UNIVERSITY OF CAMBRIDGE

Jack Parkinson

An exploration of removing wind noise from mono speech input using DSP and AI approaches

Computer Science Tripos – Part II Project
Dissertation

Jack Parkinson
jrp80@cam.ac.uk

Fitzwilliam College
May 14, 2021

Declaration of Originality

I, Jack Parkinson of Fitzwilliam College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. I am content for my dissertation to be made available to the students and staff of the University.

Name: Jack Parkinson

Date: May 14, 2021

Proforma

Candidate Number: 2429A
Project Title: An exploration of removing wind noise from mono speech input using DSP and AI approaches
Examination and Year: Computer Science Tripos – Part II, July 2021
Word count: 11927
Line count of code: 8918 (excluding blank lines and libraries, including comments)
Project Originator: Self-originated
Project Supervisors: Dr Dong Ma and Dr Jing Han

Original Aims of the Project:

This project aimed to explore existing solutions to remove wind noise from a mono input source, by implementing two solutions from research papers, and testing these with a combination of wind and speech noise. Parameters of each algorithm should then be adjusted to see how different input properties and hyperparameters affect performance. Evaluation also played a key part in this project, with a combination of evaluation methods – including human MOS testing, speech recognition accuracy using WER, SSNR, MSE, as well as other sample-wise comparisons – being used to compare each algorithm, both qualitatively and quantitatively. As an extension, I should aim to implement my own wind noise removal algorithm.

Work Completed:

In this project I:

- analysed a variety of wind noise data from multiple sources, in both the time and frequency domains, and created a general model for wind noise.
- created an artificial wind noise dataset incorporating collected real wind noise and speech from the LibriSpeech dataset.
- implemented two wind noise removal algorithms - one utilising a strictly-DSP approach [1], and other using a combination of DSP and AI methods [2].
- implemented my own improvements to these algorithms, seeing notable improvements in performance.
- thoroughly analysed and evaluated each algorithm presented, both qualitatively and quantitatively.
- determined and evaluated the effect of hyperparameters on algorithm output.
- created my own algorithm for wind noise removal, resulting in an SOTA algorithm with performance exceeding the original algorithms presented.

Special Difficulties Faced:

Due to the Coronavirus pandemic, MOS evaluation was completed online instead of in person. This should not affect the tests, since my testing plan has been adapted to ensure that the results collected are still reproducible and take into consideration additional variables (including quality of headphones/ internet signal). I also suffered from poor health towards the end of my project, which may have had some impact on the quality of the write-up.

Acknowledgements

I would like to express my gratitude to several people, who contributed their thoughts and ideas, and offered support throughout this project and in my wider university degree.

- **Dr Dong Ma** and **Dr Jing Han**, for offering to supervise me on my self-originated project, and for offering their insightful advice and guidance throughout the project.
- **My Director of Studies**, and **my Tutor**, without whose reassurance and support I would have not made it through my degree with the same level of success and confidence.
- **All participants of my MOS testing**, who offered their time to participate and offer detailed feedback on my algorithms.
- To **my friends** and **family**, in particular **my fellow college CompScis** and **my parents**, whom have selflessly cared, and offered countless time and support to help me through my degree and wider university life.

Contents

0 Proforma **i**
0.1 Declaration of Originality . . . i
0.2 Proforma . . . ii
0.3 Acknowledgements . . . iii

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Existing Solutions and Research . . . 1
1.3 Goals and Contributions . . . 2

2 Preparation **3**
2.1 Starting Point . . . 3
2.2 Requirements Analysis . . . 3
2.2.1 Development Strategy . . . 3
2.2.2 Software Engineering Strategy . . . 3
2.2.3 Choice of Language, Libraries and Tools . . . 4
2.3 Theory . . . 4
2.3.1 Digital Signal Processing . . . 4
2.3.2 Autoencoders and Principal Component Analysis . . . 5
2.3.3 Wind Noise Theory . . . 6

3 Implementation **7**
3.1 Repository overview . . . 7
3.2 Data Collection and Dataset Creation . . . 8
3.2.1 Algorithmic Bike Wind Noise Generation . . . 8
3.3 Analysis of Wind Noise . . . 9
3.3.1 Spectrograms of Wind Noise . . . 9
3.3.2 Spectrogram of Speech Audio . . . 10
3.3.3 Volume Statistics of Wind Noise and Speech . . . 10
3.3.4 Extraction of Wind Gusts from Wind Noise . . . 10
3.3.5 Properties of Wind Gusts in the Time Domain . . . 11
3.3.6 Power Spectrum Density of Wind Noise and Speech . . . 12
3.4 Original Nelke-Vary Algorithm . . . 12
3.4.1 Segmentation, Windowing, and FFT . . . 13
3.4.2 Feature Extraction and Wind Detection . . . 13
3.4.3 Noise PSD Estimation . . . 13
3.4.4 Spectral Weighting . . . 14
3.4.5 Inverse Fast Fourier Transform (IFFT) and Overlap-Add . . . 15
3.5 Hyperparameters of the Nelke-Vary Algorithm . . . 15
3.6 Adaptations to the Nelke-Vary Algorithm . . . 16
3.6.1 Heuristic Improvements . . . 16
3.6.2 Click Removal . . . 17
3.7 Original Lee-Theunissen Algorithm . . . 18
3.7.1 Click Removal, Segmentation and Windowing . . . 18
3.7.2 Bandpass Filtering . . . 18
3.7.3 Amplitude Envelope and Normalisation . . . 18

3.7.4 PCA . . . 19
3.7.5 Autoencoder . . . 20
3.7.6 Signal Reconstruction . . . 21
3.8 Hyperparameters of the Lee-Theunissen Algorithm . . . 21
3.9 Adaptations to the Lee-Theunissen Algorithm . . . 22
3.9.1 Adjustments to the Neural Network Architecture . . . 22
3.10 My Solution . . . 23
3.10.1 Input Signal Preprocessing . . . 24
3.10.2 Phase and Amplitude Data . . . 24
3.10.3 NN Autoencoder and Gains Prediction . . . 25
3.10.4 Signal Reconstruction . . . 25
3.11 Adaptations to My Solution . . . 26
3.11.1 Sine Activation Function . . . 26

4 Evaluation **27**
4.1 Metrics . . . 27
4.2 Subjective Comparison of Each Algorithm . . . 27
4.2.1 Subjective Analysis of Noisy Speech Signals . . . 28
4.2.2 Subjective Analysis of Denoised Signals . . . 28
4.2.3 Subjective Analysis of Artefacts . . . 29
4.2.4 Subjective Analysis of Denoised Voice Coherency . . . 30
4.3 Objective Comparison of Each Algorithm . . . 31
4.3.1 Comparison of Algorithms using a Variety of Metrics . . . 31
4.3.2 Comparing Algorithms by Word Error Rate . . . 32
4.3.3 Comparison of Algorithms using Spectrogram Analysis . . . 33
4.4 Analysing the Effect of Hyperparameter Values and NN Architectures on Each Algorithm . . . 35
4.4.1 Optimising Hyperparameters . . . 35
4.4.2 Analysis of the Hyperparameters of the Original Nelke-Vary Algorithm . 35
4.4.3 Analysis of the Hyperparameters of the Original Lee-Theunissen Algorithm 37

5 Conclusions **40**
5.1 Achievements and Progress . . . 40
5.2 Challenges Faced and Lessons Learned . . . 40
5.3 Future Work . . . 40

6 Bibliography **40**

A Additional Theory **46**
A.1 Digital Signal Processing . . . 46
A.1.1 Fourier Transform . . . 46
A.1.2 Spectral Weighting and Filter Banks . . . 46
A.2 Artificial Intelligence . . . 47
A.2.1 Neural Networks . . . 47
A.2.2 Supervised Learning . . . 47
A.2.3 Loss Functions . . . 47

B Additional Analysis of Wind Noise **48**
B.1 Analysis of the Amplitude of Wind Noise and Speech . . . 48
B.2 Fitting the CDF of Stationary Wind Gust Durations . . . 48

C Failed Improvements to the Nelke-Vary Algorithm **50**
C.1 Attempts to Remove Wind Peaks from the PSD . . . 50

D Additional Evaluation Results **51**
D.1 Human Testing MOS Scores . . . 51
D.2 Individual Analysis of Each Algorithm using a Variety of Metrics . . . 52
D.3 Nelke-Vary Additional Hyperparameter Analysis . . . 54
D.3.1 Exploration of Hyperparameter Values for `f2` . . . 54
D.3.2 Exploration of Hyperparameter Values for SSC_{max} . . . 54
D.3.3 Exploration of Hyperparameter Values for PSD Sampling Locations . . . 55
D.4 Lee-Theunissen Additional Hyperparameter and NN Architecture Analysis . . . 56
D.4.1 Exploration of Hyperparameter Values for `numberFilters` . . . 56
D.4.2 Exploration of the Number of PCA Components for the 'Gains PCA' NN Architecture . . . 56
D.4.3 Exploration of the Number of PCA Components for the 'No PCA' NN Architecture . . . 56

E Consent Form **58**

F Ethics Form **61**

G Glossary **66**

H List of Figures **66**

I List of Tables **68**

J List of Algorithms **70**

K Project Proposal **71**

Chapter 1

Introduction

1.1 Motivation

When a microphone is used to record in windy or turbulent conditions, air vortices (**turbules**) form on the microphone's surface, *amplifying the sound of the wind itself* [3, 4, 5, 6] and producing wind noise artefacts whose frequency and volume varies with wind speed and relative microphone-wind orientation. I highlight active noise-cancelling headphones, which use multiple microphones to sample background noise, which is then subtracted from outputted audio to attenuate noise [7, 8]. Here, *distinct turbules* form on each microphone, producing *distinct spectra* which are *amplified through noise-cancellation* – hence **using a wind noise-specific removal algorithm would greatly improve noise-cancellation performance**.

Wind noise also affects speech coherency for humans/computers since *wind is louder and occupies similar frequencies to speech* (Section 2.3.3, Figure 3.6). Humans learn through evolution/time how to denoise auditory inputs. Therefore, humans are more sensitive to the relatively modern unnatural phenomena of microphone-wind interference and cannot eliminate them effectively [9]. Finally, the artefacts created from wind noise removal are particularly intrusive to human hearing, even more so than wind [9]. Thus, I need to design denoising algorithms that minimise artefacts whilst maximising denoising performance.

Wind noise removal is pertinent in aviation, where increased turbulence at lower altitudes (Figure 2.5) [10] can greatly affect cockpit communication and alarm volumes [11]; and in speech-recognition applications such as automatic audio-descriptive captions [12]. In each case, selective wind noise removal would improve performance whilst preserving other background noises (e.g. alarms, traffic) which offer safety benefits and would otherwise be attenuated by traditional noise-removal algorithms.

1.2 Existing Solutions and Research

A naïve method of removing wind noise is to cover the microphone surface with wind-cancelling foam, preventing surface vortices from forming [13] – this, however, is unsightly, attenuates input signals, varies with microphone position, and *does not attenuate all wind noise* [12]. Most current software options focus on removing all noise and have achieved appealing performance – Haque and Bhattacharyya proposed utilising Kalman filters to remove background noise from speech, offering mean squared errors at $\approx 3.7\%$ of the input power spectrum density (**PSD**) [14]; whilst krisp [15] used a Deep Neural Network (**DNN**) on algorithmically-generated noisy speech to produce gains that attenuate noise, seeing a 1.4 point increase in Mean Opinion Score (**MOS**) score over noisy speech input (Figure 1.1) [16].

Most wind-specific noise removal algorithms have utilised a multiple-microphone setup to distinguish noise from the pure signal more accurately. Nelke and Vary, for example, exploited the variance in the phase of coherence within wind portions of a dual-microphone signal to detect and remove wind noise [17]; and Rhodes using a combined Digital Signal Processing (**DSP**) and Artificial Intelligence (**AI**) approach to remove wind noise in low power wearables, achieving a 50% improvement in Word Error Rate (**WER**) [18].

However, ***limited research has been focused on single-microphone wind noise removal***. As an example, Nemer and Leblanc used a DSP approach, utilising multiple algorithms selectively using the perceived wind noise resonance [19].

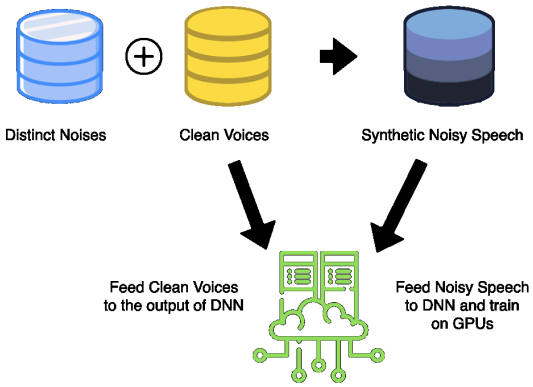


Figure 1.1: The data collection and training pipeline of the 2hz.ai noise removal algorithm [16].

1.3 Goals and Contributions

My project, therefore, focuses on removing single-microphone wind noise from a mono speech input. This involved (with extension tasks starred):

- Analysing speech and wind signals in the *time and frequency domains*; and using this to **model wind noise**.
- $\bullet^\star$ **Producing an algorithm to generate wind noise algorithmically** from wind noise clips; and combining this with speech (from the LibriSpeech dataset [20, 21]) to generate a combined speech-wind dataset.
- Implementing the DSP-based wind noise removal algorithm proposed by **Nelke and Vary**[1], utilising *vectorised libraries* (SciPy [22]) to improve performance. This algorithm approximates underlying wind noise by fitting an exponential curve to the PSD of the input signal, which is used to approximate the optimal frequency-domain denoising gains.
- Implementing the AI-based wind noise removal algorithm proposed by **Lee and Theunissen**[2], utilising vectorisation and AI libraries (Tensorflow [23]). This algorithm applies a time-domain bandpass filter bank to the segmented input signal to produce a time-domain spectrogram; then Principal Component Analysis (**PCA**) and a neural network (**NN**) autoencoder (Section 2.3.2) is used to reproduce the time-domain denoising gains.
- $\bullet^\star$ *Improving both algorithms* by making changes to both hyperparameters and the algorithms themselves, **with noticeable performance improvements**.
- $\bullet^\star$ ***Creating my own novel wind noise removal algorithm*** (utilising a NN operating on frequency-domain data to estimate gains from frame PSDs) ***that offers SOTA performance***, and improves on the other presented algorithms.
- $\bullet^\star$ Writing my solutions (where possible) to run in *real-time*, using vectorised operations and further optimisations to improve performance.
- Adjusting and evaluating algorithm hyperparameters to determine their performance impact.
- Evaluating the performance of these algorithms, using a variety of *quantitative* and *qualitative* metrics.

Chapter 2

Preparation

2.1 Starting Point

I had a rudimentary understanding of DSP and AI practices from the Part IB Group Project. I also completed the Digital Signal Processing course in Michaelmas and learnt Python and NumPy during IA Scientific Computing. My knowledge of neural networks was minimal before this project commenced; hence I learned Tensorflow and Keras from scratch.

2.2 Requirements Analysis

2.2.1 Development Strategy

The complexity of my project will likely arise from its *scope and domain-specific knowledge*. As a result, I will *modularly structure* my code to enable solutions to be built iteratively. This will enable separate mock functions to mimic un-implemented functionality during development and organises my code for easy extensibility.

All algorithms presented will be coded using a single *main* function, which can be called with many parameter values (enumerated in Section 3.5 and Section 3.8). This aids in testing and while improving each algorithm's performance (through experimenting with parameter values).

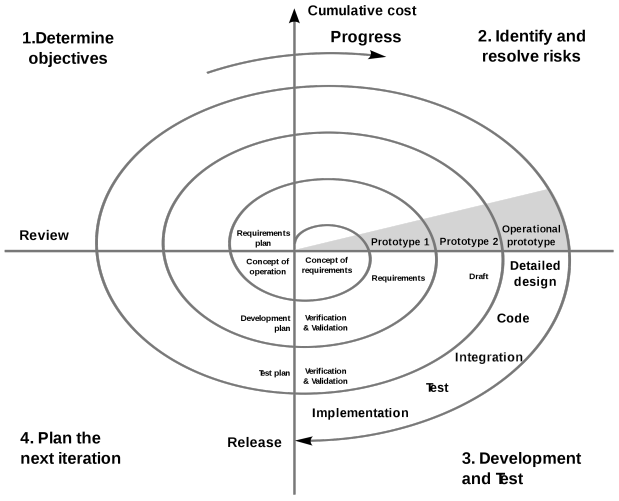


Figure 2.1: The spiral model of software development, used in my project [24].

2.2.2 Software Engineering Strategy

I will make heavy use of the **spiral model** (Figure 2.1), a popular development model which "break[s] a project into smaller [well-defined] segments" [25], enabling planning, risk analysis, development and evaluation stages to be performed more easily. The spiral model is *risk-tolerant*, enabling progress to be evaluated at regular intervals; *minimises contact* with the 'client' (supervisors); and can *adapt quickly to issues*, which makes it more suitable for a research-centric project than (for example) the waterfall model [25].

2.2.3 Choice of Language, Libraries and Tools

Vectorised operations, such as bandpassing filters and convolutions, will be handled by NumPy and SciPy, with Matplotlib used to plot graphs. For neural network development, I will use Tensorflow rather than PyTorch since it features better documentation and is easier to deploy once trained [26].

I will use Perceptual Evaluation of Speech Quality (**PESQ**) [27] and Short-Term Objective Intelligibility (**STOI**) [28] for evaluation – I will utilise Python libraries for both (python-pesq and pystoi) to ensure compatibility with other papers' results. I will also use Google Forms to question participants on the perceived performance of my algorithms.

All libraries used are distributed under free permissive open-source licenses, with SciPy distributed under a BSD license [29], Tensorflow distributed under an Apache license [30], and python-pesq/pystoi using a MIT license [31]. I note that PESQ is proprietary and usually requires a paid license [32]. However, *a specific exception exists for research purposes.*

I will utilise a combination of Git, GitHub, Google Drive and hard drives for version control/backup. I have also written two batch scripts that enable one-click commits and pulls from GitHub, enabling me to work on multiple computers sequentially. I will also use PyCharm Edu as an Integrated Development Environment (**IDE**), using code cleanup to maintain a consistent code style.

2.3 Theory

2.3.1 Digital Signal Processing

Windowing Functions

Windowing functions are functions applied to a signal whose value outside a given interval is zero-valued. A finite signal is equivalent to multiplying an infinite signal with a rectangular function in the time domain which, by the convolution theorem:

$$\mathcal{F}\{(f \cdot g)(t)\} = \mathcal{F}\{f(t)\} * \mathcal{F}\{g(t)\},$$

and $\mathcal{F}\{\text{rect}(t)\}(f) = \text{sinc}(f)$, is equivalent to convolution with a sinc function in the frequency domain, in turn causing *spectral leakage* [33, 34].

By windowing each signal with a Hann window, I *eliminate discontinuities* when Fourier-transformed. The Hann window, defined as

$$w_{\text{hann}}(x) \triangleq \begin{cases} \frac{\pi x}{L}, & \text{for } |x| \leq L/2 \\ 0, & \text{for } |x| > L/2 \end{cases},$$

is also an *'equal-voltage' cross-fade window* [35]. That is, for two signals windowed with 50% overlap, the voltage during the overlap remains constant after windowing/overlapping, as:

$$w_{\text{hann}}(x)+w_{\text{hann}}(x+\tfrac{L}{2}) = \cos^2\left(\tfrac{\pi x}{L}\right)+\cos^2\left(\frac{\pi(x+\frac{L}{2})}{L}\right) = \cos^2\left(\tfrac{\pi x}{L}\right)+\sin^2\left(\tfrac{\pi x}{L}\right) = 1 \text{ if } 0 \leq x \leq L/2.$$

Hilbert Transform

The Hilbert transform is a linear function which calculates the *analytic signal* $H(f(t))$ of an input signal $f(t)$. This operation is defined as a convolution in the time domain:

$$H(f(t)) = f(t) * \frac{1}{\pi t},$$

or a phase shift in the frequency domain:

$$H(F(t)) = \mathcal{F}^{-1}\{F(t) \cdot -i \cdot \text{sgn}(F(t))\}.$$

Here, $\text{sgn}(x)$ is the sign function of x [36].

By calculating the absolute value of the Hilbert transform, I obtain the *amplitude envelope*,

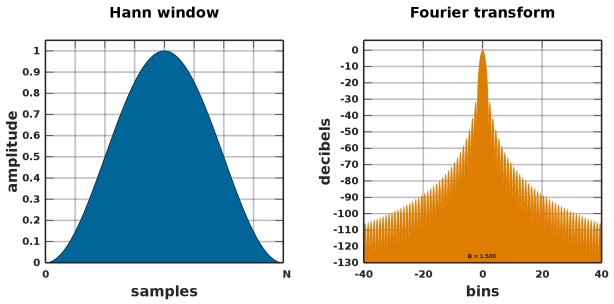


Figure 2.2: The Hann window, used in my solution both for windowing and as an equal-voltage cross-fade function [40]

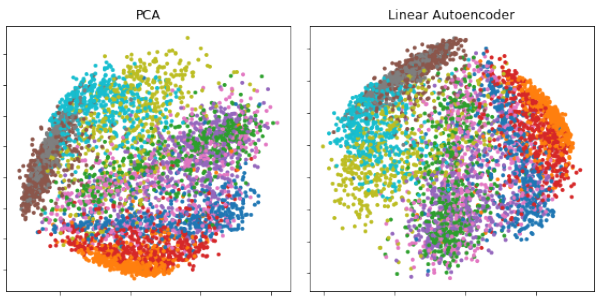


Figure 2.3: A comparison of an autoencoder with PCA for dimensionality reduction. Note both feature spaces are roughly the same [39]

a smooth, positive curve bounding the signal's amplitude that *approximates the signal's power at each point in time* [37].

2.3.2 Autoencoders and Principal Component Analysis

Autoencoders are neural networks trained to learn a *dimension-reduced encoding/decoding* ($\phi : \mathcal{X} \to \mathcal{F}$ and $\psi : \mathcal{F} \to \mathcal{X}$) from an input/output space $\mathcal{X}$ to a feature space $\mathcal{F}$, such that the *reconstruction error* $(I - (\phi \circ \psi)I)^2$ *is minimised* [38, 39].

Rather than reconstructing the input, I will train my autoencoders to produce gains that denoise the input signal. $\mathcal{F}$ then represents an *intermediate feature space* from which it can derive these gains effectively.

