improve the stability of the model and show improvements in predictability by weighing the contribution of predicted and actual input values by the function [18]

$$\text{Input} = \alpha(\text{actual}) + (1 - \alpha)(\text{predicted})$$

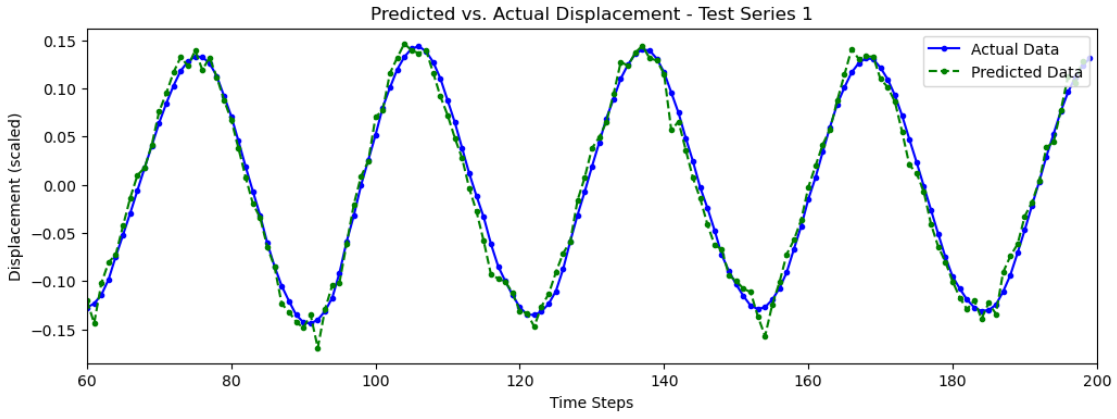
[1226]:
```
# Testing Parameters
#--------------------------#
n_warmup = 60         # Number of time steps to warm-up the model
alpha = 0.2           # Weight factor
#--------------------------#
```

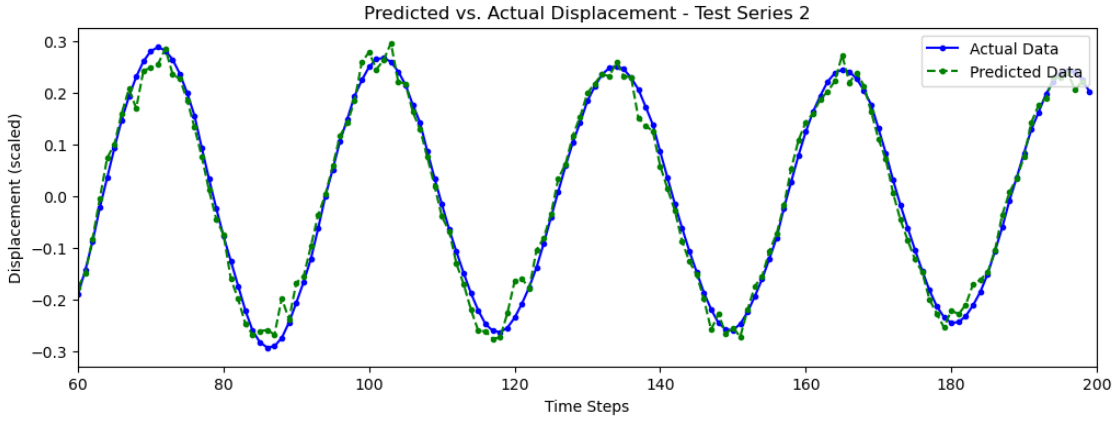
[1115]:
```
# Predict loops and time steps by iterating and feeding back predictions
all_predictions = []
mse_list = []
timeseries = []

with torch.no_grad():

    # Iteration over testing sets
    for i in range(input_data_test.shape[0]):
        test_sequence = input_data_test[i]
        test_sequence = test_sequence.clone().detach()
        actual_data = output_data_test[i]

        warmup_input = test_sequence[:n_warmup].unsqueeze(0)
        timeseries = []

        # Hidden state initialisation
        hidden = model.init_hidden(batch_size=1)

        # Warm-up phase: Initialize hidden state
        next_state, hidden = model(warmup_input, hidden)
        input = next_state[:, -1, :].unsqueeze(0)

        # Predict remaining time steps
        for i in range(len(actual_data) - n_warmup - 1):
            # Extract u feature for the current time step
            u_feature = test_sequence[n_warmup + i, 1].view(1, 1, 1)

            # (x,u) into input
            input = alpha * torch.cat((test_sequence[n_warmup + i, 0].view(1, 1,␣
↪1), u_feature), dim=-1) + (1 - alpha) * input

            # Predict the next state and update hidden state
            input, hidden = model(input, hidden)

            # Append prediction to timeseries
            timeseries.append(input.squeeze(0).numpy())

        timeseries = np.array(timeseries)
        all_predictions.append(timeseries)
```

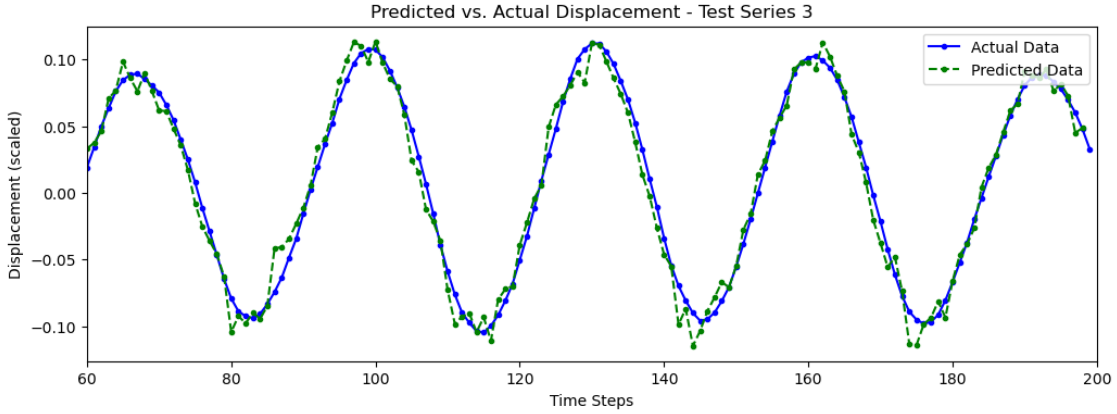
```
        # MSE
        mse = np.mean((actual_data[n_warmup:-1].numpy() - timeseries[:, 0]) ** 2)
        mse_list.append(mse)

# Average MSE over all test sequences
average_mse = np.mean(mse_list)
print(f"Average MSE on Test Set: {average_mse:.6f}")
```

```
Average MSE on Test Set: 0.001790
```

[1222]:
```
# Plot the predicted vs actual time series for the first test sequence
i_data = [0,1,2]  # Index of the sequence to plot
for i in i_data:
    plt.figure(figsize=(12, 4))
    t = np.arange(len(output_data_test[i]))
    plt.plot(t, output_data_test[i].numpy(), label="Actual Data", color="blue",␣
 ↪linestyle="-", marker=".")
    plt.plot(t[n_warmup:n_warmup + len(all_predictions[i])], all_predictions[i][:
 ↪, 0] / scaling, label="Predicted Data", color="green", linestyle="--",␣
 ↪marker=".")
    plt.xlabel("Time Steps")
    plt.ylabel("Displacement (scaled)")
    plt.legend(loc="upper right")
    plt.xlim(n_warmup, len(actual_data))
    plt.title(f"Predicted vs. Actual Displacement - Test Series {i+1}")
    plt.show()
```







The RNN was evaluated on the dataset for a subset of example test series. The model achieved an average MSE of 0.00179 on the test set, indicating good predictive accuracy. The model accurately captures the oscillatory behaviour, including amplitude, frequency, and phase alignment, even when given unseen sequences. Minor discrepancies, such as slight phase shifts or small amplitude mismatches, are negligible and do not significantly affect the overall performance. The RNN effectively learns the nonlinear temporal dependencies inherent in the NTMD system and maintains generalisability over the displayed data sets.

It should be noted that due to the large warmup and α hyperparameters chosen and truncated data set, the prediction states are relatively small sequences which may be too small to demonstrate significant nonlinearities, with significant influence from existing signal states.

1.6.5 4.5 GRU

`nn.GRU()`

The Gated Retrieval Unit (GRU) architecture enhances the standard RNN by introducing gates to manage information flow, improving its ability to handle long-term dependencies and mitigating vanishing gradient problems. The key difference in architecture is the use of gates, which abstractively perform selective memory updates of relevant information. Standard RNNs update

hidden states directly. GRUs however, possess two different gates: - reset gates: $r_t = σ(W_r ·[h_{t−1}, x_t] + b_r)$ − $updategatesz_t = σ(W_z · [h_{t−1}, x_t] + b_z)$
where $σ$ is the sigmoid activation function, and $[h_{t−1}, x_t]$ denotes a concatenation, and b_r, b_z are biases. These gates are integral in the formulation of the hidden state update for h_t [19].

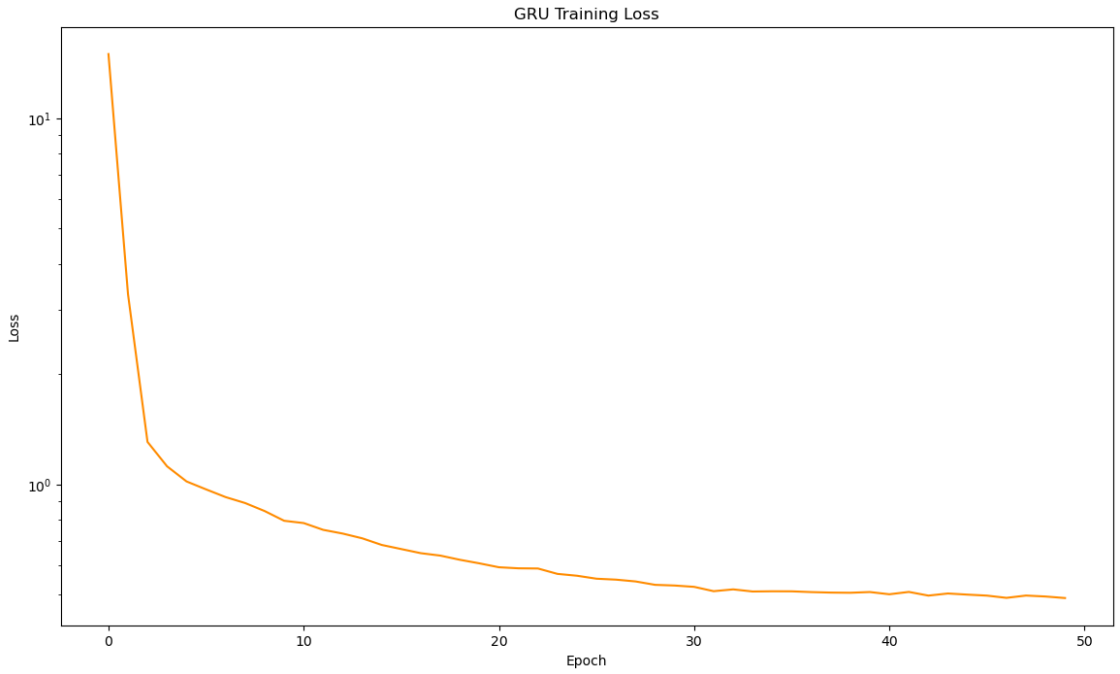
[1157]:
```
# GRU Setup
class GRU(nn.Module):
    # Initialisation
    def __init__(self, input_size=2, hidden_size=50, output_size=1,␣
↪rnn_layers=1,dropout_p = 0.3):
        super(GRU, self).__init__()
        self.hidden_size = hidden_size
        self.rnn_layers = rnn_layers
        self.gru = nn.GRU(input_size, hidden_size, batch_first=True,␣
↪num_layers=rnn_layers)
        self.fc = nn.Linear(hidden_size, output_size)
        self.activation = nn.ReLU()
        self.dropout = nn.Dropout(p=dropout_p)

    # Forward method
    def forward(self, input_data, hidden):
        out, hidden = self.gru(input_data, hidden)
        out = self.dropout(out)
        out = self.fc(out)
        # out = self.activation(out)
        return out, hidden

    # Hidden state initialisation
    def init_hidden(self, batch_size):
        return torch.zeros(self.rnn_layers, batch_size, self.hidden_size)
```