Conversely, principal component analysis (PCA) is a linear method that computes the *principal components* of a dataset. The first N principal components P_N (when ordered descending by eigenvalue) form an *orthogonal basis* that maximises variance between data points and *minimises mean-squared error upon reconstruction.* To compute PCA components, I first normalise the data, setting its mean to 0 and standard deviation to 1.

From here, the dataset's *covariance matrix* is computed, where the covariance of two values Cov(a,b) roughly denotes how a and b vary with each other. Given a dataset D length L with data elements D_i size M, I define the covariance matrix $\mathrm{Cov}(D)$[1] as:

$$\mathrm{Cov}_D = \begin{bmatrix} \mathrm{Cov}(D_1, D_1) & \mathrm{Cov}(D_1, D_2) & \dots & \mathrm{Cov}(D_1, D_{M-1}) & \mathrm{Cov}(D_1, D_M) \\ \mathrm{Cov}(D_2, D_1) & \ddots & \ddots & \ddots & \mathrm{Cov}(D_2, D_M) \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \mathrm{Cov}(D_{M-1}, D_1) & \ddots & \ddots & \ddots & \mathrm{Cov}(D_{M-1}, D_M) \\ \mathrm{Cov}(D_M, D_1) & \mathrm{Cov}(D_M, D_2) & \dots & \mathrm{Cov}(D_M, D_{M-1}) & \mathrm{Cov}(D_M, D_M) \end{bmatrix}$$

I then compute the *eigenvectors* V and eigenvalues λ of this covariance matrix to satisfy $\lambda V = \mathrm{Cov}_D V$. These eigenvectors are orthogonal, uncorrelated and store "the best possible (most accurate) representation of data using any specified finite number of terms" [41] through maximising the information stored by each successive eigenvector. By discarding eigenvectors with eigenvalues below a threshold, I can reduce the dimension of my data.

Computing PCA eigenvectors has a time complexity of $O(n^3)$, where n is the size of my dataset. However, projecting data into a reduced-dimension space defined by my eigenvectors, and reconstructing my data from this space, requires only a matrix multiplication with time complexity $O(nc)$ each, where c is the number of components used. A dense autoencoder has a time complexity of $O(n^2)$, but can represent *non-linear mappings* between the input/output and feature spaces.

[1] I note that the matrix is symmetric – $\mathrm{Cov}(a, b) = \mathrm{Cov}(b, a)$ – and further that the diagonal of the matrix has the property $\mathrm{Cov}(a, a) = \mathrm{Var}(a)$.

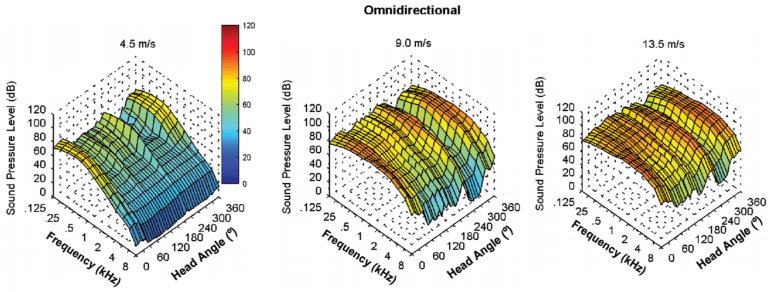


Figure 2.4: Wind noise spectrum of an omnidirectional microphone at three wind speeds [42]. Note that the spectrum is both asymmetrical and irregular in shape, owing both to differences in the geometry of the microphone at different angles and the unpredictable nature of turbules.

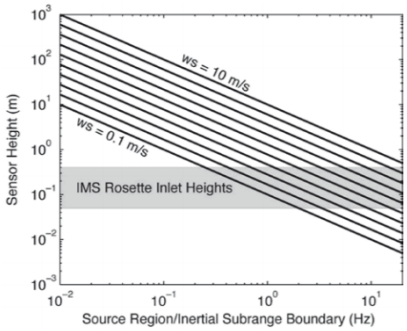


Figure 2.5: A graph showing the relationship between wind speed, height of the sensor and the source region-inertial subrange frequency boundary [10].

2.3.3 Wind Noise Theory

Panofsky and Dutton [43] define two forms of wind turbulence – **convective**, caused by thermal instability in the atmosphere, and **mechanical**, caused by wind interference with objects (the surface of the microphone [3, 5, 6]). *Taylor's hypothesis* then states that *turbules (turbulent vortices) can be treated as "spatially fixed, time-invariant anomalies"* [44, 10].

These two types of turbulence form three frequency ranges of wind noise acting at different"spatial scales of turbulence" [10], namely the "**source region** (large scales $\gg$ 10m and low frequencies $<$ 10Hz), **inertial subrange** (intermediate scales (10cm $-$ 10m) and frequencies $>$ 10Hz), and **dissipation region** (small scales $<$ 10cm and high frequencies)" (Figure 2.5) [10]. These subranges help us define the four primary types of wind noise:

- Firstly, **wind velocity** (i.e. pure convective turbulence) can directly (but negligibly [45]) affect the wind noise spectrum, with Monin and Yaglom [46] defining its PSD as:
$$V(k) = a\epsilon^{2/3}k^{-5/3}$$
in the inertial subrange. Here, $k = 2\pi f/u$ (by Taylor's hypothesis), ϵ and a are constants, f is the wind frequency (Hz), and u is the wind speed [10].
- Secondly, when wind vortices collide with objects, kinetic energy is converted to pressure energy (due to **turbulence-object interaction**) – fluctuations in the kinetic and pressure energy create wind noise in the source region and inertial subrange [47].
- Moreover, by Taylor's hypothesis, turbules (formed in turbulent flow in the absence of objects), when moving due to the local mean wind speed, create a third distinct type of wind noise through **turbulence-turbulence interaction**. This is sampled when the turbule is near the microphone [48] and mainly affects the source region/inertial subrange [10].
- Finally, when wind noise collides with obstacles at an oblique angle, a distinct form of wind noise, due to **turbulence-mean shear interaction**, is formed [47], whose frequency varies with the shear angle. This is the predominant form of wind noise in the inertial subrange but also affects the source region [10].

External factors affect wind noise significantly: increased temperatures (during the day) destabilise air more, producing both conductive and additional mechanical turbulence. In addition, coastal locations have stronger winds due to the different specific heat capacities of water and the ground producing temperature, hence pressure differentials [10].

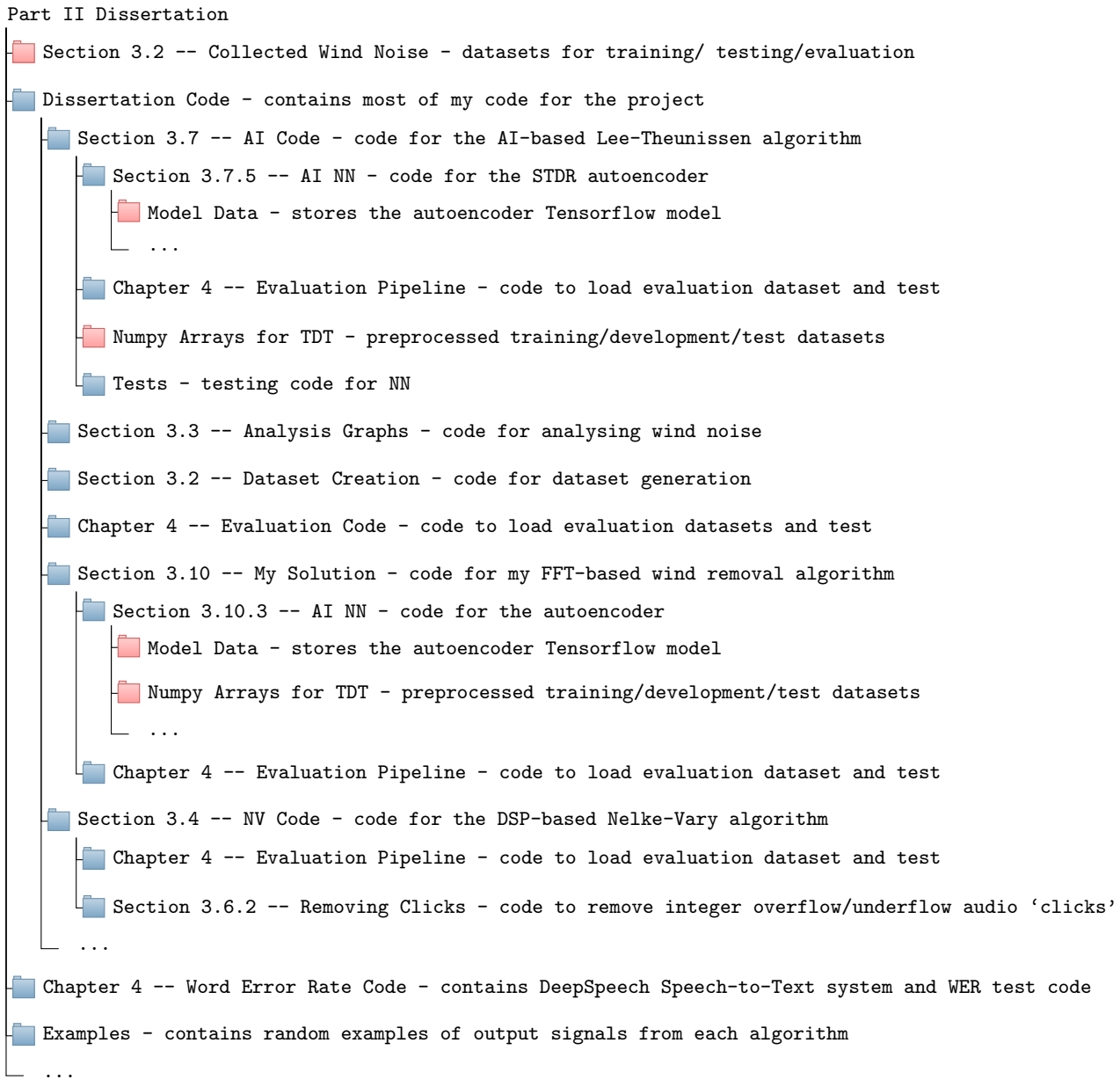
When combined, all wind noises roughly approximate the power spectrum density:
$$S_{xx}(x(f)) = b/f^a,$$
where a and b are changing constants [1] [49]. Note that turbules and other wind noises form peaks, meaning some wind noise is not removed with a simple exponential fit [1].

Chapter 3

Implementation

3.1 Repository overview

Detailed below is the layout of my repository:



Red folders are not included in the attached code files due to their large size but are crucial for context. My WER code is stored in a separate project since it requires a version of Tensorflow which conflicts with other libraries in the main project. I wrote all code in this repository from scratch, except for the PCA function, which was heavily adapted from existing Python code and external resources [50][51], and imported libraries like Tensorflow and SciPy. My dissertation will comment on each folder's contents in the labelled sections.

3.2 Data Collection and Dataset Creation

I originally planned on collecting wind noise, both from stationary and moving positions, using a two-microphone setup, with one microphone covered with wind-cancelling foam. This had the intent of producing side-by-side clean and noisy audio, which I could use directly in testing. However, due to time shift issues (from each microphone being out of phase), attenuation issues (from the foam reducing input volume by $\sim 27\%$) and pollution issues (our recorded real-world audio contained environmental noises), I had to instead produce side-by-side audio algorithmically through combining clean speech (from the LibriSpeech dataset [20, 21]) and clean wind signals in software.

I first collected three types of wind noise:

- 4 hours, 15 mins of *stationary wind noise*, recorded in the day from stationary positions, with strong convective and mechanical turbulence present. *Note this is a continuous signal, so I can simply add it to speech to produce a noisy signal.*
- 35 minutes of segmented *bike wind noise* gusts, collected from clean *segments* of wind noise recorded from a moving bicycle; and featuring strong convective turbulence and added mechanical turbulence due to the moving pose. Note that *this signal is not continuous* due to me having to remove environmental sounds (e.g. traffic) and breathing sounds manually – hence *I use the algorithm in Section 3.2.1 to produce a continuous wind noise signal.*
- 5 hours of *wind background noise*, collected from a stationary position at night. Due to low temperatures at night, this signal only contains isolated weak convective turbulence (Section 2.3.3).

Our collected wind noise and speech dataset is segmented into training/development/testing datasets, in the ratio 7:1:1. I then combine my 44.1kHz wind noise with the 16kHz voice signals by downsampling the former to 16kHz, using a sinc resampling algorithm (Algorithm 1).

Algorithm 1 Resample a signal $x(t)$ using sinc interpolation from a sample rate of A Hz to B Hz

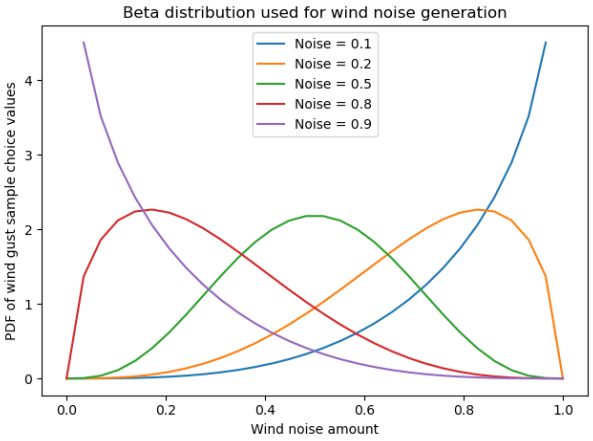
Require: **Lowpass**(x, A, B) applies a lowpass filter to x, attenuating frequencies $< B$ Hz.
Require: **EGCD**(A, B) returns the GCD, LCM and quotients of the two frequencies.
Ensure: $y(t)$ is a equivalent signal to $x(t)$ albeit with sample rate B Hz

function RESAMPLE(x, A, B)
  if $A \equiv 0 \mod B$ **then** ▷ There is no need to perform sinc interpolation
    return $x[:: A/B]$ ▷ Sample every A/Bth sample in x and return
  end if
  if $A > B$ **then** ▷ Downsampling ∴ lowpass needed due to the sampling theorem
    $x \leftarrow$**Lowpass** (x, A, B)
  end if
  $gcd \leftarrow$ **EGCD**(A, B)
  $z(t) \leftarrow$**array**$[gcd.quotient_A \cdot A]$
  $z[:: gcd.quotient_A] \leftarrow x(t)$
  $z(t) \leftarrow \sum_{n=-\infty}^{\infty} z(t) \cdot \text{sinc}(A \cdot t - \frac{n}{gcd.quotient_A})$ ▷ Convolution with scaled sinc function
  $y(t) \leftarrow z[:: gcd.quotient_B]$
  return y
end function

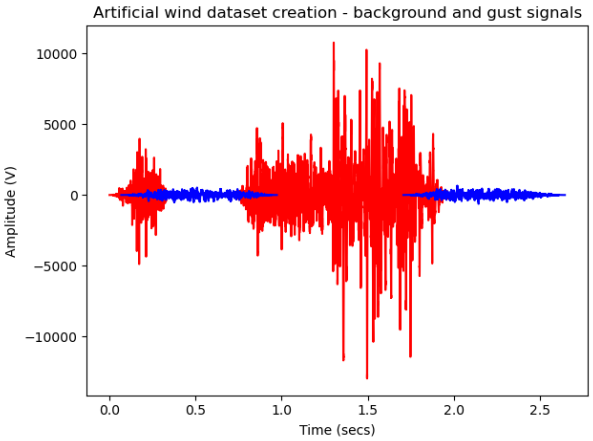
For stationary wind noise, I then perform a weighted-add of the wind noise signal and voice signal to produce its corresponding noisy signal.

3.2.1 Algorithmic Bike Wind Noise Generation

However, for bike wind noise, I utilise the volume threshold algorithm defined in Section 3.3.4 to detect and extract gusts, then order the gusts by mean volume, allowing me to fetch wind noise



(a) A graph of the `windGustCDFValue` wind noise distribution, for a set of wind intensities. I use values $noise = \{0.2, 0.5, 0.8\}$ for my wind noise.



(b) A plot of wind gusts (red) and wind background noise (blue), with Hann windows applied to each for equal-voltage cross-fading.

Figure 3.1: Graphs illustrating the process of producing my algorithmic bike wind noise.

of a specific intensity. I first define the `windGustCDFValue` CDF based on *beta distributions* (shown in Figure 3.1a):

$$\texttt{windGustCDFValue}(\texttt{noiseAmount}) = \begin{cases} \beta(4, \texttt{noiseAmount} \cdot 8) & \text{if noiseAmount} < 0.5 \\ \beta((1 - \texttt{noiseAmount}) \cdot 8, 4) & \text{otherwise} \end{cases}.$$

Using this CDF, we can pass in a noise parameter $\texttt{noiseAmount} \in [0, 1]$ to vary wind noise strength; sampling my CDF gives a new value $\in [0, 1]$ which can be used either to index into the list of volume-ordered gusts, or as the input to the wind gust duration function defined in Section 3.3.5. Consequently, larger `noiseAmount` values result in *increased gust volumes* and a *shorter gap between wind gusts*, creating *stronger-sounding wind noise*.

I then use this gust-fetching method, along with corresponding wind 'gaps' (sourced from a continuous wind background signal, which is segmented into gusts using the wind gap duration function – Section 3.3.5) to produce 'gust-gap' signal pairs. These pairs are then interleaved together, with Hann half-windows of length 0.3s applied to either end of each signal. A 0.3s overlap-add is then used as an equal-voltage crossfade to interleave these signals into one continuous signal, as shown in Figure 3.1b. Note this forces a minimum duration for gusts/gaps (at 0.3s each). Finally, my wind and speech signals are weighted-added to produce side-by-side audio for training and testing.

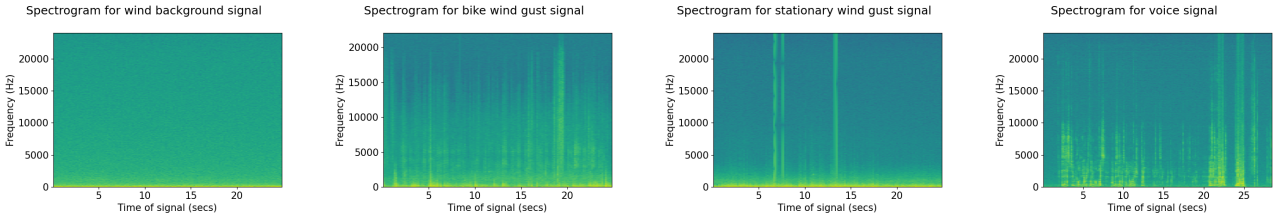
A total of 97 hours of algorithmically-generated 'fake' wind noise (from bike wind gusts) was produced (with corresponding speech signals and transcripts), at three different noise values (noise $= \{0.2, 0.5, 0.8\}$), with a 75:12:12 hour split for training, testing and development datasets. 175 hours of real (stationary) wind noise (with corresponding speech signals and transcripts) was produced in two separate recordings, with a similar 125:25:25 hour dataset split.

3.3 Analysis of Wind Noise

3.3.1 Spectrograms of Wind Noise

Our background wind noise spectrogram (Figure 3.2a) features a relatively time-invariant frequency spectrum, with small peaks at 500Hz and 1100Hz (likely due to microphone-turbulence interaction at a constant shear angle producing *resonant frequencies*) [49]. A clear b/f^a exponential trend is present, with more power in lower frequencies (particularly $<$ 400Hz).

The bike wind gust spectrogram (Figure 3.2b) features a similar b/f^a trend to background



(a) (b) (c) (d)

Figure 3.2: The spectrogram of a wind noise segment captured (a) from a stationary pose at night, (b) on a bike, (c) from a stationary pose during the day; and (d) the spectrogram of isolated male speech.

noise, albeit with additional wind noise in the inertial sub-band (particularly $0-650$Hz). Gusts appear as peaks and amplify noise at all frequencies, especially < 650Hz. Resonant frequencies are also present in this signal, with strong resonance at $\sim$ 900Hz, and weaker resonance at $\sim$ 3750Hz and $\sim$ 5kHz.

The stationary wind gust spectrogram (Figure 3.2c) features a similar b/f^a trend, albeit with much stronger volumes, particularly at lower frequencies. Resonant frequencies are again present, with a large peak at $\sim$ 2500Hz. I note the large uniform peaks at 7s, 8s and 13s, caused by the amplitude of the signal exceeding $2^{15}-1$. This results in an *amplitude integer overflow/underflow*, introducing a 'click' sound – these are removed in Section 3.6.2.

I note that the stationary and bike wind signals consist of wind 'background' noise, plus additional wind gusts due to increased convective and mechanical turbulence. This is exploited in Section 3.2.1 by combining isolated wind gusts and background wind noise to generate continuous wind noise algorithmically.

3.3.2 Spectrogram of Speech Audio

The voice spectrogram (Figure 3.2d) features a male voice speaking at different volumes. Unlike the wind spectrograms, the voice spectrogram features distinct frequency bands caused by vocal folds (in the mouth) vibrating at different speeds (dependent on the syllables spoken), resulting in dominant frequencies (voiced by the folds directly) and harmonic frequencies [52]. I highlight that vowels are both lower-pitched than consonants [53] and more important for understanding [54], hence highlighting the importance of removing low-frequency wind from noisy speech.

3.3.3 Volume Statistics of Wind Noise and Speech

Our volume graph (Figure 3.3) is particularly informative – as shown, wind gusts are often much louder ($3-10$dB higher) than speech, reaching $\sim$ 0dB for strong winds (we set 0dB as the loudest possible sound). However, speech has 5–10dB^2 more variance than wind due to speech having a silent noise floor – this indicates that variance could be used for an indicator function for wind noise. Finally, I note that the maximum volume of the background wind noise is -29.8dB – this is exploited in Section 3.3.4 to detect the presence of wind gusts.

3.3.4 Extraction of Wind Gusts from Wind Noise

I utilise the maximum volume of background wind noise (i.e. no gusts present) defined in Section 3.3.3 (-29.8dB) to hypothesise a threshold function that classifies a wind noise section as a 'wind gust' when its volume exceeds the threshold. This enables us to extract wind gusts, both for analysis and for producing algorithmic wind noise, as in Section 3.3.5. As demonstrated in Figure 3.4, this algorithm effectively detects and extracts wind gusts.