[1125]:
```
# Initialise the GRU model
gru_model = GRU(input_size=input_size, hidden_size=hidden_size,␣
↪output_size=output_size, rnn_layers=rnn_layers)

# Use the same hyperparameters as the RNN
gru_optimizer = optim.Adam(gru_model.parameters(), lr=learning_rate)
gru_scheduler = torch.optim.lr_scheduler.StepLR(gru_optimizer,␣
↪step_size=step_size, gamma=gamma)

# Train GRU
gru_losses = train_rnn(gru_model, input_data_train, output_data_train,
                       n_epochs=n_epochs, learning_rate=learning_rate,
                       criterion=criterion, optimizer=gru_optimizer,
                       scheduler=gru_scheduler, lag_weight=lag_weight)
```

```
Epoch 1/50, Loss: 14.9677, AGN Before: 0.5163, AGN After: 0.4621
Epoch 2/50, Loss: 3.3076, AGN Before: 0.2860, AGN After: 0.2860
```

```
Epoch 3/50, Loss: 1.3048, AGN Before: 0.1263, AGN After: 0.1263
Epoch 4/50, Loss: 1.1192, AGN Before: 0.0948, AGN After: 0.0948
Epoch 5/50, Loss: 1.0175, AGN Before: 0.0835, AGN After: 0.0835
Epoch 6/50, Loss: 0.9679, AGN Before: 0.0791, AGN After: 0.0791
Epoch 7/50, Loss: 0.9221, AGN Before: 0.0754, AGN After: 0.0754
Epoch 8/50, Loss: 0.8882, AGN Before: 0.0620, AGN After: 0.0620
Epoch 9/50, Loss: 0.8446, AGN Before: 0.0632, AGN After: 0.0632
Epoch 10/50, Loss: 0.7945, AGN Before: 0.0671, AGN After: 0.0671
Epoch 11/50, Loss: 0.7831, AGN Before: 0.0634, AGN After: 0.0634
Epoch 12/50, Loss: 0.7504, AGN Before: 0.0636, AGN After: 0.0636
Epoch 13/50, Loss: 0.7330, AGN Before: 0.0699, AGN After: 0.0699
Epoch 14/50, Loss: 0.7114, AGN Before: 0.0577, AGN After: 0.0577
Epoch 15/50, Loss: 0.6824, AGN Before: 0.0584, AGN After: 0.0584
Epoch 16/50, Loss: 0.6649, AGN Before: 0.0605, AGN After: 0.0605
Epoch 17/50, Loss: 0.6480, AGN Before: 0.0580, AGN After: 0.0580
Epoch 18/50, Loss: 0.6381, AGN Before: 0.0581, AGN After: 0.0581
Epoch 19/50, Loss: 0.6216, AGN Before: 0.0612, AGN After: 0.0612
Epoch 20/50, Loss: 0.6078, AGN Before: 0.0583, AGN After: 0.0583
Epoch 21/50, Loss: 0.5930, AGN Before: 0.0581, AGN After: 0.0581
Epoch 22/50, Loss: 0.5891, AGN Before: 0.0544, AGN After: 0.0544
Epoch 23/50, Loss: 0.5884, AGN Before: 0.0563, AGN After: 0.0563
Epoch 24/50, Loss: 0.5688, AGN Before: 0.0632, AGN After: 0.0632
Epoch 25/50, Loss: 0.5622, AGN Before: 0.0552, AGN After: 0.0552
Epoch 26/50, Loss: 0.5518, AGN Before: 0.0424, AGN After: 0.0424
Epoch 27/50, Loss: 0.5486, AGN Before: 0.0442, AGN After: 0.0442
Epoch 28/50, Loss: 0.5424, AGN Before: 0.0572, AGN After: 0.0572
Epoch 29/50, Loss: 0.5310, AGN Before: 0.0479, AGN After: 0.0479
Epoch 30/50, Loss: 0.5287, AGN Before: 0.0522, AGN After: 0.0522
Epoch 31/50, Loss: 0.5243, AGN Before: 0.0339, AGN After: 0.0339
Epoch 32/50, Loss: 0.5101, AGN Before: 0.0380, AGN After: 0.0380
Epoch 33/50, Loss: 0.5160, AGN Before: 0.0370, AGN After: 0.0370
Epoch 34/50, Loss: 0.5091, AGN Before: 0.0329, AGN After: 0.0329
Epoch 35/50, Loss: 0.5098, AGN Before: 0.0338, AGN After: 0.0338
Epoch 36/50, Loss: 0.5097, AGN Before: 0.0324, AGN After: 0.0324
Epoch 37/50, Loss: 0.5073, AGN Before: 0.0364, AGN After: 0.0364
Epoch 38/50, Loss: 0.5057, AGN Before: 0.0331, AGN After: 0.0331
Epoch 39/50, Loss: 0.5051, AGN Before: 0.0343, AGN After: 0.0343
Epoch 40/50, Loss: 0.5074, AGN Before: 0.0351, AGN After: 0.0351
Epoch 41/50, Loss: 0.5004, AGN Before: 0.0368, AGN After: 0.0368
Epoch 42/50, Loss: 0.5078, AGN Before: 0.0367, AGN After: 0.0367
Epoch 43/50, Loss: 0.4962, AGN Before: 0.0348, AGN After: 0.0348
Epoch 44/50, Loss: 0.5028, AGN Before: 0.0377, AGN After: 0.0377
Epoch 45/50, Loss: 0.4992, AGN Before: 0.0346, AGN After: 0.0346
Epoch 46/50, Loss: 0.4961, AGN Before: 0.0350, AGN After: 0.0350
Epoch 47/50, Loss: 0.4892, AGN Before: 0.0349, AGN After: 0.0349
Epoch 48/50, Loss: 0.4963, AGN Before: 0.0367, AGN After: 0.0367
Epoch 49/50, Loss: 0.4933, AGN Before: 0.0320, AGN After: 0.0320
Epoch 50/50, Loss: 0.4886, AGN Before: 0.0335, AGN After: 0.0335
```