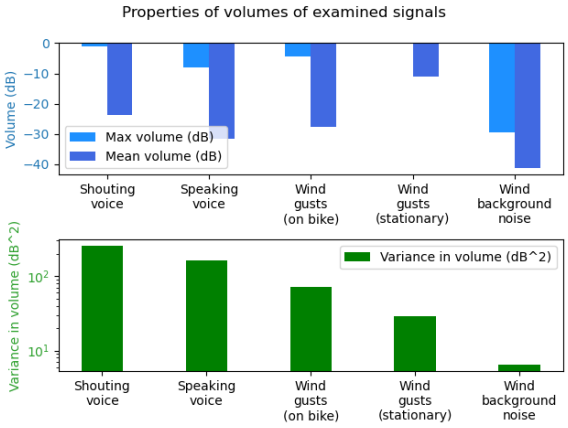


Figure 3.3: The mean volume, max volume and variance in the volume of each type of wind gust/voice.

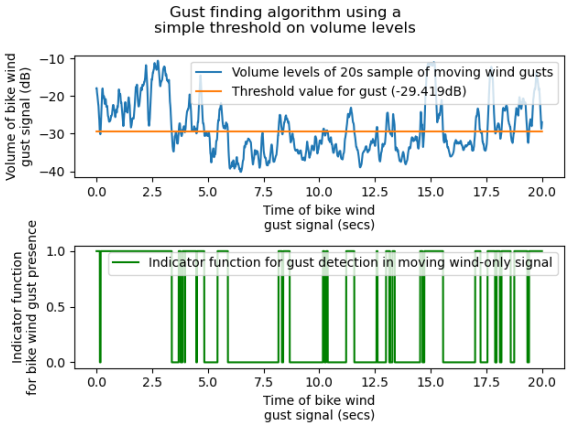
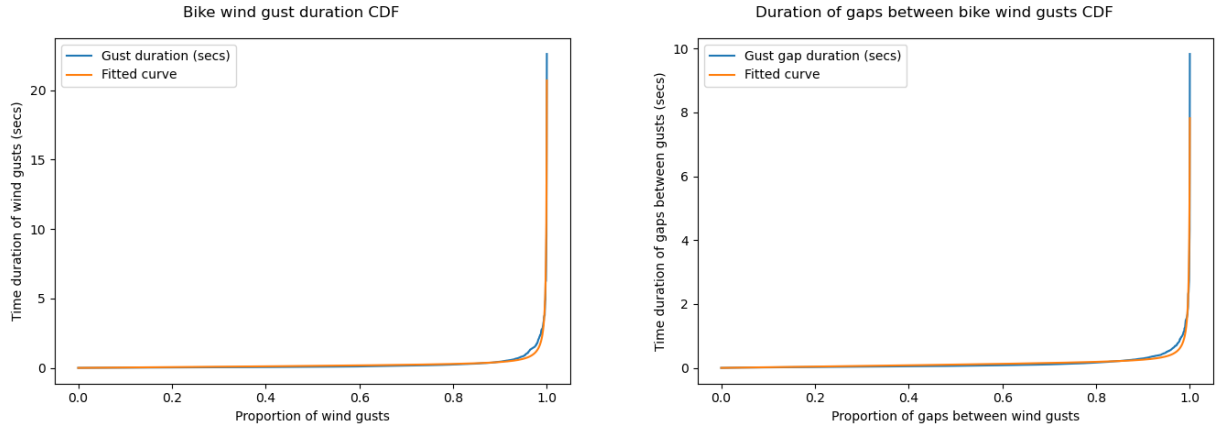


Figure 3.4: A graph of my wind gust detection method, with both time-domain and indicator function views.



(a) (b)

Figure 3.5: Cumulative distribution functions illustrating (a) the duration of wind gusts and (b) the duration of gaps between wind gusts, for bike wind noise.

3.3.5 Properties of Wind Gusts in the Time Domain

As shown in Figure 3.5, my wind gust duration and wind 'gap' duration (i.e. the time between wind gusts) cumulative distribution functions can be approximated using the equation:

$$\text{cdf}_{\text{wind}}(x) \approx a/b - x.$$

Therefore, using SciPy's `curve_fit` function, I fit each curve to the equation:

$$F(x) = ax + \left(\frac{1}{(b-x)c} - \frac{1}{bc}\right),$$

where a, b and c are constants and $\text{cdf}_{\text{wind}}(0) = 0$. This gives the fit:

$$D_{gust}(x) = 0.242x + \left(\frac{1}{(1.0011 - x) \cdot 43.44} - \frac{1}{1.0011 \cdot 43.44}\right),$$

for the duration of bike wind gusts; and

$$D_{gap}(x) = 0.186x + \left(\frac{1}{(1.0013 - x) \cdot 99.21} - \frac{1}{1.0013 \cdot 99.21}\right),$$

for the time between bike wind gusts in seconds[1]. *The latter formula is used to sample wind gap durations for my wind noise generation algorithm* (Section 3.2).

[1]Similar formulas are derived in Appendix B.2 for stationary wind noise.

3.3.6 Power Spectrum Density of Wind Noise and Speech

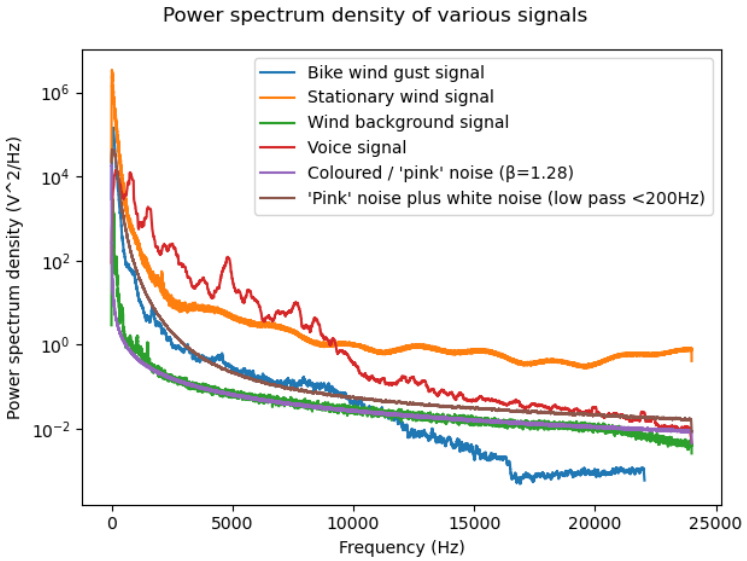


Figure 3.6: A graph showing the periodograms of speech and wind signals, as well as coloured noise approximations to the wind power spectra. Each periodogram is created using 90 minutes of each signal.

Figure 3.6 shows the periodograms for each type of input signal used in our project. I highlight that wind noise dominates the frequency spectrum for frequencies $\leq$ 500Hz, with speech dominating frequencies in $(500, 10000]$Hz. Beyond 10kHz, the stationary wind signal dominates (this can be ignored due to its negligible volume and since signals are downsampled to 16kHz). Wind resonant frequencies appear as small peaks, deviating from the b/f^a trend of each signal, whilst speech dominant frequencies produce more prominent, wider peaks, with more fluctuations in its PSD. Interestingly, the speech also roughly follows a ab^f trend.

I also plot two additional periodograms in Figure 3.6, each representing coloured noise of the form $P(f) \propto \frac{1}{f^\beta}$, where β is a constant. This highlights that *coloured noise ($\beta = 1.28$) closely approximates the* ***mechanical*** *turbulence of wind background noise*, differing from the $\beta = \frac{5}{3}$ quoted in research [10]. By adding low-passed ($<$ 200Hz) white noise, I can also approximate stationary and bike wind signals.

3.4 Original Nelke-Vary Algorithm

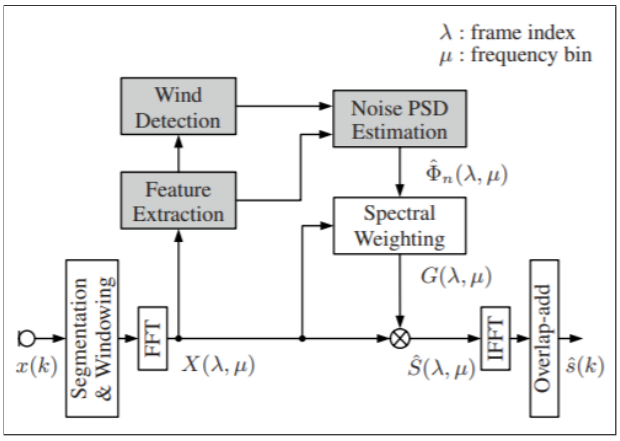


Figure 3.7: A diagram of the workflow of the Nelke-Vary algorithm. Algorithms specific to this algorithm are included in grey[1].

The Nelke-Vary algorithm (Figure 3.7) [1] is designed to remove wind noise from audio through "exploiting [the b/f^a] decay of the magnitude spectrum [...] of wind noise"[1, 49], through fitting an exponential curve to the PSD of the noisy signal; and using this to approximate gains that denoise the noisy signal.

3.4.1 Segmentation, Windowing, and FFT

Given an input signal $x(t)$, the algorithm performs preprocessing by removing clicks (explained in Section 3.6.2), then splitting the signal into *frames* $x(\lambda, t)$ of length $2^{\texttt{exponent}}$, where λ is the frame number. Here, exponent = 9, resulting in a frame length F of 512 samples, or 32ms at a 16kHz sample rate.

Padding is added to the signal to ensure it is an exact multiple of $\frac{F}{2}$; then the signal is segmented (with a 50% overlap between frames), and a Hann window applied to each frame (Section 2.3.1). Windowing the signal enables stream processing and greatly reduces processing latency. I then compute the FFT of each frame, giving us the FFT frames $X(\lambda, \mu)$, where μ is the frequency band.

3.4.2 Feature Extraction and Wind Detection

For each FFT frame $X(\lambda, \mu)$, I compute its PSD $\phi_x(\lambda, \mu)$, which approximates the absolute power of each frequency, then use this to compute the "spectral 'centre of gravity'"[1] of each frame. This uses the spectral sub-band centroid $SSC_x(\lambda)$ of each frame x, given by

$$SSC_x(\lambda) = \frac{f_s}{2^{\text{exponent}}} \cdot \frac{\sum_{\mu \in M} (\mu \cdot \hat{\phi}(\lambda, \mu))}{\sum_{\mu \in M} \hat{\phi}(\lambda, \mu)},$$

where M is the set of frequency bands μ. The SSC of each frame is used to approximate the amount of wind noise present. The average SSC of wind noise is $\sim$ 215Hz, whilst the average SSC of speech is $\sim$ 954.6Hz – hence by using two thresholds ($f_1 = 250$Hz and $f_2 = 650$Hz), I can determine if each frame is all wind, all speech, or a combination of the two.

3.4.3 Noise PSD Estimation

I now define the wind estimate PSD $|\hat{N}_{\text{exp}}(\lambda, \mu)|^2$:

$$|\hat{N}_{\text{exp}}(\lambda, \mu)|^2 = \frac{b}{\mu^a}.$$

This exploits the exponential b/f^a trend exhibited by wind noise (Section 2.3.3). To calculate a and b, I sample two locations within each PSD frame, choosing locations that lie outside the harmonic frequencies of speech [52, 1] which, when sampled, *approximate the underlying PSD of the underlying wind noise*. I derive the below formulas for a and b, as these are omitted in the original paper.

Derivation: *Derive the values of a and b as a function of $\hat{N}_{exp}(\lambda, \mu)$, μ_1 and μ_2.*

I define two frequency bins μ_1 and μ_2 where $\mu_1 < \mu_2$ WLOG. Then, for frame index λ,

$$|\hat{N}_{exp}(\lambda, \mu_1)|^2 = \phi_x(\lambda, \mu) = A_1 = \frac{b}{\mu_1^a}, \tag{1}$$

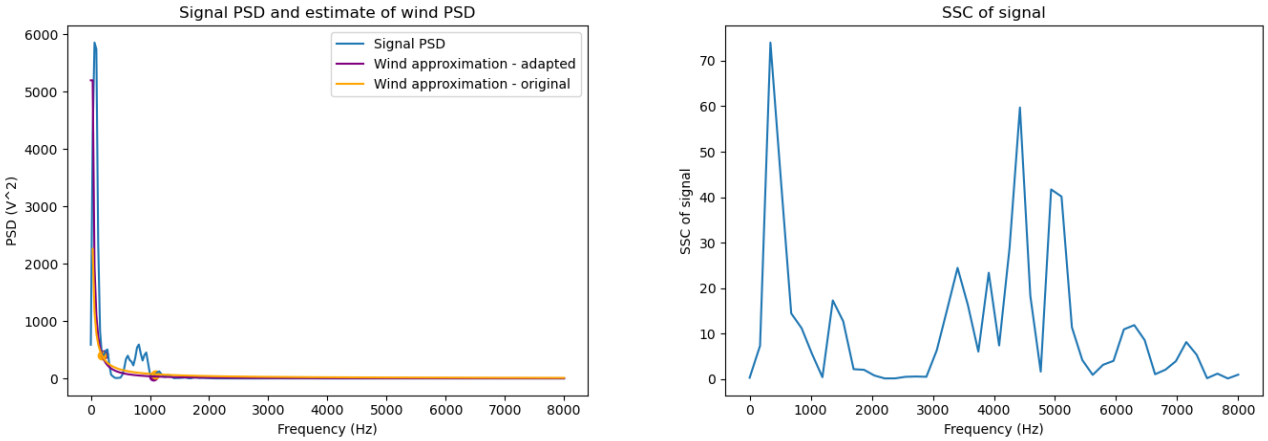
and symmetrically for μ_2. Using (1), we now derive a:

$$\frac{A_1}{A_2} = \frac{\mu_2^a}{\mu_1^a} = \left(\frac{\mu_2}{\mu_1}\right)^a \implies a = \frac{\log\left(\frac{A_1}{A_2}\right)}{\log\left(\frac{\mu_2}{\mu_1}\right)}. \tag{2}$$

Plugging (2) into (1) then enables us to derive b:

$$A_1 = \frac{b}{\mu_1^a} \implies b = A_1 \cdot \mu_1^a = A_1 \cdot \mu_1^{\frac{\log\left(\frac{A_1}{A_2}\right)}{\log\left(\frac{\mu_2}{\mu_1}\right)}}. \tag{2}$$

□



(a) (b)

Figure 3.8: Graphs showing the process of estimating our noise-removing gains. (a) A graph of the wind noise estimated by the Nelke-Vary algorithm, before and after my improvements (Section 3.6). (b) A graph of the SSC of a noisy speech frame.

Using this fit naïvely would likely result in *an over-approximation of the underlying wind noise spectrum* – the speech PSD is non-zero for all frequencies, hence wherever we sample the noisy signal, we are likely to sample some speech, resulting in a gains over-approximation. To improve performance, the Nelke-Vary algorithm utilises the values of f_1 and f_2 to determine if wind and speech are present – if wind is not present, I set the noise estimate $|\hat{N}(\lambda,\mu)|^2$ to 0. If speech is not present, the PSD itself is used as the noise estimate.

$$|\hat{N}(\lambda,\mu)|^2 = \begin{cases} |X(\lambda,\mu)|^2, & \text{if } SSC(\lambda) < f_1 \\ |\hat{N}_{\text{exp}}(\lambda,\mu)|^2, & \text{if } f_1 \leq SSC(\lambda) \leq f_2 \\ 0, & \text{if } SSC(\lambda) > f_2 \end{cases}.$$

It also defines the function $\alpha(\lambda)$, which *smooths the output gains* by combining the previous noise estimate with the current noise estimate. The amount of smoothing is dynamic, with more smoothing applied if more speech is present (and vice versa).

$$\alpha(\lambda) = \begin{cases} \alpha_{\min}, & \text{if } SSC(\lambda) < f_1 \\ \alpha_{\min} + \frac{(SSC(\lambda)-f_1)\cdot(\alpha_{\max}-\alpha_{\min})}{f_2-f_1}, & \text{if } f_1 \leq SSC(\lambda) \leq f_2 \\ \alpha_{\max}, & \text{if } SSC(\lambda) > f_2 \end{cases}$$

Our smoothed PSD wind estimate $\hat{\Phi}_{\text{wind}}(\lambda,\mu)$ then becomes:

$$\hat{\Phi}_{\text{wind}}(\lambda,\mu) = \alpha(\lambda)\cdot\hat{\Phi}_{\text{wind}}(\lambda-1,\mu) + (1-\alpha(\lambda))\cdot|\hat{N}(\lambda,\mu)|^2.$$

3.4.4 Spectral Weighting

Using this wind estimate $\hat{\Phi}_{\text{wind}}(\lambda,\mu)$ I compute the spectral weighting gains $G(\lambda,\mu)$ using:

$$\bar{G}(\lambda,\mu) = \frac{|X(\lambda,\mu)|^2 - \hat{\Phi}_{\text{wind}}(\lambda,\mu)}{|X(\lambda,\mu)|^2}$$

$$G(\lambda,\mu) = \begin{cases} 0, & \text{if } \bar{G}(\lambda,\mu) \leq 0 \\ \bar{G}(\lambda,\mu), & \text{if } 0 < \bar{G}(\lambda,\mu) \leq 1 \end{cases}$$

These gains are then applied to each frame $X(\lambda,\mu)$ via multiplication, denoising the signal.

$$\bar{X}(\lambda,\mu) = X(\lambda,\mu) \cdot G(\lambda,\mu)$$

3.4.5 Inverse Fast Fourier Transform (IFFT) and Overlap-Add

From here, I perform the inverse FFT on each frame, producing $\bar{x}(\lambda,\mu)$. By performing a 50% overlap-add of consecutive frames, I can inverse the effect of the Hann windows and reconstruct the de-noised signal $\bar{x}(t)$ [1].

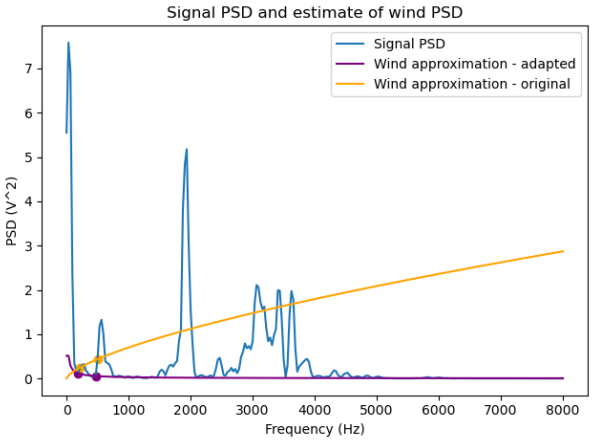
3.5 Hyperparameters of the Nelke-Vary Algorithm

I now enumerate the hyperparameters of the Nelke-Vary algorithm, which the user can adjust to change its performance and operation in the following ways:

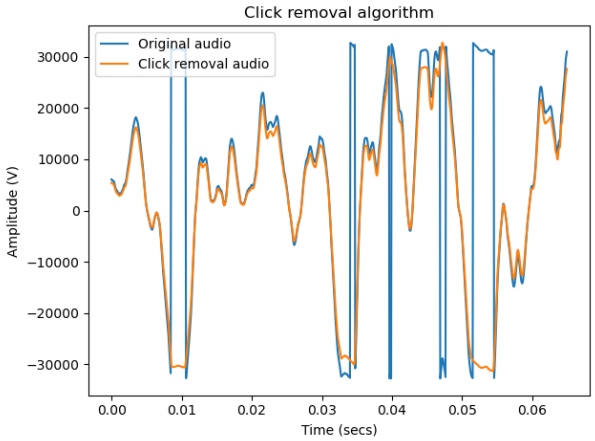
- `sizeOfWindowExponent`– a parameter which defines the size of each time-domain window:

  $$size_{samples} = 2^{\texttt{sizeOfWindowExponent}}$$

  This value defaults to 9, giving a window size of $2^9 = 512$ samples.
- `aMin` – a parameter which defines the minimum smoothing factor over successive frames, if (through my wind noise frequency threshold `f1`) I determine that only wind noise is present. I want *less smoothing* if less speech is present, to enable the *faster prediction of shorter-duration wind gusts*. This defaults to 0.1.
- `aMax` – a parameter which defines the minimum smoothing factor over successive frames, if (through my speech frequency threshold `f2`) I determine that only speech is present. If more speech is present, *more smoothing* will *reduce the effect of incorrectly predicted wind curve fits*. This defaults to 0.9.
- `SSC1MaxFrequency` – a parameter which determines the range of frequencies $[0, \texttt{SSC1MaxFrequency}]$Hz used to calculate each frame's mean frequency. By default, this is set to 3000Hz, as this is roughly the frequency where speech becomes louder than wind; however, I can set this to any frequency $\texttt{SSC1MaxFrequency} \in [0, \frac{\texttt{sampleRateOfAudio}}{2})$Hz.
- `sampleRateOfAudio` – a parameter which defines the sample rate of my input audio.
- `f1` – a parameter defining the maximum mean frequency of a signal frame, below which only wind noise is present. This was calculated using speech and wind development datasets, with `f1` set to 250Hz.
- `f2` – a parameter defining the minimum mean frequency of a signal frame, above which only speech is present. This was calculated using speech and wind development datasets, with `f2` set to 650Hz.
- `placeToSampleNo1` (μ_1) and `placeToSampleNo2` (μ_2) – parameters which determine where I sample the PSD of my noisy signal, to approximate the underlying wind noise. I substitute these sampled values into Equation (1) (Section 3.4.3), for μ_1 and μ_2 respectively, in order to predict the wind noise curve. By default I sample at positions 7 (109Hz) and 35 (537Hz) respectively.
- `removeClicks` – a parameter which specifies if clicks are removed before denoising.
- `adjustedWeighting` – a parameter which specifies if the adapted (True) or original (False) algorithm is ran.



(a) An example where the original Nelke-Vary algorithm fails, and where my adaptations greatly improve performance.



(b) A graph showing how integer overflows/underflows are detected and removed using my 'click removal' algorithm.

Figure 3.9: Graphs detailing the improvements made to the Nelke-Vary algorithm.

3.6 Adaptations to the Nelke-Vary Algorithm

Our current algorithm performs adequately at denoising inputs through approximating the b/f^a PSD of wind noise. However, this method does have two main limitations.

Firstly, *wind noise does not exactly match a b/f^a trend.* This is mainly due to resonant frequencies, created by turbulence-turbulence and turbulence-mean shear interaction, introducing short-term peaks in the power spectrum. As a result, *the base algorithm removes some of the underlying convective and mechanical turbulence but cannot remove these peaks.* These are **especially difficult to target** – their **frequencies and powers change quickly with time** due to the changing direction and strength of the wind. This leaves behind audible artefacts, which I unsuccessfully target in Appendix C.1.

Secondly, by naïvely fitting an exponential curve to each frame, I am likely to incorrectly characterise the underlying wind noise since, if I do sample a PSD at a peak (either due to a wind or speech peak), *the underlying wind noise is over-approximated at that location, leading to an over-zealous exponential fit.*

I now present the improvements that distinguish the original Nelke-Vary algorithm from my adapted Nelke-Vary algorithm. These improvements directly target the second issue.

3.6.1 Heuristic Improvements

Without constraining the values of a and b, *the algorithm makes strange and inaccurate wind noise estimations*, as shown in Figure 3.9a. For example, if a is negative, these gains will increase with frequency, attenuating (higher-frequency) speech and retaining (low-frequency) wind.

Therefore I propose the following algorithm to improve performance:

1. I first smooth the input PSD with a moving average length 3, to remove random peaks.
2. Two locations μ_1 and μ_2 are chosen to sample the PSD at. Using the average value of a for wind noise ($a = 1.28$, Figure 3.6), I then choose $\hat{\mu}_1$ and $\hat{\mu}_2$ such that for each frame λ,

$$\underset{\hat{\mu}_i \in \{\mu_i - 1, \mu_i, \mu_i + 1\}}{\arg\min} \left(\frac{\phi_x(\lambda, \hat{\mu}_i)}{\hat{\mu}_i^{1.28}} \right)$$

is minimised. This avoids larger peaks, and minimises my wind noise fit (to favour under-approximation of wind noise instead of over-approximation).

3. If $\phi_x(\lambda, \hat{\mu}_1) < \phi_x(\lambda, \hat{\mu}_2)$ then $\log\left(\frac{A_1}{A_2}\right) < 0$. Therefore, by the formula $a = \frac{\log\left(\frac{A_1}{A_2}\right)}{\log\left(\frac{\mu_2}{\mu_1}\right)}$, *the value for a will be negative* (Figure 3.9a) – here, I clip a to zero, and use $\phi_x(\lambda, \hat{\mu}_1)$ to calculate b.
4. Finally, I set $|\hat{N}_{\text{exp}}(\lambda, \mu_0)|^2 = |\hat{N}_{\text{exp}}(\lambda, \mu_1)|^2$, since $\lim_{\mu_0 \to 0}\left(|\hat{N}_{\text{exp}}(\lambda, \mu_0)|^2\right) = \infty$. This ensures the DC component ($f = 0$Hz) of the signal is not always fully attenuated.

These improvements (shown in Figure 3.9a) *greatly reduce the change of incorrect wind noise fits* at the cost of *under-approximating the underlying wind noise.*

3.6.2 Click Removal

At high volumes, particularly during strong wind gusts, the absolute amplitude of recorded wind noise often exceeds $|2^{15} - 1|$, the maximum amplitude of an `int16` audio signal. As a result, many frequencies are amplified in the frequency domain, resulting in a very loud 'click' sound. Therefore, I require a click-removal algorithm that *detects and removes integer overflows/underflows.*

By noting that higher amplitudes indicate higher signal power, I first tried removing clicks by attenuating the signal regions that exceeded a volume threshold. However, this attenuated all high-volume regions, including wind noise containing speech. Another option was to lowpass the signal to remove higher frequencies (an integer overflow/underflow creates a sharp time-domain peak and hence higher frequencies). However, this also affects speech, which lies in the higher frequencies that the lowpass filter would attenuate.

My final removal algorithm is Algorithm 2, which removes clicks based on the magnitude of the derivative of the signal amplitude. Sections of the signal are inverted where an overflow/underflow has occurred. A diagram of its operation is given in Figure 3.9b.

Algorithm 2 Remove integer overflows/underflows (clicks) from a signal array $x(t)$

Require: M is the maximum magnitude value represented in the signal.
Require: **CumulativeSum**(x) calculates the running sum of an input xx.
Ensure: $y(t)$ is a signal with integer overflows removed (with best effort)

function ClickRemove(x, T) ▷ Threshold T indicates the distance from M to check for
  $dx(t) \leftarrow \frac{\delta}{\delta t}(x(t))$
  $thresholdClose \leftarrow 0 :: [\mathbb{1}_{|dx(t)|>(M-T)}]$ ▷ Check if signal derivative exceeds magnitude threshold
  $edgeDetectionThreshold \leftarrow$ **CumulativeSum**$(thresholdClose)$ (mod 2)
  ▷ Detect if within integer overflow/underflow section
  $\bar{y}(t) = x(t)$
  for all t **do**
    if $edgeDetectionThreshold[t] = 1$ **then**
      if $x(t) < 0$ **then** $\bar{y}(t) \leftarrow x(t) + 2 \cdot M$ ▷ Overflow
      else $\bar{y}(t) \leftarrow x(t) - 2 \cdot M$ ▷ Underflow
      end if
    end if
  end for
  $y(t) \leftarrow \frac{\bar{y}(t) \cdot \min(M, \bar{y}(t))}{\textbf{max}(\bar{y}(t))}$
  return $y(t)$
end function

Note that this algorithm normalises the maximum amplitude in each signal to M – this will impact the volume of quieter signal frames; however, this does enable us to remove $\sim 92\%$ of clicks in each signal accurately and improves speech coherency considerably.

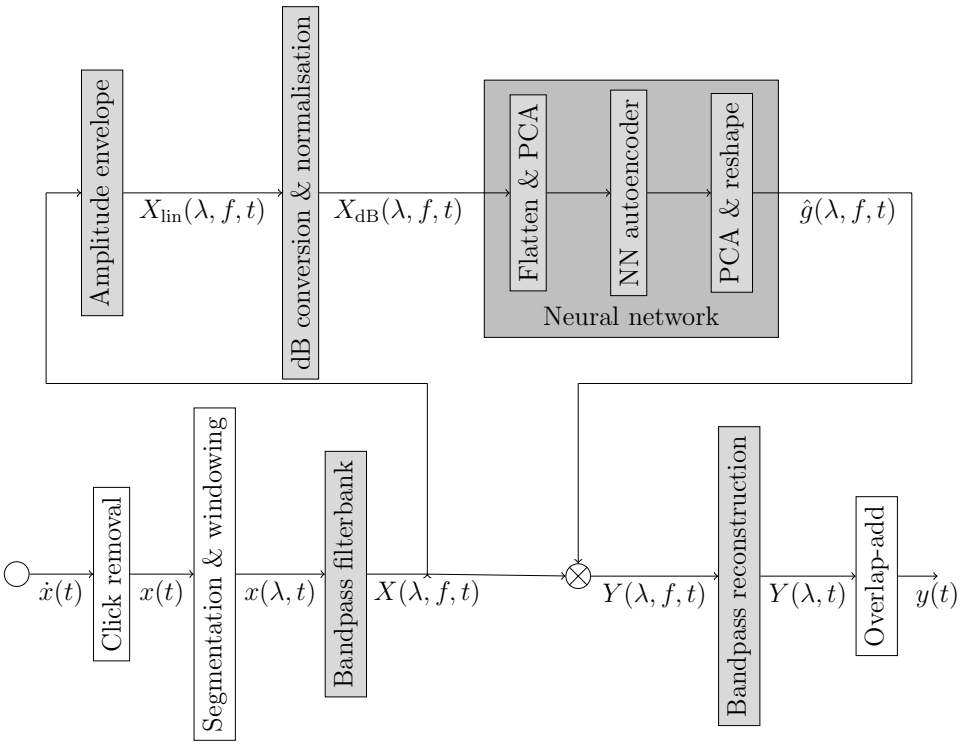


Figure 3.10: A diagram of the workflow of the Lee-Theunissen algorithm. Algorithms specific to this algorithm are included in grey [2].

3.7 Original Lee-Theunissen Algorithm

The original Lee-Theunissen algorithm (Figure 3.10) is a noise reduction algorithm aiming to mimic speech processing in the brain by *projecting spectro-temporal signals into a lower-dimension feature space* that is more conducive to better generalisation and separation of noise from speech [55]. I utilise spectro-temporal detection-reconstruction (STDR) filters, operating on time-domain, frequency-banded data, to produce time-domain band-specific gains that denoise the input.

3.7.1 Click Removal, Segmentation and Windowing

I preprocess the signal similarly to Section 3.4.1 - given an input $\dot{x}(t)$, I remove clicks, split the signal into windows $x(\lambda, t)$ size 20ms, with 50% overlap between consecutive frames, then apply a Hann window to each window (Section 2.3.1).

3.7.2 Bandpass Filtering

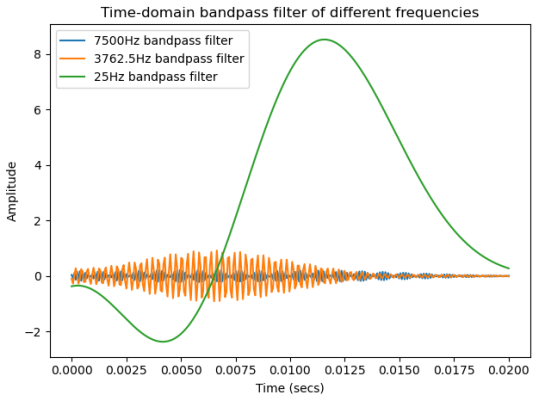
I define a **filter bank** of 223 **Chebyshev Type II** bandpass filters with centre frequencies $\{25 + k \cdot \frac{(7500-25)}{222} \mid k \in \mathbb{N}_0 \cup [0, 222]\}$H, a flat passband of width 256Hz, and $\sim 87.5\%$ overlap between adjacent filters. This differs from the Gaussian-shaped filters used in the paper – a flat passband enables *more accurate signal reconstruction*, and Chebyshev Type II filters offer *better frequency selectivity* than other bandpass filters.

This filterbank is applied to my signal $x(\lambda, t)$, returning the *linear spectrogram* $X_{\text{lin}}(\lambda, f, t)$, where f is the filter number.

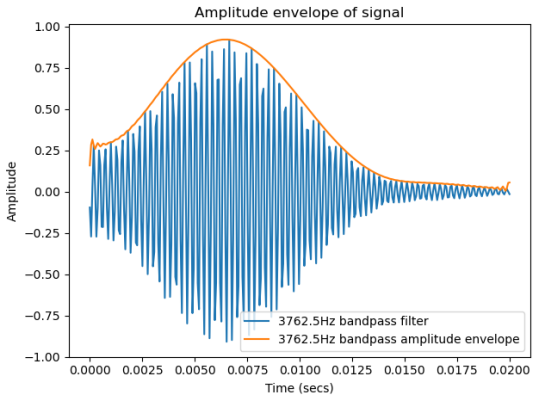
3.7.3 Amplitude Envelope and Normalisation

For each bandpassed signal $X_{\text{lin}}(\lambda, f, t)$, I compute the absolute value of the Hilbert transform of each (the amplitude envelope – Section 2.3.1). This approximates the power of each signal at time t. Each envelope is downsampled to 1kHz, then converted to decibels:

$$X_{\text{dB}}(\lambda, f, t) = 10 \cdot (\log_{10} \frac{X_{\text{lin}}(\lambda, f, t)}{M}).$$

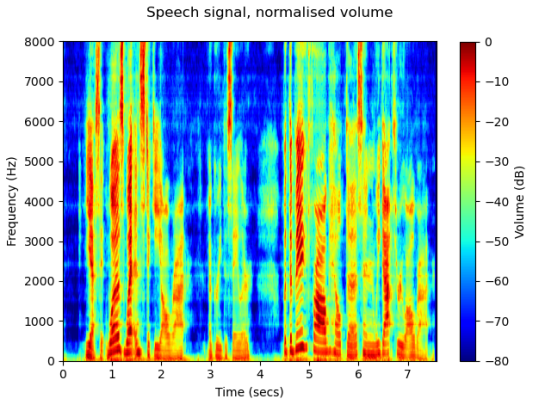


(a) A graph of my bandpassed signals, showing the different frequencies of my input signals.

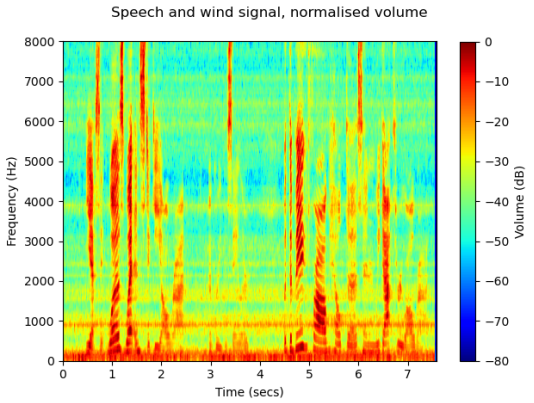


(b) A graph showing the amplitude envelope of a bandpassed signal, which approximates the power of my signal at each point in time.

Figure 3.11: Graphs illustrating the various stages of preprocessing within the Lee-Theunissen algorithm.



(a) A spectrogram of my clean speech signal, after dB normalisation. Note the very clear harmonic frequencies, especially at time 5 secs and frequency < 4000Hz. This is only computed via a debug flag, which fetches the clean speech signal.



(b) A spectrogram of my noisy speech and wind signal, after dB normalisation. Note the horizontal bands of resonant frequencies ($\sim$ 950Hz, 2100Hz) and the additional vertical lines almost indistinguishable from speech.

Figure 3.12: Graph illustrating the spectrogram creation process within the Lee-Theunissen algorithm.

A floor value of -80dB is used to limit the volume domain of the signal, and the mean volume level for each frequency band is subtracted from each signal in my spectrogram. This returns the *log spectrogram* $X_{\text{dB}}(\lambda, f, t)$ [2].

3.7.4 PCA

To reduce the dimensionality of my neural network's input layers, I utilise PCA. This involved creating 4460 PCA components[2] for both clean speech and wind. The wind components were computed by taking 90 minutes of wind, computing the spectrograms $X_{\text{lin}}(\lambda, f, t)$ and $X_{\text{dB}}(\lambda, f, t)$ for each 20ms frame of segmented audio, computing the PCA components for each spectrogram frame, and saving these in order of their corresponding eigenvalues in a Numpy `.npz` file for future component loading; and symmetrically for speech components. I then

[2]Each window of my spectrogram is 4460 samples long (4460 samples = 223 bandpass filters $\times$0.02 secs $\times$1000Hz downsampled amplitude envelope), hence the need for dimensionality reduction.

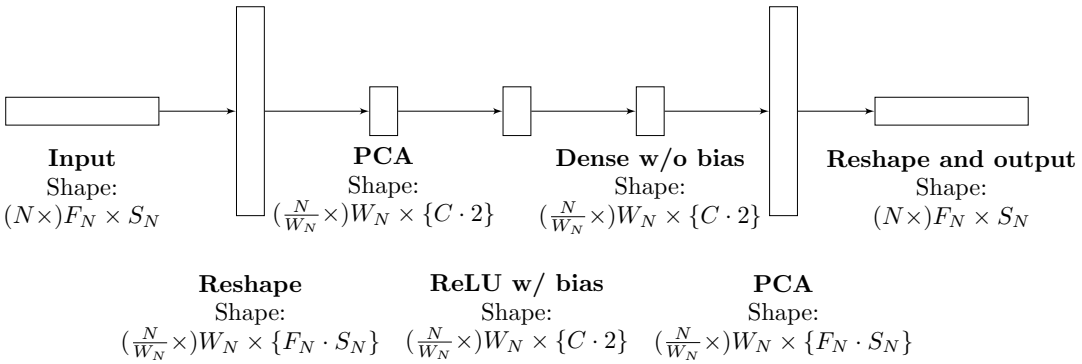


Figure 3.13: A diagram showing the basic architecture of the Lee-Theunissen autoencoder, as it appears in the original paper [2]. Here, N is the number of windows in my signal; F_N is the number of bandpass filters applied to my signal; C is the number of PCA components each from my speech and wind components; S_N is the number of samples in each window in my signal; and W_N is the number of windows passed at a time to the NN, to retrieve additional temporal information.

choose N components (here $N = 50$) each from my speech and wind components to use in my autoencoder's input layers.

3.7.5 Autoencoder

I define a six-layered autoencoder trained to learn the gains:

$$\bar{g}(\lambda, f, t) = \frac{S_{\text{lin}}(\lambda, f, t)}{X_{\text{lin}}(\lambda, f, t)},$$

where $S_{\text{lin}}(\lambda, f, t)$ is the optimal spectrogram of my de-noised signal.

Layout

Figure 3.13 defines the *autoencoder architecture* for my neural network, consisting of six main layers:

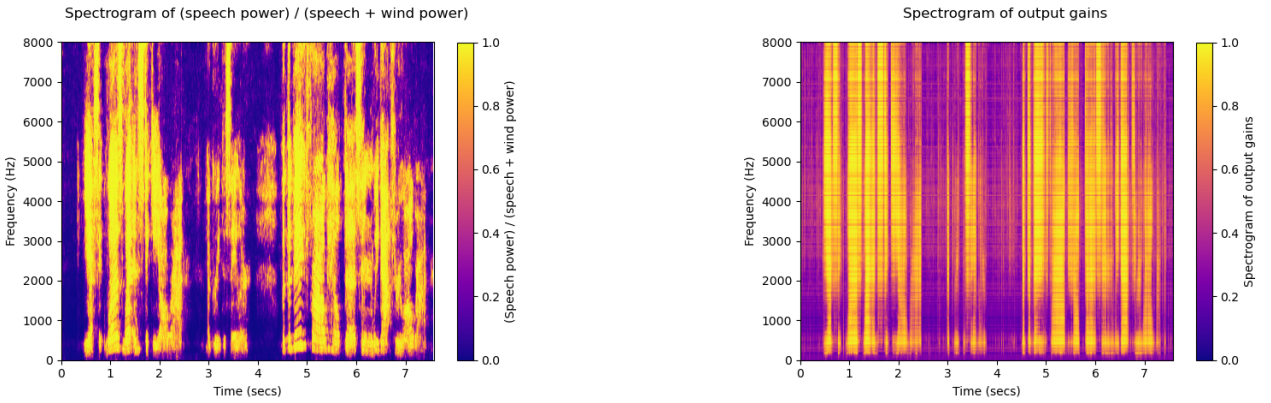
1. A reshape layer that groups together every 5 adjacent frames, enabling the exploitation of temporal information.
2. The spectro-temporal filter layer, using PCA to reduce the dimensionality of the data.
3. A dense layer, with biases and a non-linear ReLU activation function. The non-linearity introduced by this activation function enables us to capture more complex behaviours through the Universal Approximation Theorem [56].
4. A dense layer without biases.
5. The spectro-temporal reconstruction filter layer, using PCA and a sigmoid activation function to reconstruct my output gains $\bar{g}(\lambda, f, t) \in [0, 1]$.
6. Another reshape layer to separate my gains into individual frames.

Training

To train the network, I preprocessed 500 signal pairs (speech signal and corresponding noisy signal) from my training dataset by calculating the log and linear spectrograms $X_{\text{dB}}(\lambda, f, t)$ and $X_{\text{lin}}(\lambda, f, t)$ respectably, then computing the gains $g(\lambda, f, t)$ with

$$g(\lambda, f, t) = \frac{\phi_{\text{lin, clean}}(\lambda, f, t)}{\phi_{\text{lin, noisy}}(\lambda, f, t)}.$$

I then created the pairs $(X_{\text{lin}}(\lambda, f, t), g(\lambda, f, t))$ as my expected inputs and outputs, and saved these in a `.npz` dataset. These are loaded in batches of 512 and are shuffled after each epoch. Training halted once the mean squared losses failed to improve after 25 epochs. 100 signal pairs from my development dataset were used to validate performance after each epoch, with another 100 pairs used to evaluate post-training performance.



(a) (b)

Figure 3.14: Spectrograms of the input gains, showing the difference between the (a) optimal and (b) predicted spectrograms. note that the optimal gains have either near-zero or near-one gains (with few in between), whilst our predicted gains features more intermediate gains values, indicating uncertainty. Also note the horizontal and vertical banding.

3.7.6 Signal Reconstruction

I first upscale the predicted gains $\hat{g}(\lambda, f, t)$ to 16kHz (matching my signal sample rate) using sinc resampling (Algorithm 1), then multiply this with my noisy spectrogram $X(\lambda, f, t)$ of bandpassed signals to get my denoised spectrogram $Y(\lambda, f, t)$. By performing a summation over these bandpassed signals and dividing by the number of overlapping filters, I retrieve my denoised signal frames $Y(\lambda, t)$, which are then overlap-added to reconstruct my denoised signal $y(t)$.

3.8 Hyperparameters of the Lee-Theunissen Algorithm

The Lee-Theunissen algorithm (presented in Section 3.7) features the following hyperparameters:

- `model` – a parameter that defines which autoencoder architecture (NN model) is used to calculate output gains.
- `filterSize` – a parameter that defines the number of consecutive windows passed into the neural network autoencoder at once. A larger value offers improved temporal resolution. However, doubling the number of windows quadruples the number of weights and nodes needed to process each input. By default, I set this to 5, meaning that 0.05s of audio will be passed into my autoencoder at a time (taking into account my 50% window overlap and 0.02s default value for `sizeOfWindowSecs`).
- `sizeOfWindowSecs` – a parameter which sets the size of my windows in seconds. Again, smaller windows offer more temporal information, albeit at the cost of reduced frequency resolution, and vice versa. I set this to 0.02 secs by default.
- `stdDev` and `numberStdDev` – these parameters are used for determining the size of my bandpass filters. The total size of my passband is given as:

$$|\text{passband}|_{\text{Hz}} = \texttt{stdDev} \cdot \texttt{numberStdDev}.$$

  The size of my window directly affects the range of frequencies each bandpassed signal will contain, in turn affecting both the size of the input to my neural network and the amount of frequency overlap between signals. I will refer to this as my `passBand` parameter – however these are separate in my code for testing purposes.
- `numberFilters` – a parameter which defines the total number of bandpass filters in my

filter bank. I apply these filters at equally-spaced centre frequencies defined by the set:

$$\{(25 + k \cdot \frac{7500 - 25}{\texttt{numberFilters} - 1})\text{Hz} \mid k \in [0, \texttt{numberFilters} - 1] \cup \mathbb{N}_0\} \subset [25, 7500]\text{Hz}.$$

- `floorVolume` – a parameter which sets the minimum floor value for my log (dB) spectrogram. This functions identically to the corresponding Nelke-Vary equivalent, and is set to -80dB by default.
- `downsampleFrequency` – a parameter defining the frequency I downsample my analytical signal to before passing as an input to my NN. Smaller values reduce the resolution of my output gains, but of course, I can utilise a smaller, simpler autoencoder architecture with fewer input nodes and weights/biases. This is set to 1000Hz, corresponding to a downsampling ratio of 16 : 1.