[1147]:
```
#Visualisation of Loss against Epochs - GRU
plt.figure(figsize=(14, 8))
plt.axes().set_yscale('log')
plt.plot(range(n_epochs),gru_losses, linestyle='-',color = 'darkorange')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("GRU Training Loss")
plt.show()
```



[1162]:
```
# Evaluate GRU performance on the test set
gru_all_predictions = []
gru_mse_list = []

with torch.no_grad():
    # Loop for data set
    for i in range(input_data_test.shape[0]):
        test_sequence = input_data_test[i]
        test_sequence = scaling * test_sequence.clone().detach()
        actual_data = output_data_test[i]

        # Warmup phase
        warmup_input = test_sequence[:n_warmup].unsqueeze(0)

        # Hidden state initialisation
        gru_timeseries = []
```

```
        hidden = gru_model.init_hidden(batch_size=1)

        # Warm-up phase
        next_state, hidden = gru_model(warmup_input, hidden)
        input = next_state[:, -1, :].unsqueeze(0)

        # Predict remaining time steps
        for i in range(len(actual_data) - n_warmup - 1):
            u_feature = test_sequence[n_warmup + i, 1].view(1, 1, 1)
            input = alpha * torch.cat((test_sequence[n_warmup + i, 0].view(1, 1,␣
↪1), u_feature), dim=-1) + (1 - alpha) * input
            input, hidden = gru_model(input, hidden)
            gru_timeseries.append(input.squeeze(0).numpy())

        gru_timeseries = np.array(gru_timeseries)
        gru_all_predictions.append(gru_timeseries)

        # MSE
        mse = np.mean((actual_data[n_warmup:-1].numpy() - gru_timeseries[:, 0])␣
↪** 2)
        gru_mse_list.append(mse)

# Average MSE for GRU
gru_average_mse = np.mean(gru_mse_list)
```

1.6.6 4.6 LSTM

`nn.LSTM()`
In similar fashion to the GRU, the Long Short Term Memory Unit (LSTM) networks are designed to address issues like vanishing and exploding gradients while effectively learning long-term dependencies. It is more computationally expensive than both GRUs and traditional RNNs, owing to their use of three different gates which also serve to selectively regulate information flow: - Input gates: $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$ $- Forget gates f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$ - Output gates: $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$
in addition to gates, LSTM differs from GRU by maintaining a hidden state and an internal cell state to increase memory retention - which is evolved by

$$c_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

The modification to the `init_hidden()` method accounts for the cell state being initialised as another tuple of zero tensors. where σ is the sigmoid activation function, and $[h_{t-1}, x_t]$ denotes a concatenation, and b_i, b_f, b_o are biases[19]

[1131]:
```
# LSTM Setup
class LSTM(nn.Module):
    # Initialisation
    def __init__(self, input_size=2, hidden_size=50, output_size=1,␣
↪rnn_layers=1,dropout_p = 0.3):
```

```
        super(LSTM, self).__init__()
        self.hidden_size = hidden_size
        self.rnn_layers = rnn_layers
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True,␣
↪num_layers=rnn_layers)  # LSTM layer
        self.fc = nn.Linear(hidden_size, output_size)
        self.activation = nn.ReLU()
        self.dropout = nn.Dropout(p=dropout_p)

    # Forward method
    def forward(self, input_data, hidden):
        out, hidden = self.lstm(input_data, hidden)
        out = self.dropout(out)
        out = self.fc(out)
        # out = self.activation(out)
        return out, hidden

    # Hidden and cell state initialisation
    def init_hidden(self, batch_size):

        return (torch.zeros(self.rnn_layers, batch_size, self.hidden_size),
                torch.zeros(self.rnn_layers, batch_size, self.hidden_size))
```