3.9 Adaptations to the Lee-Theunissen Algorithm

To improve the performance of this algorithm, I propose the below adaptations:

3.9.1 Adjustments to the Neural Network Architecture

1. Firstly, I note that the PCA components used come equally from speech and wind signals. Since the *wind components* mostly *represent signal components that are later attenuated, it seems wasteful to compute these components as part of the gains reconstruction filter.* Therefore, I propose a new architecture (the 'gains PCA autoencoder'), with the output layer using PCA components derived from a dataset of *optimal gains.* My middle layers then focus on *converting between the two latent spaces.*
2. Secondly, I propose a second architecture that *negates using PCA entirely*, instead using *dense PCA layers that learn a reduced-dimension representation* of the data and can produce denoising gains (the 'no PCA autoencoder').
3. Finally, I note that *increasing the number of PCA components* in the middle layers should *improve performance* by enabling a more accurate representation of the signal to be produced that is used to *reduce error in the returned gains.*

Gains PCA Autoencoder

I propose the following NN architecture for the gains PCA autoencoder (Figure 3.15):

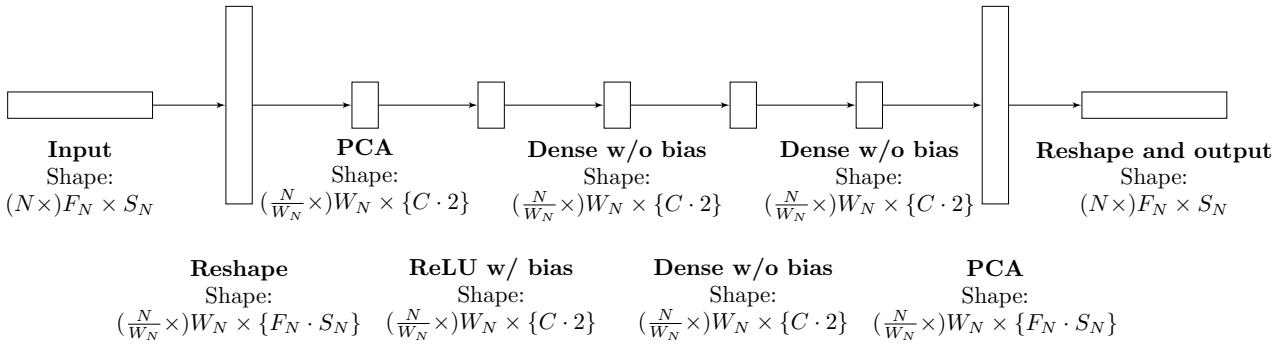


Figure 3.15: A diagram showing the architecture of my 'gains PCA' autoencoder architecture. Separate PCA components are used for the output, based on a development set of optimal gains.

This is similar architecture to the original network, albeit with different PCA components and *two additional dense middle layers to aid conversion between the differing latent spaces* of the two PCA component sets.

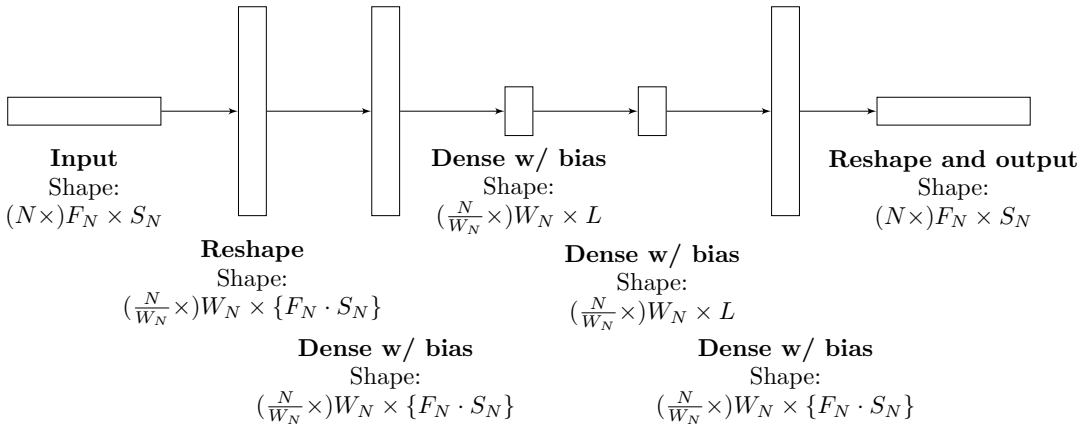


Figure 3.16: A diagram showing the architecture of my no-PCA autoencoder architecture. Here, L is the number of nodes per input frame in the middle layer.

No PCA Autoencoder

I propose the following architecture for removing wind noise, which replaces PCA with dense layers (Figure 3.16) :

Our neural network is split into six layers:

- A reshape layer that groups together every 5 adjacent frames, enabling the *exploitation of temporal information.*
- A dense layer, with biases and a ReLU activation function – this introduces *non-linearity, enabling it to represent more complex patterns*, and enables me to convert the input into a form more conducive to dimensionality reduction [56].
- Two dense layers of size L ($L \in \{100, 1000\}$) that enable the network to become more generalised.
- A final dense layer of size $F_N \cdot S_N$, with a sigmoid activation function applied to restrict gains to the range $[0, 1]$
- A final reshape layer, to match my input shape.

Adjustments to PCA Components

I note that the gains in Figure 3.14b are relatively flat, with little variation in the time domain. However, with an increased number of PCA layers, I should be able to represent the data with less error, extract more complex patterns and hopefully return more accurate, less flat gains. Note that the 'adapted Lee-Theunissen algorithm' is similar to the original architecture, albeit with additional PCA components.

3.10 My Solution

We first note that the Nelke-Vary (Section 3.4) is *rapid to compute*, operates in the frequency domain (*avoiding time-consuming bandpass filtering* and much larger data representations), very effectively removes background wind noise from the signal, and *exploits wind-specific characteristics* (its b/f^a exponential trend); however, it *performs poorly at removing short-term turbules* and can *easily misfit its exponential curve* to the underlying wind noise.

Conversely, the Lee-Theunissen (Section 3.7) algorithm is *much more effective at selectively attenuating wind, even in strong gusts* (due to using a neural network to produce gains) and works in the time domain, hence offering *more granular time-domain gains*; however, it is *much slower* (due to the time-domain filter bank) and *utilises a much less compact data representation.*

My solution (illustrated in Figure 3.17) aims to **combine the fast processing times** of the Nelke-Vary algorithm, with the **more accurate noise removal** of the Lee-Theunissen

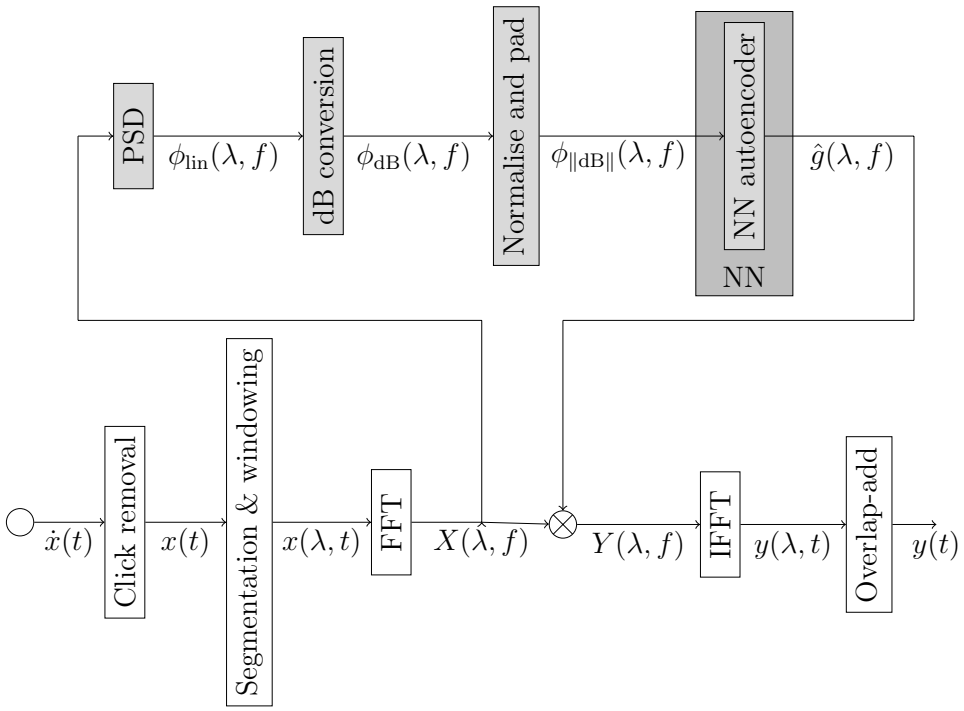


Figure 3.17: A diagram of the workflow of my solution. Algorithms specific to this algorithm are included in grey.

algorithm, by utilising an **autoencoder** that operates on **frequency-domain inputs**. As a result, I avoid the time-consuming process of bandpassing my signals, yet I can also extract wind noise with more granularity than a simple exponential fit.

3.10.1 Input Signal Preprocessing

The preprocessing of my input signal is somewhat similar to both of my algorithms – similarly to the Nelke-Vary algorithm, I first remove clicks, segment my signal into 50% overlap windows (this time of length 100ms), then apply Hann windows to each window. The FFT of each window is then computed, giving us the frequency-domain windows $X(\lambda, t)$.

The PSD is computed for each FFT; then, each PSD frame is converted to a decibel value in the range $[-50, 0]$dB. I then normalise each frames by frequency – that is, the mean of each frequency's volume across all frames is set to 0, and the variance of these volumes to 1. This gives us the output $\phi_{\|\text{dB}\|}(\lambda, f)$.

3.10.2 Phase and Amplitude Data

I note that both the Nelke-Vary and Lee-Theunissen algorithms *do not correct the phase of the output signal*, only the amplitude of each frequency within each window. As a result, *the outputs of each algorithm can never perfectly denoise a signal since the noisy and clean signals differ in phase*, resulting in different time-domain signals.

I originally aimed to produce *two* neural networks, utilising a *polar FFT input* (amplitude and phase) and outputting either the denoised gains or *denoised phase difference*. However, as shown in Figure 3.19a, I notice two characteristics of the optimal phase difference:

1. Firstly, the magnitude of phase differences seems to correlate with the magnitudes of the optimal gains – this makes sense since noisier signals feature more of a noise signal whose phase conflicts with the clean signal. Note that this implies that *smaller gains result in larger phase differences* – thus, *reducing output phase error would only benefit parts of the signal which are heavily attenuated*.
2. Secondly, the phase differences seem very noisy, with *no clear pattern between adjacent phase deltas* in Figure 3.19a, making it unlikely that these phase differences can be eliminated.

3.10.3 NN Autoencoder and Gains Prediction

I define a six-layer NN autoencoder, detailed in Figure 3.18. We split this architecture into three main units:

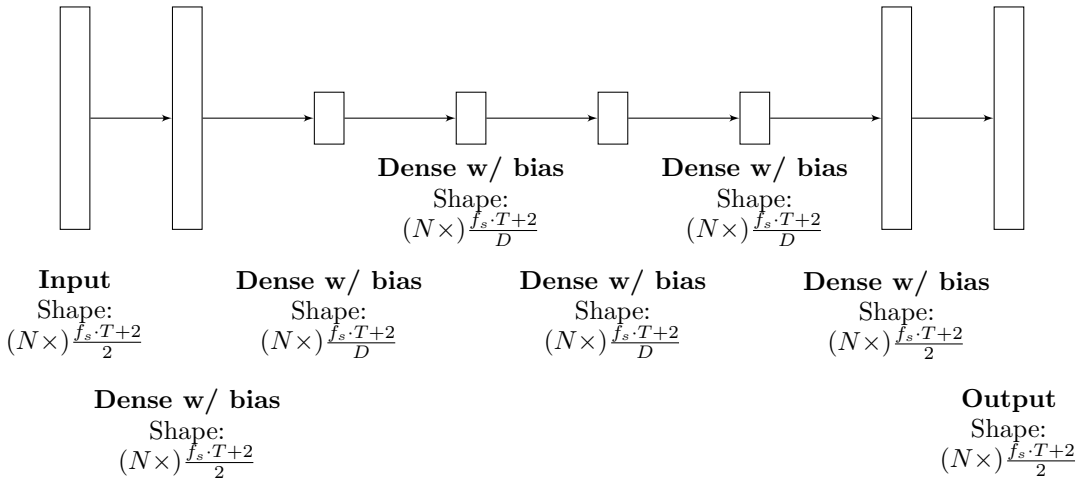


Figure 3.18: A diagram showing the architecture of my solution's AI autoencoder. I note that f_s is the sampling frequency of my signal, T is the length (in seconds) of each window, and D is the division factor used for my middle layers (here, this is set to 4).

- A dense layer, which enables the input to be linearly transformed before dimensionality reduction occurs.
- 4 smaller dense layers, with $\frac{f_s\cdot T+2}{D}$ nodes per layer. I set D to 4; hence (in my case), the number of nodes in each layer is halved. Reducing the number of nodes in these layers roughly quarters the number of multiply-adds needed to compute my gains and enables my network to generalise wind noise better.
- A final dense layer which computes output gains which are the same size as my input. A sigmoid (non-linear) activation function bounds these gains in the range $[0, 1]$.

I incorporate far more layers into this architecture versus the other architectures. This is because my NN operates on one frame at once and so cannot exploit temporal information to extract more accurate gains.

Training

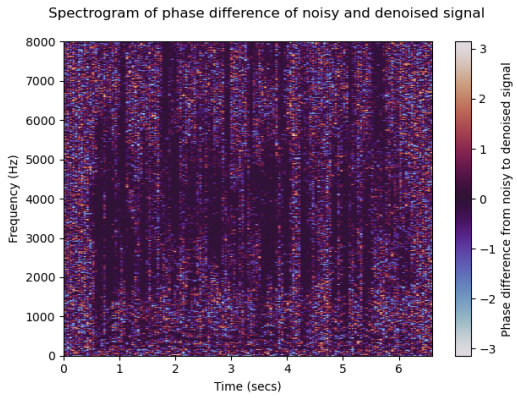
I preprocessed 5477 signal pairs (speech signal and corresponding noisy signal) from my training dataset, into FFT windows $X(\lambda, f)$ and normalised PSD windows $\phi_{\text{dB}}(\lambda, f)$). I then used $\phi_{\text{dB, noisy}}(\lambda, f)$ as my NN input, and defined the output gains $g(\lambda, f)$ as:

$$g(\lambda, f) = \frac{\phi_{\text{lin, clean}}(\lambda, f)}{\phi_{\text{lin, noisy}}(\lambda, f)}.$$

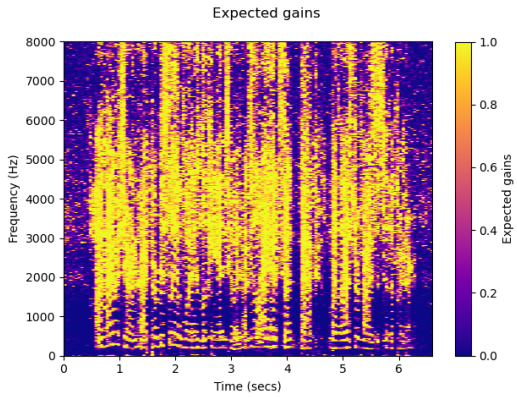
A development set of 694 signals was used to compute losses after each epoch, and a final test dataset of 811 signals used to evaluate the overall performance of my solution. I flatten each dataset into a list of pairs of signal frames for training. My training dataset was randomly shuffled each time, with batches of pairs size 512 passed into the network. Similarly to the Lee-Theunissen architecture, I utilised a mean-squared error loss function and early stopping to stop training after failing to improve for 25 consecutive epochs.

3.10.4 Signal Reconstruction

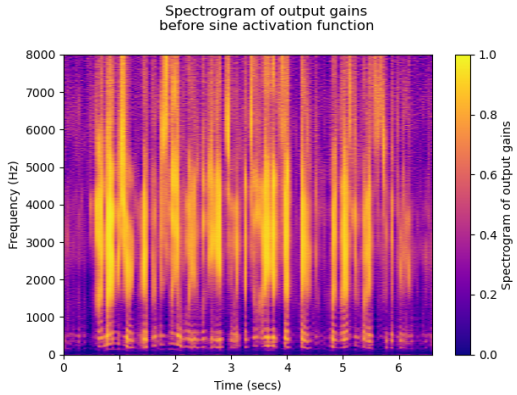
Reconstruction of my signal mirrors the Nelke-Vary algorithm. This involves multiplying my predicted gains $\hat{g}(\lambda, f)$ by the noisy FFT $X(\lambda, f)$, computing the real IFFT of each window, and then overlap-adding each window to retrieve the denoised signal $y(t)$.



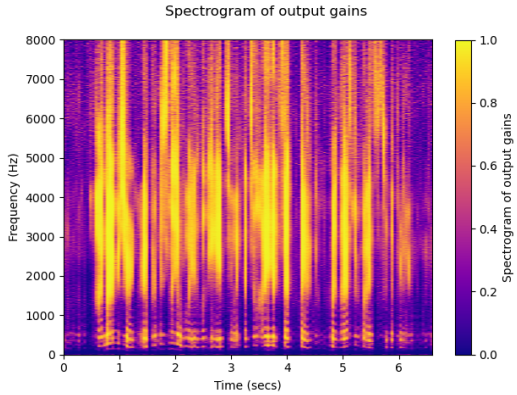
(a) A spectrogram of the phase difference between the input and output.



(b) A spectrogram of the optimal gains for my input signal.



(c) A spectrogram of the output gains from my network, before applying the sine activation function.



(d) A spectrogram of the output gains from my network, after applying the sine activation function.

Figure 3.19: Graphs showing the operation of my solution.

3.11 Adaptations to My Solution

3.11.1 Sine Activation Function

As shown in Figures 3.19b and 3.19c, my expected and predicted gains are noticeably different, with my predicted gains being especially cautious. This may be due to either the neural network being unable to fully represent this data or the wind noise being too random and unpredictable to detect reliably.

However, by applying the sinusoidal function:

$$\bar{g}(\lambda, f) = \sin^2\left(\frac{g(\lambda, f) \cdot \pi}{2}\right)$$

to my output gains, I return the output gains in Figure 3.19d, which produces higher gains for speech and lower gains for wind noise. This results in a *flatter noise floor and louder speech*, at the cost of *slightly less speech clarity* (speech misclassified as noise is now more heavily attenuated, and vice versa). My solution uses the sine activation function – however, for completeness, I also include 'my solution without sine activation' in my evaluation.

Due to exploding gradients, I apply the sinusoidal function after my trained autoencoder returns its predicted gains; however, by utilising gradient clipping [57] or gradient normalisation, which are relatively easy to use in Keras [58], I could have included this activation function in my output layer and perhaps learnt more accurate gains predictions. However, due to time constraints, I could not test this.

Chapter 4

Evaluation

Our evaluation is split into two main sections. Firstly, I provide a comparison of each algorithm, using subjective DCR scores, descriptions of each noise signal, and objective metrics such as SSNR and PESQ. Secondly, I analyse the effect hyperparameters/AI architectures have on each of the Nelke-Vary and Lee-Theunissen algorithms.

4.1 Metrics

In each section, I utilise a selection of metrics to extract the performance of each algorithm:

- **Perceptual Evaluation of Speech Quality (PESQ)** [27] is a common standard for testing the speech quality of a signal and *is used regularly in audio research* [27, 59]. This metric is scored between -0.5 and 4.5, with higher scores indicating a cleaner signal. Two results are presented – PESQ narrow band, which evaluates performance in frequencies < 4000Hz, and PESQ wide band, which evaluates performance in frequencies < 8000Hz.
- **Short-Term Objective Intelligibility (STOI)** [28] is a relatively modern standard *specifically designed for quantitatively evaluating denoised signals* [60]. Scores between 0 and 1 are returned, with a higher score indicating a more coherent voice component.
- **Segmented Signal-to-Noise Ratio (SSNR)** is an improved version of the common SNR evaluation metric, which utilises segmentation to better average the SNR over the entire signal. This metric is bounded in $[0, \infty)$, with larger values indicating less noise.
- **Mean Squared Error (MSE)** is another common metric used to compare signals – by computing differences in the amplitudes of two signals, we return a constant, which increases as error increases.
- **Word Error Rate (WER)** is a common metric used to *compute speech intelligibility.* I use Mozilla's DeepSpeech [61, 62] speech recognition algorithm against my de-noised signals, and the mean word error rate (bound between 0 and 1) is returned. Lower error rates indicate fewer incorrectly recognised words.
- **Mean Opinion Score (MOS)** (Mean Opinion Score) is a subjective test score *commonly used to determine the 'quality of experience' of a given signal*, according to a specific property. Each participant scores each signal using an integer between 1 and 5, with higher scores indicating a better-perceived signal.
- **Degradation Category Rating (DCR)** is an alternative test score used for similar purposes to MOS. Participants judge *the performance of one signal relative to the other*, scoring with an integer between 0 and 1.

I will aim to answer several key questions; namely, a) *how intelligible speech is* for each algorithm, b) *how much wind noise is removed* by each algorithm, c) *how many artefacts are introduced* by each algorithm; and d) *which algorithm is best for different scenarios and purposes.*

4.2 Subjective Comparison of Each Algorithm

I first present the results of human testing. 14 participants were given a selection of 18 signals, *each featuring a single voice masked by wind noise.* Each participant listened to the noisy signal, then the same signal *de-noised using one of five algorithms.* Participants provided MOS

scores for each signal, indicating how clear the voice was in each signal, how many artefacts were present, and how much wind noise was removed. DCR scores were also provided for each algorithm's output, comparing its output with the original noisy signal. Finally, I also asked participants for *subjective comments* on the wind noise, speech and artefacts within each signal. Each participant utilised either over-the-ear headphones or in-ear earphones, and the volume for each participant was calibrated using a test signal. Every participant also signed a consent form – a blank version is included in Appendix E, along with the ethics form (Appendix F).