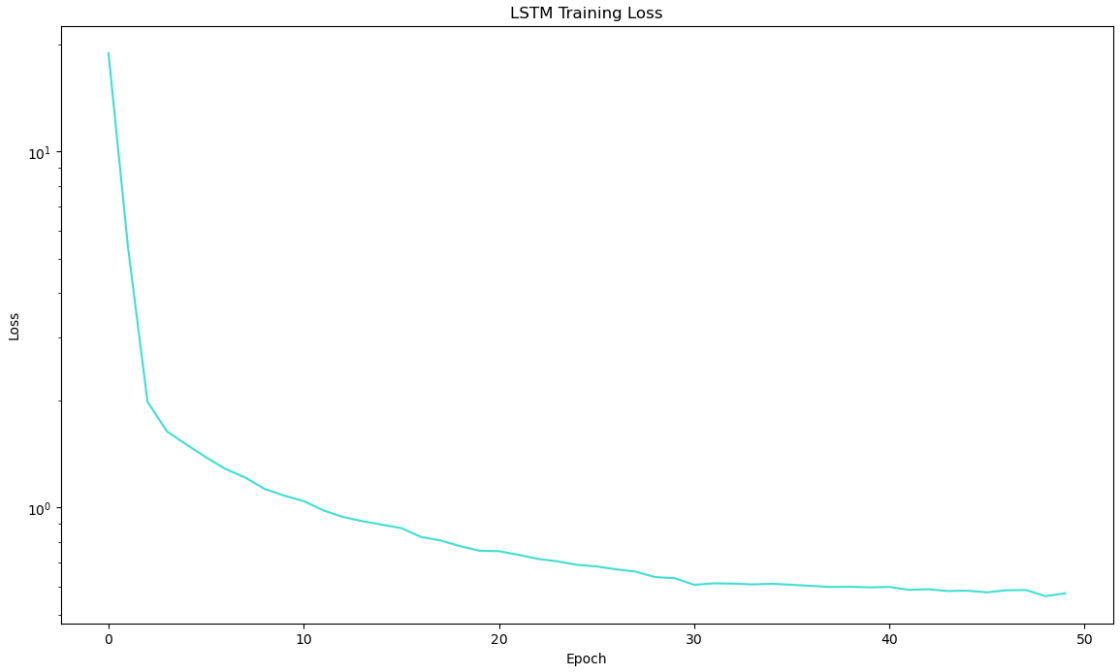
[1133]:
```
# Initialise the LSTM model
lstm_model = LSTM(input_size=input_size, hidden_size=hidden_size,␣
↪output_size=output_size, rnn_layers=rnn_layers)

# Use the same hyperparameters as the RNN
lstm_optimizer = optim.Adam(lstm_model.parameters(), lr=learning_rate)
lstm_scheduler = torch.optim.lr_scheduler.StepLR(lstm_optimizer,␣
↪step_size=step_size, gamma=gamma)

# Train LSTM
lstm_losses = train_rnn(lstm_model, input_data_train, output_data_train,
                        n_epochs=n_epochs, learning_rate=learning_rate,
                        criterion=criterion, optimizer=lstm_optimizer,
                        scheduler=lstm_scheduler, lag_weight=lag_weight)
```

```
Epoch 1/50, Loss: 18.8997, AGN Before: 0.3556, AGN After: 0.3556
Epoch 2/50, Loss: 5.3927, AGN Before: 0.4113, AGN After: 0.3796
Epoch 3/50, Loss: 1.9808, AGN Before: 0.2036, AGN After: 0.2036
Epoch 4/50, Loss: 1.6342, AGN Before: 0.1369, AGN After: 0.1369
Epoch 5/50, Loss: 1.5019, AGN Before: 0.1255, AGN After: 0.1255
Epoch 6/50, Loss: 1.3827, AGN Before: 0.1200, AGN After: 0.1200
Epoch 7/50, Loss: 1.2822, AGN Before: 0.1097, AGN After: 0.1097
Epoch 8/50, Loss: 1.2139, AGN Before: 0.1051, AGN After: 0.1051
Epoch 9/50, Loss: 1.1270, AGN Before: 0.0941, AGN After: 0.0941
```

```
Epoch 10/50, Loss: 1.0798, AGN Before: 0.1050, AGN After: 0.1050
Epoch 11/50, Loss: 1.0421, AGN Before: 0.0994, AGN After: 0.0994
Epoch 12/50, Loss: 0.9816, AGN Before: 0.0929, AGN After: 0.0929
Epoch 13/50, Loss: 0.9408, AGN Before: 0.0977, AGN After: 0.0977
Epoch 14/50, Loss: 0.9154, AGN Before: 0.1042, AGN After: 0.1042
Epoch 15/50, Loss: 0.8949, AGN Before: 0.1046, AGN After: 0.1046
Epoch 16/50, Loss: 0.8750, AGN Before: 0.0945, AGN After: 0.0945
Epoch 17/50, Loss: 0.8268, AGN Before: 0.0862, AGN After: 0.0862
Epoch 18/50, Loss: 0.8079, AGN Before: 0.0914, AGN After: 0.0914
Epoch 19/50, Loss: 0.7791, AGN Before: 0.0880, AGN After: 0.0880
Epoch 20/50, Loss: 0.7560, AGN Before: 0.0881, AGN After: 0.0881
Epoch 21/50, Loss: 0.7537, AGN Before: 0.0952, AGN After: 0.0952
Epoch 22/50, Loss: 0.7362, AGN Before: 0.0921, AGN After: 0.0921
Epoch 23/50, Loss: 0.7170, AGN Before: 0.0891, AGN After: 0.0891
Epoch 24/50, Loss: 0.7058, AGN Before: 0.0833, AGN After: 0.0833
Epoch 25/50, Loss: 0.6901, AGN Before: 0.0917, AGN After: 0.0917
Epoch 26/50, Loss: 0.6828, AGN Before: 0.0891, AGN After: 0.0891
Epoch 27/50, Loss: 0.6705, AGN Before: 0.0805, AGN After: 0.0805
Epoch 28/50, Loss: 0.6603, AGN Before: 0.0766, AGN After: 0.0766
Epoch 29/50, Loss: 0.6375, AGN Before: 0.0717, AGN After: 0.0717
Epoch 30/50, Loss: 0.6330, AGN Before: 0.0716, AGN After: 0.0716
Epoch 31/50, Loss: 0.6064, AGN Before: 0.0495, AGN After: 0.0495
Epoch 32/50, Loss: 0.6122, AGN Before: 0.0493, AGN After: 0.0493
Epoch 33/50, Loss: 0.6112, AGN Before: 0.0569, AGN After: 0.0569
Epoch 34/50, Loss: 0.6079, AGN Before: 0.0520, AGN After: 0.0520
Epoch 35/50, Loss: 0.6106, AGN Before: 0.0591, AGN After: 0.0591
Epoch 36/50, Loss: 0.6065, AGN Before: 0.0507, AGN After: 0.0507
Epoch 37/50, Loss: 0.6022, AGN Before: 0.0546, AGN After: 0.0546
Epoch 38/50, Loss: 0.5980, AGN Before: 0.0473, AGN After: 0.0473
Epoch 39/50, Loss: 0.5987, AGN Before: 0.0509, AGN After: 0.0509
Epoch 40/50, Loss: 0.5964, AGN Before: 0.0553, AGN After: 0.0553
Epoch 41/50, Loss: 0.5981, AGN Before: 0.0525, AGN After: 0.0525
Epoch 42/50, Loss: 0.5871, AGN Before: 0.0508, AGN After: 0.0508
Epoch 43/50, Loss: 0.5893, AGN Before: 0.0550, AGN After: 0.0550
Epoch 44/50, Loss: 0.5826, AGN Before: 0.0524, AGN After: 0.0524
Epoch 45/50, Loss: 0.5838, AGN Before: 0.0573, AGN After: 0.0573
Epoch 46/50, Loss: 0.5777, AGN Before: 0.0520, AGN After: 0.0520
Epoch 47/50, Loss: 0.5854, AGN Before: 0.0570, AGN After: 0.0570
Epoch 48/50, Loss: 0.5863, AGN Before: 0.0545, AGN After: 0.0545
Epoch 49/50, Loss: 0.5635, AGN Before: 0.0514, AGN After: 0.0514
Epoch 50/50, Loss: 0.5739, AGN Before: 0.0548, AGN After: 0.0548
```

[1145]:
```
#Visualisation of Loss against Epochs - LSTM
plt.figure(figsize=(14, 8))
plt.axes().set_yscale('log')
plt.plot(range(n_epochs),lstm_losses, linestyle='-',color = 'turquoise')
plt.xlabel("Epoch")
```

```
plt.ylabel("Loss")
plt.title("LSTM Training Loss")
plt.show()
```



[1137]:
```
# Evaluate LSTM performance on the test set
lstm_all_predictions = []
lstm_mse_list = []

with torch.no_grad():
    # Loop for data set
    for i in range(input_data_test.shape[0]):
        test_sequence = input_data_test[i]
        test_sequence = scaling * test_sequence.clone().detach()
        actual_data = output_data_test[i]

        # Warmup phase
        warmup_input = test_sequence[:n_warmup].unsqueeze(0)

        # Hidden and cell state initialisation
        hidden = lstm_model.init_hidden(batch_size=1)

        # Warm-up phase
        next_state, hidden = lstm_model(warmup_input, hidden)
        input = next_state[:, -1, :].unsqueeze(0)
```

```
        # Predict remaining time steps
        lstm_timeseries = []
        for i in range(len(actual_data) - n_warmup - 1):
            u_feature = test_sequence[n_warmup + i, 1].view(1, 1, 1)
            input = alpha * torch.cat((test_sequence[n_warmup + i, 0].view(1, 1,␣
↪1), u_feature), dim=-1) + (1 - alpha) * input
            input, hidden = lstm_model(input, hidden)
            lstm_timeseries.append(input.squeeze(0).numpy())

        lstm_timeseries = np.array(lstm_timeseries)
        lstm_all_predictions.append(lstm_timeseries)

        # MSE
        mse = np.mean((actual_data[n_warmup:-1].numpy() - lstm_timeseries[:, 0])␣
↪** 2)
        lstm_mse_list.append(mse)

# Average MSE for LSTM
lstm_average_mse = np.mean(lstm_mse_list)
```

1.6.7 4.7 Performance Validation

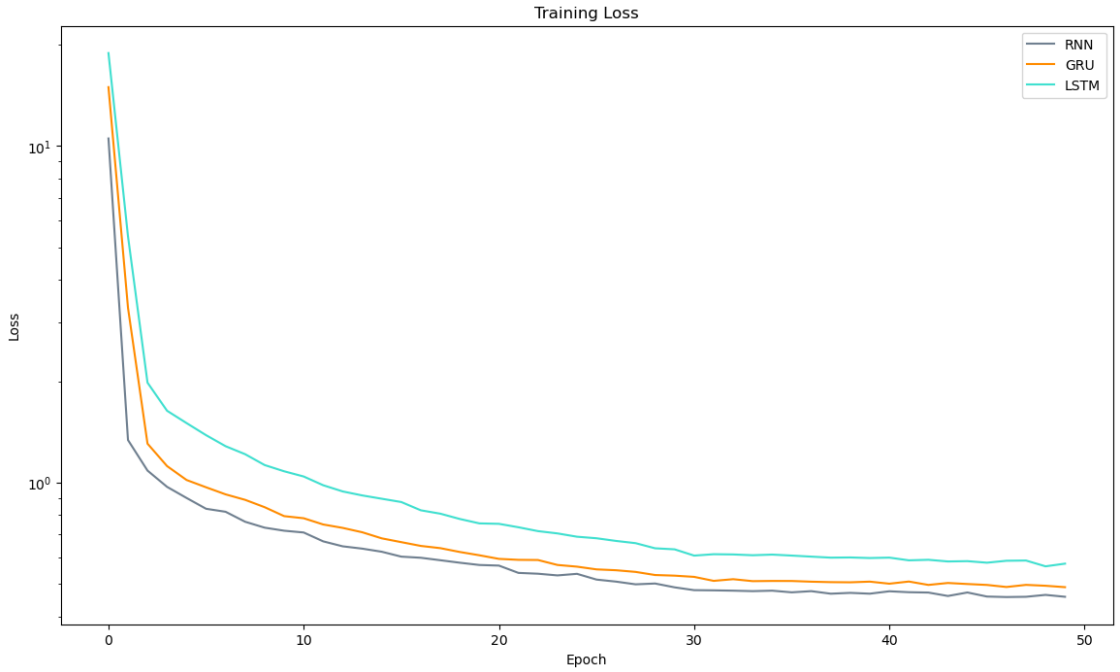
Comparison of RNN Architectures

The section below comprehensively compares the performance of the three RNN architectures by their testing prediction MSE and convergence plots.