Note that I did not include all wind strengths in my MOS testing, nor include both of my own solutions (I only used my final solution). This was due to the time taken by each participant to undergo briefing/debriefing and complete the questionnaire – on average, *each participant took* ~ 40 *minutes to complete all questions*, hence doubling the number of questions would likely discourage people from participating.

One issue encountered was that participants would rate initial audio clips very favourably or very poorly, then ask if they could change previous answers. I focus here on DCR scores to avoid this issue – however, MOS scores are included in Appendix D.1.

4.2.1 Subjective Analysis of Noisy Speech Signals

Participants were first asked to quantitatively score and qualitatively describe the three strengths/types of wind noise presented:

- **Weak algorithmic wind of strength '0.2'**. This is light wind, with quiet wind gusts and some "static" background noise. The masked speech signal was "clear" and "understandable", with one stronger gust "masking" two words.
- **Strong real wind noise** *(with peak wind speeds of 27 knots)*. This was characterised as "obstructive", "very gusty", and "distracting" from the "very quiet", "hard to hear" underlying speech signal.
- **Strong algorithmic wind of strength '0.8'**. This features "loud wind gusts", strong "whooshing" background noise, and noticeable clicking artefacts from amplitude overflows/underflows. Participants noted that some words were "not fully decipherable" due to wind gusts.

4.2.2 Subjective Analysis of Denoised Signals

	Algo wind – 0.2		Real wind – Strong		Algo wind – 0.8	
	Mean	Variance	Mean	Variance	Mean	Variance
Nelke-Vary Original	4.21	0.883	**4.86**	**0.122**	**4.5**	**0.25**
Nelke-Vary Adapted	3.79	0.597	4.57	0.531	4.29	0.633
Lee-Theunissen Original	3.71	**0.204**	3.57	0.245	3.5	0.536
Lee-Theunissen Adapted	3.57	0.531	3.64	0.801	3.5	0.964
My Solution	**4.43**	0.388	4.5	0.25	4.36	0.801

Table 4.1: Human DCR scores for wind noise.

Overall, my scores seem positive – all DCR scores are above 3, indicating that my algorithms remove some wind noise. However, score variance is particularly high for some algorithms, indicating that their wind denoising performance is particularly subjective/not consistent.

Nelke-Vary Algorithm

The Nelke-Vary algorithm performed very well at removing wind noise, with the original algorithm performing best for stronger wind and offering decent performance at lower wind speeds, albeit with a much higher variance of ~ 0.7 across the two algorithms. Note that my original algorithm offered better wind noise removal performance – when a frame is fitted incorrectly

with a wind fit, it often results in overapproximation of the wind noise, resulting in a near-silent frame that attenuates speech and wind. My adapted Nelke-Vary algorithm averaged a DCR score ~ 0.31 lower than the original algorithm, indicating worse wind noise attenuation.

Lee-Theunissen Algorithm

Our Lee-Theunissen algorithm performed relatively well at attenuating wind noise but failed to remove it fully – participants quantised the wind intensity of the denoised signal at "$\sim 50\% - 75\%$" of the original noisy signal. This explains the (on average) ~ 0.2 lower DCR scores versus other algorithms. My improvements result in roughly equivalent performance in wind noise attenuation, albeit with much higher variance (disagreement) in scores.

My Solution

My solution roughly matched the Nelke-Vary algorithm's performance at removing wind noise, with my solution performing better at lower wind intensities but worse at higher intensities. It offered less consistency than the Lee-Theunissen algorithm – perhaps since it cannot exploit temporal patterns in the wind noise – but **it did offer a "very flat noise floor"**.

4.2.3 Subjective Analysis of Artefacts

	Algo wind – 0.2		Real wind – Strong		Algo wind – 0.8	
	Mean	Variance	Mean	Variance	Mean	Variance
Nelke-Vary Original	1.79	1.45	1.5	1.11	1.5	0.821
Nelke-Vary Adapted	2.64	1.37	1.43	1.1	1.71	0.776
Lee-Theunissen Original	3	1.29	2.93	**0.638**	3.14	**0.694**
Lee-Theunissen Adapted	3.14	**0.694**	**3.21**	0.74	3	1.14
My Solution	**3.64**	0.801	2.93	1.21	**3.21**	0.74

Table 4.2: Human DCR scores for artefacts.

Unlike wind noise, my artefact DCR scores vary considerably, both in terms of mean score and variance.

Nelke-Vary Algorithm

The Nelke-Vary algorithms offered inferior DCR performance for wind artefacts, with an average of 1.6 for the original algorithm and 1.93 for my improved algorithm. Participants noted that artefacts were "very loud", "unpredictable", and "short"-term, perhaps indicating incorrect wind noise exponential fits. "Warbling" artefacts were also present, which could be explained by turbules (Section 2.3.3) not being properly removed by exponential fits (Section 3.8a)), hence leaving behind frequency-changing short term peaks in the frequency domain. Participants also commented on the number of clicks present for louder wind signals, which caused additional loud short-term artefacts.

Lee-Theunissen Algorithm

The Lee-Theunissen algorithms offered respectable performance for artefacting, with both algorithms scoring > 3 in most cases. These algorithms managed to preserve not only speech but also background artefacts such as traffic noises (from the pollution of our wind noise dataset). Therefore, this algorithm performed very well at selectively removing wind noise from the noisy speech signal. This is likely due to the more granular time-frequency gains returned by the autoencoder [2].

In terms of added artefacts, participants noted a static sound throughout the recording, which is likely due to us operating in the time domain – an error in one amplitude gain introduces many frequencies into the resulting signal, producing white noise. This algorithm also offered

improved attenuation of the remaining clicks (not removed by our click removal algorithm – Algorithm 2) compared to the other algorithms.

My adaptations to this algorithm offered marginal improvements in mean artefact DCR scores at the cost of increased variance.

My Solution

My solution offered strong performance in artefact DCR scores, scoring highest in algorithmic wind noise signals but lower in the real wind noise signal. Participants highlighted the artefacts as being mainly quieter "short-term pitch-changing" artefacts (perhaps due to the lack of temporal information), as well as unattenuated clicking, both of which do drown out the voice signal. The "quiet noise floor" however indicated a lack of artefacts in quieter sections of the denoised signal.

4.2.4 Subjective Analysis of Denoised Voice Coherency

	Algo wind – 0.2		Real wind – Strong		Algo wind – 0.8	
	Mean	Variance	Mean	Variance	Mean	Variance
Nelke-Vary Original	2.14	0.408	2	1.43	2.36	0.944
Nelke-Vary Adapted	3.36	0.658	1.5	**0.393**	2.5	1.39
Lee-Theunissen Original	3.29	0.633	3.29	0.633	3.36	0.944
Lee-Theunissen Adapted	3.57	**0.388**	3.5	**0.393**	3.21	**0.597**
My Solution	**4.14**	0.551	**3.57**	0.531	**4**	0.857

Table 4.3: Human DCR scores for speech.

Nelke-Vary Algorithm

The Nelke-Vary algorithms performed very poorly in speech DCR scores, averaging at least ~ 1.1 points below all other algorithms presented, and produced high variance in scores, especially for stronger wind. Participants noted that voices in my denoised signal sounded "robotic", "distorted", "quiet", and "hard to decipher", perhaps indicating that poor wind noise exponential fits resulted in uneven voice attenuation across the frequencies.

My adaptations offered better speech performance, especially in weaker wind conditions (where this algorithm offered similar performance to the AI solutions), albeit at much quicker processing speeds. This is due to a reduced chance of misfitting the wind noise, resulting in a "clearer", "smoother" voice signal.

Lee-Theunissen Algorithm

The Lee-Theunissen algorithms offered consistently good scores for voice clarity, with an average DCR score of 3.37 – hence improving the original signal's coherency. Many participants noted that the speech signal was "quiet" yet "decipherable" and "clearer" than the noisy signal. However, some "voice distortion" artefacts were present, likely due to small gains errors.

My Solution

My solution performed consistently better than the other algorithms for speech clarity, with average DCR scores 0.53 higher than the (next best) Lee-Theunissen algorithms. However, the adapted Lee-Theunissen algorithm was more consistent in performance, with score variance on average 0.187 lower than my solution.

Participants noted that, while not all words were fully decipherable, the voice signal was "clear" and "loud", albeit with "fluctuations in speech volume". The latter is likely due to us not utilising temporal information.

4.3 Objective Comparison of Each Algorithm

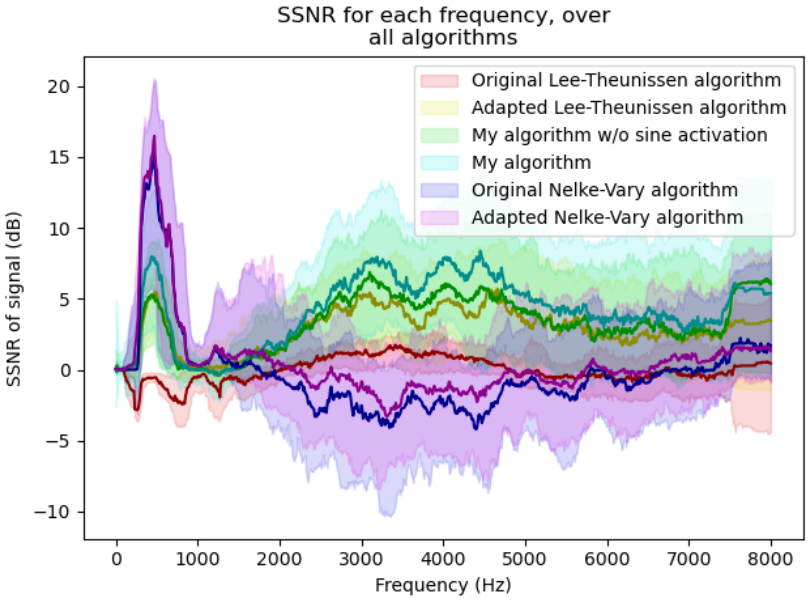


Figure 4.1: The SSNR of each algorithm, averaged over all audio files, as a function of frequency.

Our objective analysis was calculated over a subset of 500 randomly chosen test signals, lasting a total of 59m 28s and varying in length from $\sim$ 1s to 31s. This subset varied in wind type (232 algorithmic wind signals, 268 real wind signals) and wind strength. For each noisy signal $x_S(t)$, I compute the denoised signal $y_{S,A}(t)$ using each algorithm A, comparing each result to the expected denoised signal $\hat{y}_S(t)$ during analysis. For each result, I provide the overall mean and variance to signify the uncertainty of my results. All results are recorded to 3 significant figures, with the best score in each column in bold. Each plotted graph provides both the mean, plus the inter-quartile range as a lighter highlight.

	MSE (V^2)		SSNR (dB)		PESQ narrow-band		PESQ wide-band		STOI		Time to run
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	(500 files)
Nelke-Vary Original	2.88×10^6	9.98×10^{12}	-1.79	**1.1**	1.84	0.21	1.22	0.0469	0.824	0.0137	**48.3 secs**
Nelke-Vary Adapted	2.3×10^6	$\mathbf{9.95 \times 10^{12}}$	2.35	5.45	1.97	0.209	1.33	0.0809	**0.845**	**0.00975**	49.2 secs
Lee-Theunissen Original	2.75×10^6	2.6×10^{13}	4.48	11.0	1.72	**0.176**	1.13	**0.0133**	0.761	0.0156	111mins, 3.76 secs
Lee-Theunissen Adapted	3.53×10^6	4.35×10^{13}	**4.5**	13.2	1.77	0.201	1.15	0.0167	0.789	0.0157	120 mins, 12.4 secs
My Solution w/o sine	2.13×10^6	1.54×10^{13}	3.19	9.5	2.14	0.333	1.54	0.174	**0.845**	0.0137	9 mins, 37.3 secs
My Solution	$\mathbf{2.11 \times 10^6}$	1.76×10^{13}	4.3	14.3	**2.23**	0.332	**1.67**	0.204	0.841	0.0132	9 mins, 38.0 secs

Table 4.4: Main metrics comparing each algorithm presented.

4.3.1 Comparison of Algorithms using a Variety of Metrics

I first present overall results comparing each algorithm across all my metrics.

Analysis of Nelke-Vary Algorithms

The Nelke-Vary algorithm offered a mixture of good and bad results across my metrics (Table 4.4), with inferior results in the SSNR metric. This links back to the subjective feedback – despite the voice itself being clear, the loud artefacts created by the algorithm "drowned out" the clean speech signal, resulting in poor noise removal.

However, it performed well in PESQ narrow-band – I reason about this by examining Figure D.1a and noticing that it performs especially well at attenuating lower frequencies $<$ 2000Hz, then performs poorly up to 8000Hz. Therefore, *PESQ narrow-band may have overestimated its performance by only analysing its performance below 4kHz.* It also performed well in my STOI metric – my hypothesis for its strong performance here is that STOI is a short-term

metric, and variance in volume due to incorrectly-fitted wind gusts (which affects human perception) may not affect STOI due to this short-term analysis, hence causing an overestimation of performance.

My alterations to the Nelke-Vary algorithm offered palpable improvements to performance, namely a reduction in speech attenuation through better wind noise approximation. This directly improves my SSNR result – **at 2.35dB, my signal now contains more speech than noise**.

This algorithm does offer advantages – for example, it takes just 0.014s on average to denoise 1s of audio, indicating its suitability for embedded applications. It also performs well when SSNR is plotted as a function of frequency (Figure 4.1). It beats my other algorithms at removing frequencies < 1500Hz while struggling with higher frequencies and consistency (wide inter-quartile range).

Analysis of Lee-Theunissen Algorithms

The Lee-Theunissen algorithms offered relatively poor performance in Table 4.4, despite scoring well in my human testing (Section 4.2). This may be since *the voice signal was heavily attenuated by the algorithm and wind, causing this attenuation to be treated as noise by my metrics.*

The original Lee-Theunissen algorithm performed noticeably poorly at removing lower-frequency noise (Figure D.1c) – this could indicate that the neural network is unable to fully capture the behaviour of wind noise, making attenuation less robust. Indeed, my adapted algorithm (with more PCA layers) offered improved noise removal in frequencies < 1000Hz, and reduced noise in frequencies > 2500Hz, partially due to a louder speech signal (Table 4.3).

One disadvantage of these algorithms is their speed, each taking > 110 minutes to process 59 minutes of audio. This was due to my bandpass filters taking $\sim 86\%$ of the total time for the algorithm – despite using vectorisation to process all frames at once, we bandpassed using each filter sequentially. Since we also subtract the mean frequency volume across our input signal, this slow runtime means that this algorithm cannot handle real-time processing.

Finally, my original Lee-Theunissen algorithm was trained (by the original researchers) on 16 different voices and multiple types of noise. I instead trained the autoencoder with > 600 distinct voices, meaning that my algorithm likely could not effectively represent individual speakers' characteristics, particularly the harmonic structure of voice, resulting in poor performance. Increasing PCA components enabled us to better capture the harmonic structure of general speech audio, improving noise attenuation.

Analysis of My Solutions

I claim that my solutions seem to offer the best approach for removing wind noise by offering the best performance on average across all of my metrics; and by offering a middle-ground between the Nelke-Vary and Lee-Theunissen algorithms. My final solution **scored highest in the MSE, and both PESQ metrics**, whilst the 'no-sine' version (without the improvement defined in Section 3.11.1) **scored highest using the STOI metric**.

Our SSNR plot (Figure 4.1) shows that **my algorithm performed best for frequencies > 1500Hz**, plus offered reasonable performance for frequencies < 800Hz. However, the Nelke-Vary algorithm came top for frequencies outside this range (800-1500Hz) (due to a harsher noise removal function). The IQR of my solutions' SSNR was also large compared to the other algorithms, resulting in a less consistent output.

Finally, both of my solutions ran relatively quickly, taking $\sim$ 9mins, $\sim$ 40secs to process 59mins of audio, meaning that I could process a signal in real-time without additional latency.

4.3.2 Comparing Algorithms by Word Error Rate

In terms of WER, all algorithms presented offer disappointing results (as shown in Table 4.5), with only the Lee-Theunissen algorithms and my non-sine solution offering comparable performance to the original noisy signal. Conversely, the Nelke-Vary algorithms offer abysmal

	Word Error Rate	
	Mean	Variance
Speech and Wind	0.188	0.0503
Speech Only	0.066	0.0123
Nelke-Vary Original	0.424	0.0759
Nelke-Vary Adapted	0.347	0.0704
Lee-Theunissen Original	**0.191**	0.0502
Lee-Theunissen Adapted	0.199	**0.0501**
My Solution	0.218	0.054
My Solution w/o Sine	0.199	0.0537

Table 4.5: The mean word error rate of each algorithm, plus the corresponding input and optimal output signals.

performance, at $\sim 40\%$ WER on average.

Note that our original noisy signals consist of clean speech added to wind noise – however, *through denoising the signals, I remove components of both noise and speech.* Therefore, as *the speech recognition algorithm has not been trained to deal with artefacts* (that remove components of speech) robustly, *even minor (inaudible) artefacts can greatly affect speech recognition performance.* We could make these speech recognition algorithms more robust by training with noisy data.

4.3.3 Comparison of Algorithms using Spectrogram Analysis

I finally compare each algorithm, plus noisy and clean signals, using spectrograms of an example strong real wind noise audio clip.

Original Noisy and Clean Signals

Our clean signal notably features clear noise-less gaps where speech is not present, particularly in the lower frequencies. Our noisy signal additionally features a clear b/f^a trend of background wind noise matching the theory presented in Section 2.3.3 and Section 3.2. Finally, we note that each speech section is almost as loud as wind gusts (as is demonstrated in Section 3.3.5), hence **making both detection and attenuation very difficult**.

Original Nelke-Vary Algorithm

The original Nelke-Vary algorithm removes the b/f^a trend of wind noise very effectively, resulting in a flat noise floor (Figure 4.2c). However:

- frequencies > 1500Hz are heavily attenuated, affecting the underlying voice signal. *I could avoid fitting wind noise above a threshold frequency to fix this, hence avoiding attenuation of frequencies where wind noise is less prevalent.*
- lower frequencies are poorly attenuated, with background wind noise present for frequencies < 1500Hz – to fix this, we could use a more complex wind fitting algorithm (e.g. using the average of multiple frames to reduce the chance of incorrect wind approximation).

Adapted Nelke-Vary Algorithm

My adaptations provide some clear improvements to the output over the original Nelke-Vary algorithm (Figure 4.2d), trading additional background noise for clearer voice peaks. I note the white vertical line at ~ 5s, caused by a division by zero from us trying to fit a perfectly flat wind PSD to the signal – this is a minor bug I overlooked.

Original Lee-Theunissen Algorithm

As defined in the original paper, the original Lee-Theunissen algorithm [2] offers clear advantages over the Nelke-Vary algorithm (Figure 4.2e). Firstly, speech gains at frequencies > 2000Hz

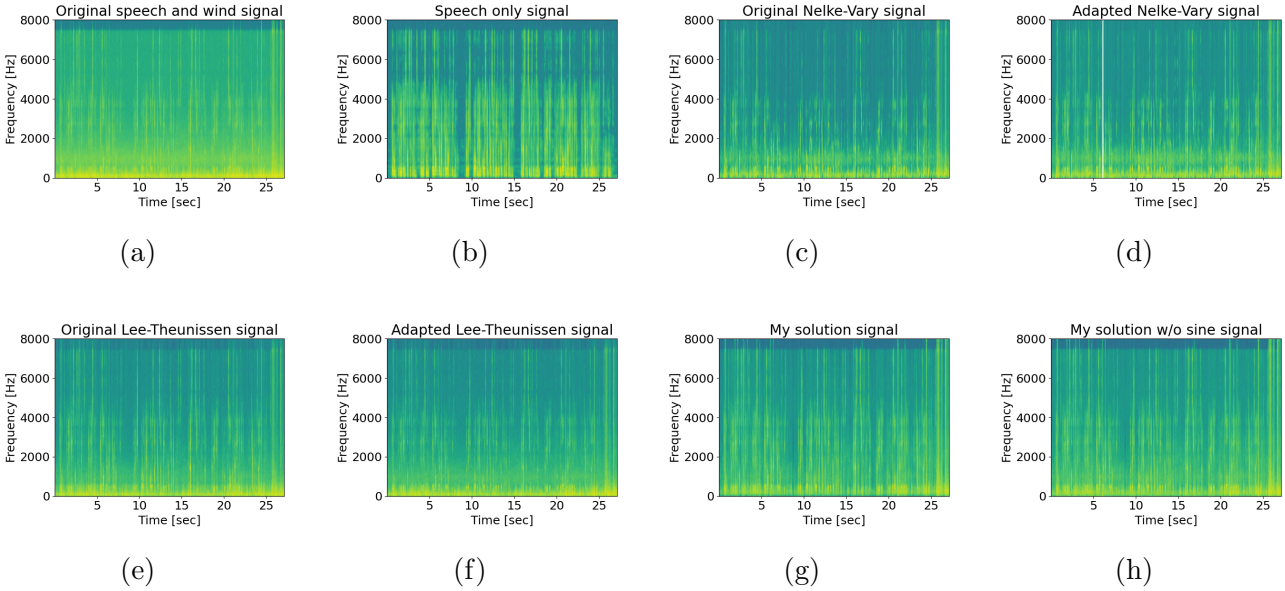


Figure 4.2: A set of spectrograms highlighting the various upsides and downsides of each algorithm. (a) and (b) show the original noisy signal, and the optimal clean speech signal; our other spectrograms show our actual output from denoising using the (c) original and (d) adapted Nelke-Vary algorithms, the (e) original and (f) adapted Lee-Theunissen algorithms, and my solution (g) with and (h) without the sine activation function present.

are much clearer and higher, resulting in much louder, clearer speech. However, performance in lower frequencies (< 500Hz) is poor, with poor attenuation of wind noise.

To improve performance, I could *combine this algorithm with the adapted Nelke-Vary algorithm, using the latter to produce a flat noise floor; then the Lee-Theunissen algorithm (trained on denoised signals) to remove artefacts and remove more granular wind noise.* This should *reconstruct parts of the speech signal attenuated by the Nelke-Vary algorithm.* Alternatively, running this algorithm several times should further attenuate wind noise whilst preserving speech.

Adapted Lee-Theunissen Algorithm

My alterations do offer minor improvements (Figure 4.2f), namely better lower-frequency performance (with clearer gaps between speech peaks for frequencies < 500Hz), at the cost of less noise attenuation – this improves speech coherency whilst reducing wind noise removal.

My Solution

My solution (with the sine activation function applied) offers a balanced solution (as shown in Figure 4.2g), with a flat noise floor, improved attenuation of wind, and gaps in the spectrogram in the absence of speech/wind gusts.

To improve my algorithm further, I could:

- *Incorporate the sine activation function into my output layer*, using gradient clipping or gradient normalisation[58], to eliminate gradient explosion (when the loss function value spikes during training).
- Run my algorithm several times or combine algorithms. For example, *I could first run the Nelke-Vary algorithm on frequencies* $< 1000Hz$ *(where speech is quieter than wind), then run my solution (to more selectively remove noise whilst preserving speech* – see Figure 4.1).
- Finally, *I could operate on multiple frames at once to exploit temporal information* to better distinguish speech and wind (wind gusts are relatively short term phenomena compared to speech [1], hence they can be distinguished using their duration).

My solution without sine activation

Without the sine activation function, I note that the spectrogram (Figure 4.2h) is relatively flat, with quiet voice peaks and relatively loud wind noise. However, it does attenuate wind noise whilst preserving the speech signal more accurately than the other algorithms.

4.4 Analysing the Effect of Hyperparameter Values and NN Architectures on Each Algorithm

4.4.1 Optimising Hyperparameters

An important consideration is how hyperparameters are inextricably linked. For example, consider the equation for $\alpha(\lambda)$:

$$\alpha(\lambda) = \begin{cases} \alpha_{min}, & \text{if } SSC(\lambda) < f_1 \\ \alpha_{min} + \frac{(SSC(\lambda)-f_1)\cdot(\alpha_{max}-\alpha_{min})}{f_2-f_1}, & \text{if } f_1 \leq SSC(\lambda) \leq f_2 \\ \alpha_{max}, & \text{if } SSC(\lambda) > f_2 \end{cases}$$

Adjusting any one parameter in this equation will affect the value of $\alpha(\lambda)$ – therefore, I cannot simply optimise hyperparameters independently of each other. Instead, to optimise my solution to maximise a particular metric X, I need to test combinations of hyperparameter values using a method like random search or grid search [63]. Due to the time needed to test all parameter values using grid search for each algorithm, I instead test parameters individually and only edit values which significantly benefit the algorithm's performance.

4.4.2 Analysis of the Hyperparameters of the Original Nelke-Vary Algorithm

I now provide analyses for each hyperparameter (enumerated in Section 3.5) used in my Nelke-Vary implementation, with the values used in the original Nelke-Vary algorithm in grey. Less insightful analyses have been moved to the appendix due to brevity.

Exploration of Hyperparameter Values for `f1`

	MSE (V^2)		SSNR (dB)		PESQ narrow-band		PESQ wide-band		STOI	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
0	4.16×10^6	4.65×10^{13}	4.04	11.2	**1.87**	0.235	**1.15**	0.0185	**0.803**	0.0145
100	4.25×10^6	4.69×10^{13}	3.96	**10.9**	**1.87**	0.235	**1.15**	0.018	0.802	**0.0143**
200	4.09×10^6	4.66×10^{13}	4.06	11.2	1.84	0.22	**1.15**	0.017	0.799	0.0144
250	3.53×10^6	4.35×10^{13}	4.5	13.2	1.77	0.201	**1.15**	0.0167	0.789	0.0157
300	3.1×10^6	4.02×10^{13}	**4.71**	14.8	1.71	0.18	**1.15**	0.0161	0.773	0.0178
400	2.73×10^6	3.19×10^{13}	4.2	17.7	1.62	0.144	**1.15**	0.0133	0.736	0.0203
500	$\mathbf{2.67 \times 10^6}$	2.45×10^{13}	2.86	20.2	1.53	0.101	1.13	0.00938	0.694	0.0197
600	2.78×10^6	$\mathbf{1.89 \times 10^{13}}$	1.03	21.9	1.45	**0.0635**	1.11	**0.00582**	0.653	0.0167

Table 4.6: Metrics showing the different hyperparameter values for `f1`, for the Nelke-Vary algorithm.

I initially set the value for `f1` at 250Hz – however, frequencies above and below this perform better for different metrics. For example, MSE indicates that 500Hz is the optimal frequency for `f1`; whilst STOI instead offers 0Hz as a much better option – this is because my metrics prioritise either the accuracy of my output, the quality of the speech signal, or the signal to noise error. Larger values of `f1` remove all noise above a higher frequency, resulting in more noise and voice attenuation; whilst lower values preserve more speech at the expense of additional noise. Therefore, I do not adjust my value of `f1`, since 250Hz performs relatively well in all metrics.

Exploration of Hyperparameter Values for Window Size

	MSE (V^2)		SSNR (dB)		PESQ narrow-band		PESQ wide-band		STOI	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
6	1.08×10^7	2.57×10^{14}	3.03	**8.01**	**1.85**	0.241	1.18	0.0331	**0.82**	**0.0151**
7	7.4×10^6	1.33×10^{14}	3.46	9.58	**1.85**	0.247	**1.19**	0.0353	0.794	0.0163
8	3.97×10^6	5.09×10^{13}	4.0	11.6	1.79	0.207	1.16	0.0185	0.779	0.0156
9	3.53×10^6	4.35×10^{13}	4.5	13.2	1.77	0.201	1.15	**0.0167**	0.789	0.0157
10	$\mathbf{3.28 \times 10^6}$	4.23×10^{13}	**4.94**	14.7	1.78	0.21	1.16	0.0204	0.792	0.0162
11	3.29×10^6	4.18×10^{13}	4.71	14.7	1.8	0.211	1.17	0.0244	0.778	0.0174
12	3.37×10^6	$\mathbf{4.08 \times 10^{13}}$	4.3	14.3	1.77	**0.192**	1.17	0.0234	0.76	0.0165

Table 4.7: Metrics showing the different hyperparameter values for window size (in the form 2^n), for the Nelke-Vary algorithm.

Window size causes a large effect on my output signal. Larger windows result in much larger FFTs, allowing for more accurate wind noise fits in the frequency domain and better speech preservation, at the cost of reduced temporal resolution and reduced ability to respond to short-term phenomena such as turbules. However, smaller windows result in more numerous, smaller FFTs, capturing and removing short-lasting turbules from single frames, at the cost of reduced frequency resolution and noticeable frequency artefacting due to inaccurate wind noise fits.

Our data (Table 4.7) shows different window sizes appealing to different metrics, with shorter windows offering better noise reduction (and hence better PESQ and STOI scores) at the cost of speech artefacts (which harm MSE and SSNR scores); and vice versa. The Nelke-Vary paper specifies a 0.5s = $2^{\sim 9.64}$sample window – however, my original implementation approximates this with a 2^9 sample = 0.32s window, which offers comparable performance to the optimal-sized window (0.64s).

Exploration of Hyperparameter Values for `aMin` and aMax

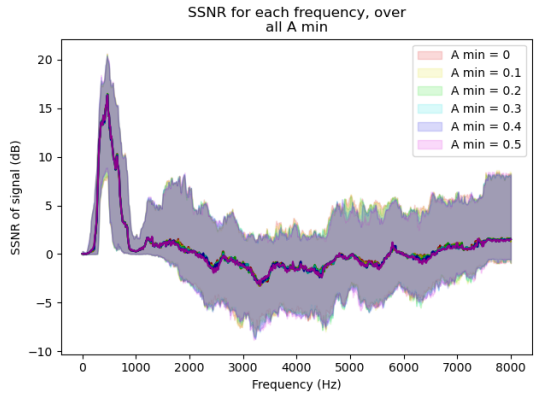
	MSE (V^2)		SSNR (dB)		PESQ narrow-band		PESQ wide-band		STOI	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
0.5	3.78×10^6	5.06×10^{13}	4.26	**11.5**	1.76	0.198	1.14	**0.0141**	**0.791**	**0.015**
0.6	3.72×10^6	4.89×10^{13}	4.32	11.8	1.76	0.199	1.14	0.0146	**0.791**	0.0151
0.7	3.65×10^6	4.71×10^{13}	4.38	12.2	**1.77**	0.2	**1.15**	0.0154	**0.791**	0.0153
0.8	3.59×10^6	4.54×10^{13}	4.44	12.7	**1.77**	0.201	**1.15**	0.016	**0.791**	0.0155
0.9	3.53×10^6	4.35×10^{13}	4.5	13.2	**1.77**	0.201	**1.15**	0.0167	0.789	0.0157
1	$\mathbf{3.47 \times 10^6}$	$\mathbf{4.14 \times 10^{13}}$	**4.54**	13.6	**1.77**	**0.197**	**1.15**	0.0163	0.786	0.0156

Table 4.8: Metrics showing the different hyperparameter values for aMax for the Nelke-Vary algorithm.

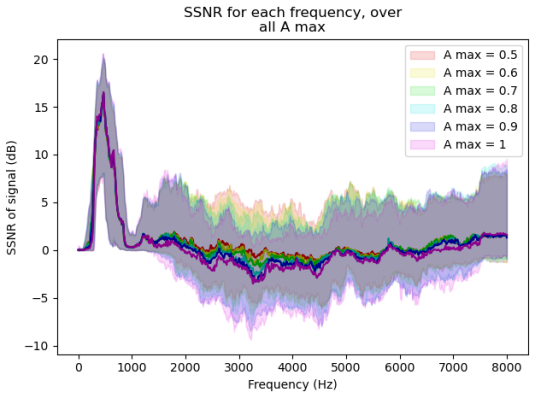
	MSE (V^2)		SSNR (dB)		PESQ narrow-band		PESQ wide-band		STOI	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
0	3.59×10^6	4.4×10^{13}	4.41	**12.8**	1.62	**0.172**	1.12	**0.0121**	0.785	0.0159
0.1	3.53×10^6	4.35×10^{13}	4.5	13.2	1.77	0.201	1.15	0.0167	0.789	0.0157
0.2	3.49×10^6	$\mathbf{4.34 \times 10^{13}}$	4.57	13.4	1.82	0.216	1.16	0.0186	0.793	0.0155
0.3	$\mathbf{3.48 \times 10^6}$	4.35×10^{13}	4.62	13.5	1.84	0.223	1.16	0.0195	0.797	0.0153
0.4	$\mathbf{3.48 \times 10^6}$	4.36×10^{13}	4.66	13.6	**1.85**	0.227	1.16	0.0203	0.8	0.0153
0.5	3.49×10^6	4.39×10^{13}	**4.68**	13.7	**1.85**	0.229	**1.17**	0.0208	**0.802**	**0.0152**

Table 4.9: Metrics showing the different hyperparameter values for `aMin` for the Nelke-Vary algorithm.

My results for `aMin` and `aMax` are unexpected – I see that larger values are preferable over smaller values regardless of the metric. This is understandable for `aMax` – for speech, I want to



(a) The segmented SSNR for various values of `aMin`.



(b) The segmented SSNR for various values of `aMax`.

Figure 4.3: The SSNR for my values of `aMin` and `aMax`

smooth the PSD as much as possible to counteract the detrimental effects of misclassification of speech as wind. This exponential smoothing does not affect speech since speech is a relatively long-term phenomenon compared to wind. However, my result for `aMin` would indicate that smaller values are too eager at removing wind noise or often misfit my wind noise. Hence, smoothing for `aMin` preserves speech more than it affects wind noise removal. This would indicate that even the adapted wind fitting algorithm used to test cannot always remove wind noise effectively whilst leaving speech intact.

Our SSNR plots (Figure 4.3) conveys the opposite – the value for `aMin` does not seem to affect my plot significantly, but `aMax` does seem to vary significantly by frequency. Indeed, lower values of `aMax` seem to better retain speech. This would convey the opposite of my other explanation – that my wind noise removal works effectively at `aMin` $= 0.1$, and that `aMax` can be set lower since speech misclassification is unlikely. Of course, this assumes that frequencies are equally likely to appear, despite the b/f^a trend of wind noise.

Finally, I note that all hyperparameter values give similar results – hence I will stick to the values proposed in the original paper [1].

Exploration of my Adjusted Weighting Algorithm

The only difference between the original and adapted Nelke-Vary algorithms is that the latter uses the adjusted weighting algorithm discussed in Section 3.6.1. Therefore, I omit this analysis and direct the reader to Sections D.2 and 4.3.

4.4.3 Analysis of the Hyperparameters of the Original Lee-Theunissen Algorithm

I now provide analyses for each hyperparameter (enumerated in Section 3.8), with less insightful analyses moved to the appendix. Rows in dark grey correspond to hyperparameter values used in my adapted algorithm; rows in grey are used in my original Lee-Theunissen algorithm, or both Lee-Theunissen algorithms if no dark grey column is present.

I split my analysis into two sections. For parameter changes that do not need the neural network, I bypass the network entirely and use my optimal gains as the predicted gains. This is due to the training speed of this algorithm, which is prohibitively slow to perform for every evaluation hyperparameter value. For parameters that directly change the neural network's architecture, we use predicted gains from the autoencoder output.

Lee-Theunissen Algorithm – Analysis Bypassing the NN

Exploration of Hyperparameter Values for `sizeOfWindowSecs`

	MSE (V^2)		SSNR (dB)		PESQ narrow-band		PESQ wide-band		STOI	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
0.005	3.58×10^6	8.53×10^{12}	-18.8	1.97	1.67	**0.1**	1.17	**0.0186**	0.83	0.0036
0.01	1.94×10^6	3.27×10^{12}	-3.97	**0.916**	2.59	0.251	1.47	0.12	0.921	0.00108
0.02	$\mathbf{6.88 \times 10^5}$	$\mathbf{1.53 \times 10^{12}}$	7.39	8.53	3.11	0.411	1.99	0.326	0.95	**0.000751**
0.05	$\mathbf{6.88 \times 10^5}$	2.0×10^{12}	**10.3**	12.0	**3.31**	0.397	**2.16**	0.369	**0.954**	0.00086
0.1	8.1×10^5	2.35×10^{12}	9.77	8.61	3.26	0.368	2.12	0.344	0.952	0.000914
0.2	8.5×10^5	2.45×10^{12}	9.6	7.92	3.22	0.354	2.1	0.331	0.952	0.000914

Table 4.10: All main metrics for the size of each time-domain window, for the Lee-Theunissen algorithm.

As shown in Table 4.10, choosing larger time-domain windows offers better performance on average, with windows of size 0.02s and 0.05s scoring highest in my metrics. As mentioned by my participants, smaller time-domain windows exhibit robotic-sounding artefacts, which may be due to the smaller windows creating more noticeable artefacts due to my reduced frequency resolution. Larger time-domain windows seem to amplify speech, which may explain their poor performance, especially for MSE and SSNR, where amplified speech will be treated as noise. By attenuating these signals, I should see improved performance in my metrics.

Exploration of the Number of Overlapping Windows

	MSE (V^2)		SSNR (dB)		PESQ narrow-band		PESQ wide-band		STOI	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
1 (32Hz)	4.2×10^6	1.14×10^{13}	-25.3	**0.651**	1.06	**0.000583**	1.03	$\mathbf{8.75 \times 10^{-5}}$	0.448	0.00186
2 (64Hz)	3.28×10^6	7.38×10^{12}	-15.1	1.01	2.19	0.189	1.36	0.0687	0.902	0.00126
4 (128Hz)	$\mathbf{6.88 \times 10^5}$	$\mathbf{1.53 \times 10^{12}}$	7.39	8.53	**3.11**	0.411	**1.99**	0.326	**0.95**	**0.000751**
8 (256Hz)	8.26×10^5	2.49×10^{12}	**9.25**	11.1	3.06	0.399	1.85	0.301	0.94	0.00126

Table 4.11: All main metrics for the number of overlapping windows (and size of each window), for the Lee-Theunissen algorithm.

As shown above, 4 overlapping windows offer a good compromise between a flat frequency spectrum (once the denoised signal is reassembled) and selective noise removal. However, my 8-overlap solution produces a louder voice output, explaining its poorer performance here (since amplification is treated as noise in these metrics).

Lee-Theunissen Algorithm – Analysis Utilising the NN

Exploration of the Number of PCA Components for the Original NN Architecture

	MSE (V^2)		SSNR (dB)		PESQ narrow-band		PESQ wide-band		STOI	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
PCA50	3.36×10^6	1.83×10^{13}	-1.85	1.08	1.83	0.23	1.23	**0.0641**	0.814	0.0142
PCA100	3.34×10^6	1.83×10^{13}	-1.66	1.02	1.84	0.235	1.23	0.0655	0.817	0.0139
PCA200	3.28×10^6	1.82×10^{13}	-1.26	**0.985**	1.85	**0.227**	1.25	0.0695	0.819	0.0133
PCA500	2.93×10^6	$\mathbf{1.64 \times 10^{13}}$	0.00761	1.89	1.91	0.233	1.3	0.0868	0.828	0.0119
PCA1000	$\mathbf{2.73 \times 10^6}$	1.69×10^{13}	**2.14**	5.07	**1.97**	0.245	**1.34**	0.103	**0.839**	**0.00971**

Table 4.12: Comparing my original PCA architecture autoencoders using all main metrics for the Lee-Theunissen algorithm. The number in the name indicates the number of PCA components used (each from speech and wind) for the input of that autoencoder.

As shown in Table 4.12, as the number of PCA components increases, the performance of my algorithm also increases – this is since the input data can be more accurately represented within

the middle layer, allowing more complex patterns to be extracted and hence better denoising gains derived. This only works up to a point – adding too many PCA components may cause the network to overfit the data, causing a reduction in accuracy.

Chapter 5

Conclusions

5.1 Achievements and Progress

I feel that my project was a success. I have completed all core success criteria, including implementing two existing wind noise algorithms, making novel improvements to each, evaluating/comparing their performance, and linking this performance back to wind noise theory. In addition, I proposed and created a novel wind-noise removal algorithm that offers state-of-the-art performance, removing noise both faster and more effectively, and with a ~ 0.3 improvement in mean MOS scores over the other presented algorithms (Appendix D.1). Two of these algorithms also run in real-time, with both Nelke-Vary and my solution processing a 7s signal with < 2s latency.

I have also overcome unforeseen difficulties and challenges with my project – notably creating algorithms to generate wind noise from otherwise polluted datasets and removing clicks from collected wind noise (due to amplitude overflows/underflows) with high accuracy.

5.2 Challenges Faced and Lessons Learned

Firstly, my original project timeline was incredibly optimistic, predicting that the core deliverables would be completed by January and extensions/writeup completed by April. Unfortunately, this was not achievable, resulting in some extension tasks not being completed. I have therefore learned how to manage time more effectively, including multitasking, setting realistic goals and learning independent research skills. Secondly, I experienced considerable issues with 'clicks' in my wind noise; this has taught me to test equipment and explore alternatives (both in hardware and software) when a problem arises. Finally, I had major difficulty training my neural network, both in terms of time and storage (our dataset was 20GB and took $\sim$ 10hrs to recreate) – this was rectified by preprocessing and storing my processed dataset, which reaffirmed the importance of memoisation in improving performance.

5.3 Future Work

Given the promising performance of my solution, I propose the following future work:

- Firstly, I could combine the strong performance of the Nelke-Vary algorithm at lower frequencies < 1000Hz, with the better selectivity of my solution above 1000Hz. This should greatly improve SSNR compared to the algorithms presented here, through combining more eager noise removal (via an exponential fit) for wind-heavy frequencies with more selective removal (via an autoencoder) for speech-heavy frequencies.
- Secondly, I could improve my solution by exploiting spectro-temporal patterns in the data. By passing multiple frequency-domain PSD frames into the neural network at once or incorporating memory units (e.g. LSTM [64]) into the autoencoder, I should be able to produce gains that offer improved denoising performance through being able to detect the duration of certain phenomena, aiding speech-wind categorisation.
- I could focus on improving speech recognition performance in wind further by training a network specifically to recognise speech denoised by a given algorithm.
- Finally, I could generalise my algorithms to remove wind from mixed signals, including speech and environmental noises.

Bibliography

[1] Christoph Matthias Nelke et al. "Single Microphone Wind Noise PSD Estimation Using Signal Centroids". In: *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. May 2014. URL: http://ikspub.iks.rwth-aachen.de/pdfs/nelke14.pdf (visited on 12/30/2020).

[2] Tyler Lee and Frédéric Theunissen. "A single microphone noise reduction algorithm based on the detection and reconstruction of spectro-temporal features". In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 471.2184 (2015), p. 20150309. DOI: 10.1098/rspa.2015.0309. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rspa.2015.0309. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rspa.2015.0309 (visited on 12/30/2020).

[3] Chris Woolf. "Characterization and Measurement of Wind Noise around Microphones". In: *Audio Engineering Society Convention 140*. May 2016. URL: http://www.aes.org/e-lib/browse.cfm?elib=18194 (visited on 12/30/2020).

[4] J. A. Zakis. "Wind noise at microphones within and across hearing aids at wind speeds below and above microphone saturation". In: *J Acoust Soc Am* 129.6 (May 2011), pp. 3897–3907. DOI: 10.1121/1.3578453. URL: https://pubmed.ncbi.nlm.nih.gov/21682412/ (visited on 03/21/2021).

[5] George H. Goedecke and Harry J. Auvermann. "Acoustic scattering by atmospheric turbules". In: *The Journal of the Acoustical Society of America* 102.2 (1997), pp. 759–771. DOI: 10.1121/1.419951. eprint: https://doi.org/10.1121/1.419951. URL: https://doi.org/10.1121/1.419951.

[6] D. A. de Wolf. "A random-motion model of fluctuations in a nearly transparent medium". In: *Radio Science* 18.2 (1983), pp. 138–142. DOI: https://doi.org/10.1029/RS018i002p00138. eprint: https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/RS018i002p00138. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/RS018i002p00138.

[7] Sheng Shen et al. "MUTE: bringing IoT to noise cancellation". In: Aug. 2018, pp. 282–296. DOI: 10.1145/3230543.3230550. URL: https://synrg.csl.illinois.edu/papers/mute-sigcomm18.pdf (visited on 03/21/2021).

[8] Sony. *What is Noise-Cancellation and what can I expect?* 2020. URL: https://www.sony.co.uk/electronics/support/articles/00203389 (visited on 03/25/2021).