[324]:
```
print(f"RNN Average MSE on Test Set: {average_mse:.6f}")
print(f"GRU Average MSE on Test Set: {gru_average_mse:.6f}")
print(f"LSTM Average MSE on Test Set: {lstm_average_mse:.6f}")
```

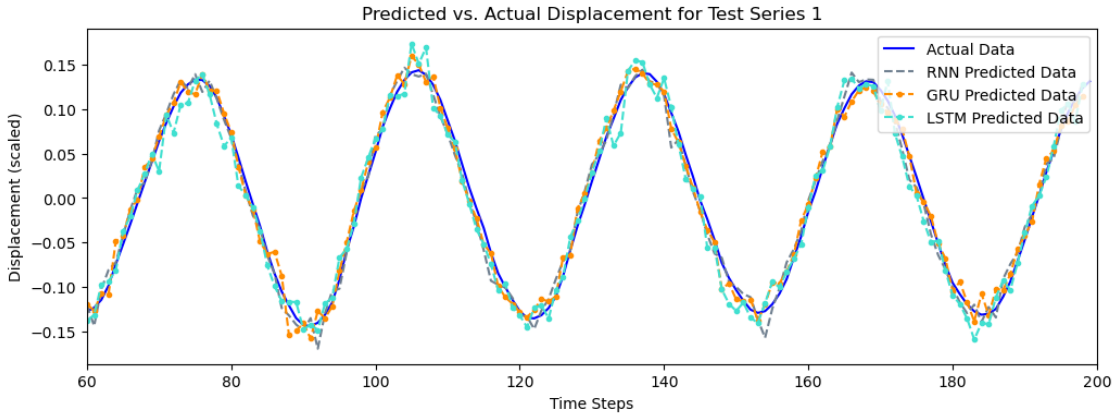
```
RNN Average MSE on Test Set: 0.001019
GRU Average MSE on Test Set: 0.000920
LSTM Average MSE on Test Set: 0.001274
```

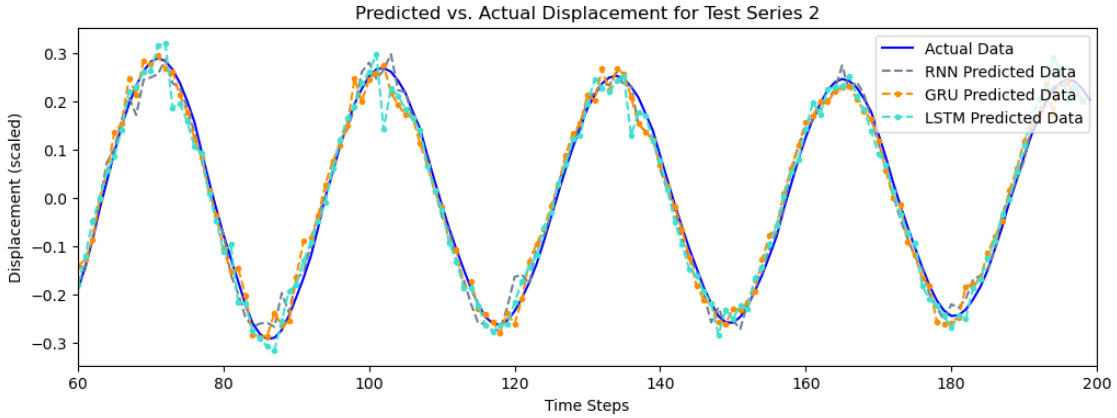
[1239]:
```
#Visualisation of Loss against Epochs
plt.figure(figsize=(14, 8))
plt.axes().set_yscale('log')
plt.plot(range(n_epochs),losses, linestyle='-',color ='slategrey',label = "RNN")
plt.plot(range(n_epochs),gru_losses, linestyle='-',color ='darkorange',label =␣
↪"GRU")
plt.plot(range(n_epochs),lstm_losses, linestyle='-',color = 'turquoise',label␣
↪="LSTM")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.title("Training Loss")
plt.show()
```

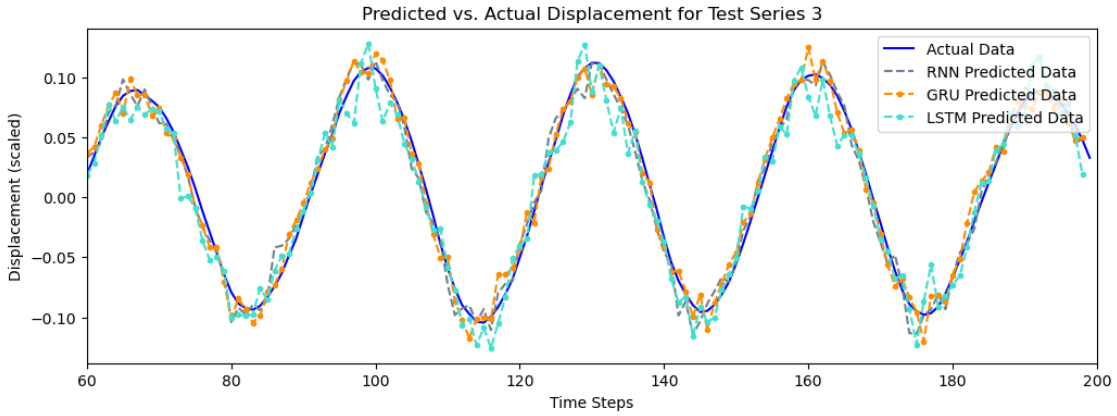


[1261]:
```
# Visualise prediction test for RNN architectures
for i in i_data:
    plt.figure(figsize=(12, 4))
    t = np.arange(len(output_data_test[i]))

    # Plot actual data
    plt.plot(t, output_data_test[i].numpy(), label="Actual Data", color="blue",␣
↪linestyle="-")

    # Plot RNN predictions
    plt.plot(t[n_warmup:n_warmup + len(all_predictions[i])], all_predictions[i][:
↪, 0] / scaling,
             label="RNN Predicted Data", linestyle="--", color='slategrey')

    # Plot GRU predictions
    plt.plot(t[n_warmup:n_warmup + len(gru_all_predictions[i])],␣
↪gru_all_predictions[i][:, 0] / scaling,
             label="GRU Predicted Data", linestyle="--", marker=".",␣
↪color='darkorange')

    # Plot LSTM predictions
    plt.plot(t[n_warmup:n_warmup + len(lstm_all_predictions[i])],␣
↪lstm_all_predictions[i][:, 0] / scaling,
             label="LSTM Predicted Data", linestyle="--", marker=".",␣
↪color='turquoise')
```

```
    plt.xlabel("Time Steps")
    plt.ylabel("Displacement (scaled)")
    plt.legend(loc="upper right")
    plt.xlim(n_warmup, len(actual_data))
    plt.title(f"Predicted vs. Actual Displacement for Test Series {i+1}")
    plt.show()
```







The average MSE scores for predictions show that the GRU (0.000920) slightly outperforms both the RNN (0.001019) and the LSTM (0.001274), though the differences are marginal.

For the convergence plot, Aal models exhibit a steep decline in loss during the early epochs and stablise as training progresses as the expected behaviour. The RNN stabilises at the lowest loss, followed by the GRU, and then the LSTM. This is unexpected behaviour, as LSTM and GRU architectures are intended to perform better while trading an increase in computational overhead. A possible implication can be due to the short sequence of the truncated data, which can negatively counteract GRU and LSTM's intended purpose for mitigating gradients and memory retention over long-term sequences leading to overfitting; while the tradtiional RNN maintains viable in capturing short-term dynamics. Further study with the untruncated data should be performed to validate this.

While the LSTM captures the same overall trends in predictions, the higher variability and phase misalignment in the predictions suggest the complex gating mechanisms are less appropriate for generalising short sequences.

1.7 5. Neural Ordinary Differential Equations (NODEs)

Not completed

1.8 References

[1] Szalai, RS. (n.d.). University of Bristol SEMTM0007. © 1997-2024 Blackboard Inc. All Rights Reserved. US Patent No. 7,493,396 and 7,558,853. Additional Patents Pending. https://www.ole.bris.ac.uk/ultra/courses/_259188_1/
[2] Barton, D. A. (2016). Control-based continuation: Bifurcation and stability analysis for physical experiments. Mechanical Systems and Signal Processing, 84, 54–64. https://doi.org/10.1016/j.ymssp.2015.12.039
[3] Chrysafides, A., & Blanchard, P. A. (2002). On the Influence of Material Hardening on the Fatigue Life of Springs. Journal of Engineering Materials and Technology, 124(2), 226-231
[4] Yao, X., & Liu, Y. (2004). Modeling Nonlinear Time Series with Artificial Neural Networks: A Comparison of Different Architectures. Neurocomputing, 57(1-4), 213-228
[5] Jones, R. L., & McDonald, M. (2000). Nonlinear Vibrations in Mechanical Systems: Harmonics

and Bifurcations. Journal of Sound and Vibration, 234(3), 539-558
[6] Yang, Y., & Liu, Y. (2018). A Comprehensive Comparison of Normalization Methods in Machine Learning Applications. Proceedings of the 2018 International Conference on Data Science and Machine Learning
[7] Jaeger, H. (2001). The "Echo State" Approach to Analysing and Training Recurrent Neural Networks. GMD Report 148, German National Research Center for Information Technology
[8] Jaeger, H. (2002). Adaptive Nonlinear System Identification with Echo State Networks. Proceedings of the International Workshop on Self-Organization and Autonomous Machines, 71-74
[9] Lukoševičius, M., & Jaeger, H. (2009). Reservoir Computing Approaches to Recurrent Neural Network Training. Computer Science Review, 3(3), 127-149
[10] Sklansky, J. (1989). Cluster Validation Using Cross-Validation. Pattern Recognition Letters, 9(1), 35-39
[11] Buonomo, I., & Manfredi, L. (2011). Echo State Networks: A Comparative Study on Reservoir Hyperparameters and Nonlinearities. Neurocomputing, 74(11), 1845-1853
[12] Bergstra, J., & Bengio, Y. (2012). Random Search for Hyper-Parameter Optimization. Journal of Machine Learning Research, 13(1), 281-305
[13] Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical Bayesian Optimization of Machine Learning Algorithms. Advances in Neural Information Processing Systems (NeurIPS), 25, 2951-2959
[15] Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. Neural Computation, 9(8), 1735-1780
[16] Kingma, D. P., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. International Conference on Learning Representations (ICLR)
[17] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning Representations by Back-propagating Errors. Nature, 323(6088), 533-536
[18] Williams, R. J., & Zipser, D. (1989). A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. Neural Computation, 1(2), 270-280
[19] Cho, K., Merrienboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. Empirical Methods in Natural Language Processing (EMNLP), 1724-1734