[9] Bahar Khalighinejad et al. "Adaptation of the human auditory cortex to changing background noise". In: *Nature Communications* 10.1 (June 2019), p. 2509. ISSN: 2041-1723. DOI: 10.1038/s41467-019-10611-4. URL: https://doi.org/10.1038/s41467-019-10611-4.

[10] Kristoffer Walker and Michael Hedlin. "A Review of Wind-Noise Reduction Methodologies". In: Jan. 2010, pp. 141–182. ISBN: 978-1-4020-9507-8. DOI: 10.1007/978-1-4020-9508-5.

[11] FAA. *Hearing and Noise in Aviation*. URL: https://www.faa.gov/pilots/safety/pilotsafetybrochures/media/hearing.pdf (visited on 10/16/2020).

[12] Yujia Wang et al. “Toward Automatic Audio Description Generation for Accessible Videos”. In: May 2021. DOI: 10.1145/3411764.3445347. URL: https://dingzeyu.li/files/automatic-accessible-audio-descrition-chi-2021-wang-et-al.pdf (visited on 03/24/2021).

[13] George Hessler et al. “Experimental study to determine wind-induced noise and windscreen attenuation effects on microphone response for environmental wind turbine and other applications”. In: *Noise Control Engineering Journal* 56 (July 2008). DOI: 10.3397/1.2949926. (Visited on 03/21/2021).

[14] Minajul Haque and Kaustubh Bhattacharyya. “Speech Background Noise Removal Using Different Linear Filtering Techniques”. In: Jan. 2018, pp. 297–307. ISBN: 978-981-10-8239-9. DOI: 10.1007/978-981-10-8240-5_33. URL: https://www.researchgate.net/publication/325622133_Speech_Background_Noise_Removal_Using_Different_Linear_Filtering_Techniques/link/5b507b2caca27217ffa3c155/download (visited on 03/21/2021).

[15] krisp. *Krisp – #1 Noise Cancelling App*. URL: https://krisp.ai/ (visited on 05/11/2021).

[16] Davit Baghdasaryan. *Real-Time Noise Suppression Using Deep Learning*. Oct. 2018. URL: https://developer.nvidia.com/blog/nvidia-real-time-noise-suppression-deep-learning/ (visited on 10/16/2020).

[17] Christoph Matthias Nelke and Peter Vary. “Dual Microphone Wind Noise Reduction by Exploiting the Complex Coherence”. In: *ITG-Fachtagung Sprachkommunikation* (Sept. 2014). (Visited on 03/21/2021).

[18] Anthony Rhodes. “Real-Time Wind Noise Detection and Suppression with Neural-Based Signal Reconstruction for Multi-Channel, Low-Power Devices”. In: (Sept. 2017). URL: https://arxiv.org/ftp/arxiv/papers/1710/1710.00082.pdf (visited on 12/30/2020).

[19] Elias Nemer and Wilf Leblanc. “Single-microphone wind noise reduction by adaptive postfiltering”. In: Nov. 2009, pp. 177–180. DOI: 10.1109/ASPAA.2009.5346518. (Visited on 03/21/2021).

[20] Vassil Panayotov et al. “Librispeech: An ASR corpus based on public domain audio books”. In: Apr. 2015, pp. 5206–5210. DOI: 10.1109/ICASSP.2015.7178964. (Visited on 03/21/2021).

[21] Daniel Povey. *OpenSLR - LibriSpeech ASR corpus*. URL: http://www.openslr.org/12 (visited on 12/29/2020).

[22] SciPy developers. *SciPy.org¶*. URL: https://www.scipy.org/ (visited on 12/30/2020).

[23] Google. *TensorFlow*. URL: https://www.tensorflow.org/ (visited on 12/30/2020).

[24] Wikimedia Commons. *File:Spiral model (Boehm, 1988).svg — Wikimedia Commons, the free media repository*. 2020. URL: https://commons.wikimedia.org/w/index.php?title=File:Spiral_model_(Boehm,_1988).svg&oldid=480509142 (visited on 03/25/2021).

[25] A. Alshamrani, A. Bahattab, and I. Fulton. “A Comparison Between Three SDLC Models Waterfall Model, Spiral Model, and Incremental/Iterative Model”. In: 2015. URL: https://www.ijcsi.org/papers/IJCSI-12-1-1-106-111.pdf (visited on 03/21/2021).

[26] Simran Kaur Arora. *PyTorch vs TensorFlow: Difference you need to know*. 2020. URL: https://hackr.io/blog/pytorch-vs-tensorflow#:~:text=Tensorflow%5C%20works%5C%20on%5C%20a%5C%20static,nature%5C%20of%5C%20creating%5C%20the%5C%20graphs. (visited on 03/21/2021).

[27] A. W. Rix et al. “Perceptual evaluation of speech quality (PESQ)-a new method for speech quality assessment of telephone networks and codecs”. In: *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No.01CH37221)*. Vol. 2. 2001, 749–752 vol.2. DOI: 10.1109/ICASSP.2001.941023. (Visited on 03/21/2021).

[28] Cees Taal et al. “A short-time objective intelligibility measure for time-frequency weighted noisy speech”. In: Apr. 2010, pp. 4214–4217. DOI: 10.1109/ICASSP.2010.5495701.

[29] Enthought Inc. *SciPy BSD-3 License*. URL: https://github.com/scipy/scipy/blob/master/LICENSE.txt (visited on 03/24/2021).

[30] The TensorFlow Authors. *Tensorflow Apache License*. URL: https://github.com/tensorflow/tensorflow/blob/master/LICENSE (visited on 03/24/2021).

[31] ludlows. *python-pesq License*. URL: https://github.com/ludlows/python-pesq/blob/master/LICENSE (visited on 03/25/2021).

[32] OPTICOM GmbH. Psytechnics Ltd. *PESQ License and Evaluation Code*. URL: https://github.com/ludlows/python-pesq/blob/master/pesq/pesqdsp.c (visited on 03/25/2021).

[33] Douglas Lyon. “The Discrete Fourier Transform, Part 4: Spectral Leakage.” In: *Journal of Object Technology* 8 (Nov. 2009), pp. 23–34. DOI: 10.5381/jot.2009.8.7.c2.

[34] Wikimedia Commons. *File:Spectral leakage caused by windowing.svg — Wikimedia Commons, the free media repository*. 2020. URL: https://commons.wikimedia.org/w/index.php?title=File:Spectral_leakage_caused_by_%5C%22windowing%5C%22.svg&oldid=479800875 (visited on 03/25/2021).

[35] Robert Bristow-Johnson. “Equal power crossfade”. May 2020. URL: https://dsp.stackexchange.com/a/49989 (visited on 03/22/2021).

[36] M Johansson. “The Hilbert Transform”. Växkö University. URL: http://www.fuchs-braun.com/media/d9140c7b3d5004fbffff8007fffffff0.pdf (visited on 04/20/2021).

[37] S.G. Braun et al. *Encyclopedia of Vibration: F-P*. Encyclopedia of Vibration. Academic Press, 2002. ISBN: 9780122270857. URL: https://books.google.co.uk/books?id=5uJUAAAAMAAJ.

[38] Mark A. Kramer. “Nonlinear principal component analysis using autoassociative neural networks”. In: *AIChE Journal* 37.2 (1991), pp. 233–243. DOI: https://doi.org/10.1002/aic.690370209. eprint: https://aiche.onlinelibrary.wiley.com/doi/pdf/10.1002/aic.690370209. URL: https://aiche.onlinelibrary.wiley.com/doi/abs/10.1002/aic.690370209.

[39] Wikimedia Commons. *File:PCA vs Linear Autoencoder.png — Wikimedia Commons, the free media repository*. 2020. URL: https://commons.wikimedia.org/w/index.php?title=File:PCA_vs_Linear_Autoencoder.png&oldid=441638145 (visited on 03/25/2021).

[40] Wikimedia Commons. *File:Window function and its Fourier transform – Hann (n = 0...N).svg — Wikimedia Commons, the free media repository*. 2020. URL: https://commons.wikimedia.org/w/index.php?%20title=File:Window_function_and_its_Fourier_transform_%5C%E2%5C%80%5C%93_Hann_(n_%5C%3D_0...N).svg&oldid=518579190 (visited on 03/25/2021).

[41] John Daugman. “Computer Vision - Slides”. 2021. URL: https://www.cl.cam.ac.uk/teaching/2021/CompVision/CompVisLectureNotes2021.pdf (visited on 03/26/2021).

[42] King Chung. "Comparisons of spectral characteristics of wind noise between omnidirectional and directional microphones". In: *The Journal of the Acoustical Society of America* 131 (June 2012), pp. 4508–17. DOI: 10.1121/1.3699216.

[43] H.A. Panofsky and J.A. Dutton. *Atmospheric Turbulence: Models and Methods for Engineering Applications*. Wiley, 1984. ISBN: 9780471057147. DOI: 10.2514/3.48643. URL: https://arc.aiaa.org/doi/abs/10.2514/3.48643?journalCode=aiaaj.

[44] P. MOIN. "Revisiting Taylor's hypothesis". In: *Journal of Fluid Mechanics* 640 (2009), pp. 1–4. DOI: 10.1017/S0022112009992126.

[45] Stuart Bradley et al. "The Mechanisms Creating Wind Noise in Microphones". In: *Audio Engineering Society Convention 114*. Mar. 2003. URL: http://www.aes.org/e-lib/browse.cfm?elib=12547 (visited on 12/30/2020).

[46] A.S. Monin, A.M. I?A?glom, and J.L. Lumley. *Statistical Fluid Mechanics: Mechanics of Turbulence*. Dover books on physics v. 1. Dover Publications, 2007. ISBN: 9780486458830. URL: https://books.google.co.uk/books?id=EtTyyI4%5C_DvIC.

[47] Richard Raspet, Jeremy Webster, and Kevin Dillion. "Framework for wind noise studies". In: *The Journal of the Acoustical Society of America* 119 (Feb. 2006), pp. 834–843. DOI: 10.1121/1.2146113.

[48] William K. George, Paul D. Beuther, and Roger E. A. Arndt. "Pressure spectra in turbulent free shear flows". In: *Journal of Fluid Mechanics* 148 (1984), pp. 155–191. DOI: 10.1017/S0022112084002299.

[49] Stuart Bradley et al. "The mechanisms creating wind noise in microphones". In: *Journal of the Audio Engineering Society* (Mar. 2003).

[50] *Implementing PCA with Numpy*. 2019. URL: https://stackoverflow.com/q/58666635 (visited on 04/13/2021).

[51] Zakaria Jaadi. *A Step-by-Step Explanation of Principal Component Analysis*. 2019. URL: https://builtin.com/data-science/step-step-explanation-principal-component-analysis (visited on 03/26/2021).

[52] Kenneth Stevens. "Acoustic Phonetics". In: vol. 109. Jan. 2000, pp. 607–607. DOI: 10.1121/1.1327577.

[53] Daniel Fogerty and Larry E. Humes. "The role of vowel and consonant fundamental frequency, envelope, and temporal fine structure cues to the intelligibility of words and sentences". In: *The Journal of the Acoustical Society of America* 131.2 (2012), pp. 1490–1501. DOI: 10.1121/1.3676696. eprint: https://doi.org/10.1121/1.3676696. URL: https://doi.org/10.1121/1.3676696.

[54] Saint-Gobain Ecophon. *Generating and understanding speech*. URL: https://www.ecophon.com/en/about-ecophon/acoustic-knowledge/basic-acoustics/generating-and-understanding-speech/ (visited on 05/04/2021).

[55] David Klein, Didier Depireux, and Shihab Shamma. "Stimulus-invariant processing and spectrotemporal reverse correlation in primary auditory cortex". In: *Journal of computational neuroscience* 20 (May 2006), pp. 111–36. DOI: 10.1007/s10827-005-3589-4.

[56] G. Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of Control, Signals and Systems* 2 (1989), pp. 303–314.

[57] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org (visited on 05/04/2021).

[58] Keras documentation. *Optimizers*. URL: https://keras.io/api/optimizers/ (visited on 04/28/2021).

[59] Jingdong Li. *python-pesq*. URL: https://github.com/vBaiCai/python-pesq (visited on 02/04/2021).

[60] Morten Kolbaek, Zheng-Hua Tan, and Jesper Jensen. "On the Relationship Between Short-Time Objective Intelligibility and Short-Time Spectral-Amplitude Mean-Square Error for Speech Enhancement". In: *IEEE/ACM Transactions on Audio, Speech, and Language Processing* 27.2 (Feb. 2019), pp. 283–295. ISSN: 2329-9304. DOI: 10.1109/taslp.2018.2877909. URL: http://dx.doi.org/10.1109/TASLP.2018.2877909.

[61] Awni Y. Hannun et al. "Deep Speech: Scaling up end-to-end speech recognition". In: *CoRR* abs/1412.5567 (2014). arXiv: 1412.5567. URL: http://arxiv.org/abs/1412.5567 (visited on 03/21/2021).

[62] Mozilla. *mozilla/DeepSpeech*. URL: https://github.com/mozilla/DeepSpeech (visited on 02/04/2021).

[63] Marc Claesen and Bart De Moor. *Hyperparameter Search in Machine Learning*. 2015. arXiv: 1502.02127 [cs.LG].

[64] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[65] Markus Kuhn. "Digital Signal Processing - Slides". 2020. URL: https://www.cl.cam.ac.uk/teaching/2021/DSP/dsp-slides.pdf (visited on 03/21/2021).

[66] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. USA: Prentice Hall Press, 2009. ISBN: 0136042597. DOI: 10.5555/1671238.

Appendix A

Additional Theory

A.1 Digital Signal Processing

A.1.1 Fourier Transform

The Fourier transform is a mathematical transform which decomposes time-domain signals into sinusoids, whose frequencies lie in the range $[0, \frac{f_s}{2}]$ (where f_s is the sampling frequency of the signal). Using the complex magnitude of each sinusoid frequency, the signal can be decomposed from a time-domain function to a frequency-domain function. This transformation is especially useful in spectral weighting (Appendix A.1.2), since it enables us to attenuate frequencies in the signal directly based on its amplitude.

The Fourier transform for a signal $g(t)$ is given as: [65]:

$$\mathcal{F}\{g(t)\}(f) = G(f) = \int_{-\infty}^{\infty} g(t) \cdot e^{-2\pi i f t}\, dt$$

with its inverse given as:

$$\mathcal{F}^{-1}\{G(f)\}(t) = g(t) = \int_{-\infty}^{\infty} G(f) \cdot e^{2\pi i f t}\, df$$

Since I are dealing with real-valued discrete audio signals, I can define the real-valued discrete Fourier transform (DFT) as a special case of the DFT. Given a signal $g(t)$ of length N and sample rate f_s the real DFT is given as:

$$\mathcal{F}(g(t))(f) = G(f) = \sum_{k=0}^{\lceil \frac{N}{2} \rceil} g(t) \cdot e^{-2\pi i \frac{f}{f_s} k}$$

$$\mathcal{F}^{-1}(G(f))(t) = g(t) = \frac{1}{N} \sum_{k=0}^{\lceil \frac{N}{2} \rceil} G(f) \cdot e^{2\pi i \frac{f}{f_s} k}$$

Note that, due to the Hermitian symmetry of the real-valued DFT:

$$G^*(f) = G(-f)$$

I can discard all negative frequencies, halving the number of terms, yet still fully reconstruct the original signal.

A.1.2 Spectral Weighting and Filter Banks

Spectral weighting is the process of selectively weighing frequency bands in a signal. By identifying the presence and strength of noise in the frequency domain, I can then use the proportion of noise to signal to approximate gains, which would then attenuate the noise in my signal.

Two methods are commonly used for spectral weighting. Firstly, the time-domain method (used in the Lee-Theunissen algorithm [2]) utilise a filter bank, consisting of bandpass filters which select different parts of the frequency spectrum. When applied to a time-domain signal (using convolution in the time domain, or multiplication in the frequency domain) I return a set of bandpassed signals, each representing different frequency bands of the original signal. By attenuating bandpassed signals by the amount of noise present in that frequency band,

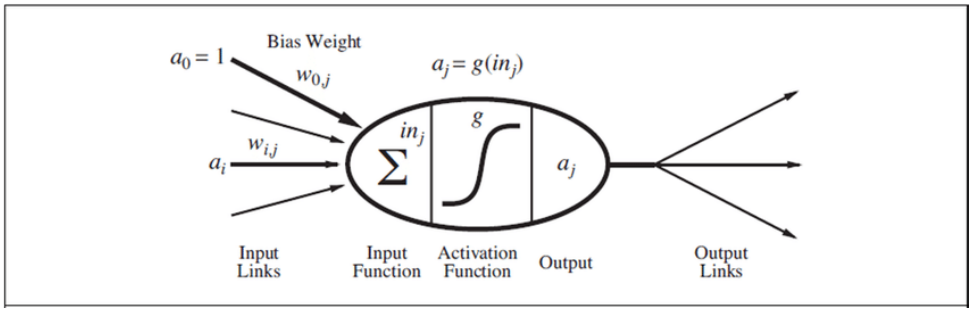


Figure A.1: A simple mathematical model for a neuron [66]. Note that $a_0 = 1$, meaning that $w_{0,j}$ is the bias for the neuron.

and then summing these signals together, I can reassemble the cleaned signal. Alternatively, the frequency-domain method (used in the Nelke-Vary algorithm [1]) uses the FFT to compute the complex magnitude of each frequency in my signal; then multiplies each magnitude with corresponding real-valued 'gains', which attenuates each frequency by different amounts to remove noise. The latter algorithm should offer improved performance through avoiding inefficient bandpass operations, whilst sacrificing granularity of my gains (through only having frequency-domain gains for a given signal window, rather than for each sample and frequency band).

A.2 Artificial Intelligence

A.2.1 Neural Networks

A neural network is a graph of bias nodes, connected by weights (or edges). The output activation of a node a_j is given by:

$$a_j = g(\sum_{i=0}^{N} w_{i,j} a_i),$$

where a_i is an input node, $w_{i,j}$ is the weight of the link from a_i to a_j, and g is the activation function applied to the result [66]. A simple mathematical model for a neuron is given in Figure A.1.

A.2.2 Supervised Learning

Supervised learning is where a neural network learns a mapping from inputs to outputs, given an example dataset of input-output pairs [66]. Given an input and output space $\mathcal{I}$ and $\mathcal{O}$ respectively. and a set of training data $D = \{(I_1, O_1), (I_2, O_2), \ldots, (I_N, O_N)\} \subseteq \mathcal{I} \times \mathcal{O}$, then the neural network aims to learn the mapping function $g : \mathcal{I} \to \mathcal{O}$ where $g(I_i) = O_i$.

A.2.3 Loss Functions

The loss function is a function $f(h_w(I), O)$, which maps the output of a network $f_w(I)$ and the expected output O to a real-valued number $n \in [0, 1]$, indicating the error or 'loss' between these values. I use the mean squared error loss function:

$$Loss(w) = [O - h_w(I)]^2,$$

to evaluate my network's performance. By following the gradient $\frac{\delta Loss(w)}{\delta w}$ via gradient descent, I can calculate the optimum weight updates δw for each epoch, reducing my loss (error) of my network. Formally, these weight updates are of the form:

$$w_{i,j} \leftarrow w_{i,j} + \alpha \times h_w(I)_i \times \Delta_j,$$

where $\Delta_j = Err_j \times \frac{\delta}{\delta w} a_j = |y - h_w(x)|_j \times \frac{\delta}{\delta w} a_j$. Backpropagation is then used to update weights, by backpropagating errors through the network.

Appendix B

Additional Analysis of Wind Noise

B.1 Analysis of the Amplitude of Wind Noise and Speech

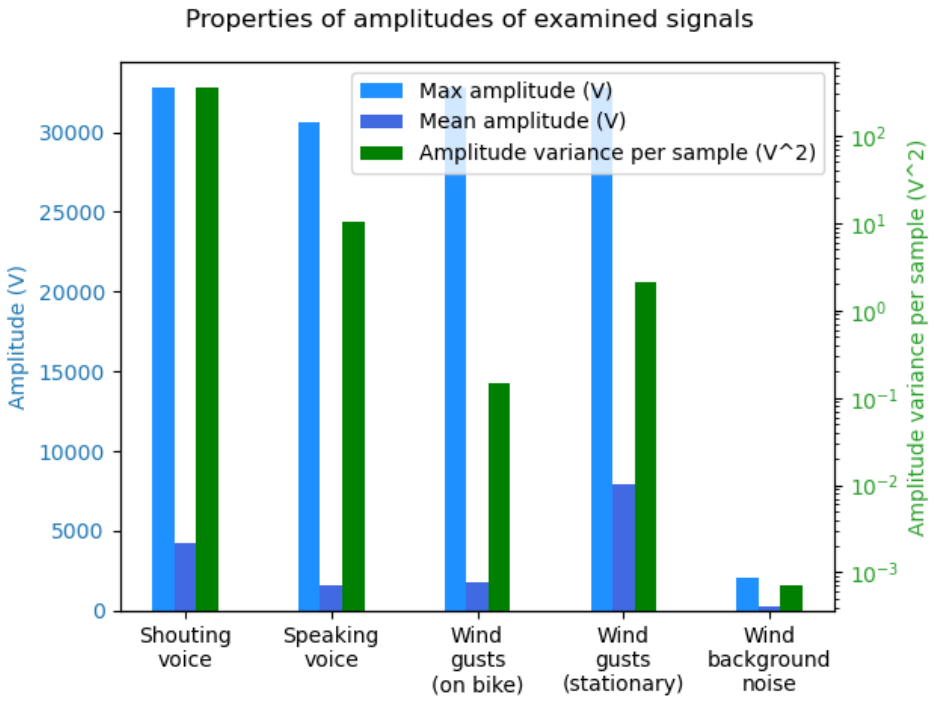


Figure B.1: The mean amplitude, max amplitude and variance in the amplitude of each type of wind gust/ voice.

Our amplitude graph (Figure B.1) shows that strong wind noise (represented by my stationary wind noise) has a much higher average amplitude, with a maximum amplitude which reaches the maximum (2^{15}-1). My shouting voice audio and bike wind noise similarly reach maximum amplitude, with my speaking voice audio also nearing this value. As a result, I may introduce further integer overflows/underflows if I naïvely combine my wind and speech signals; and in the case of my wind noise (recorded using my dictaphone), this indicates that the louder my wind is (indicated by mean amplitude and my volume graph) the more likely I are to have integer overflows/underflows.

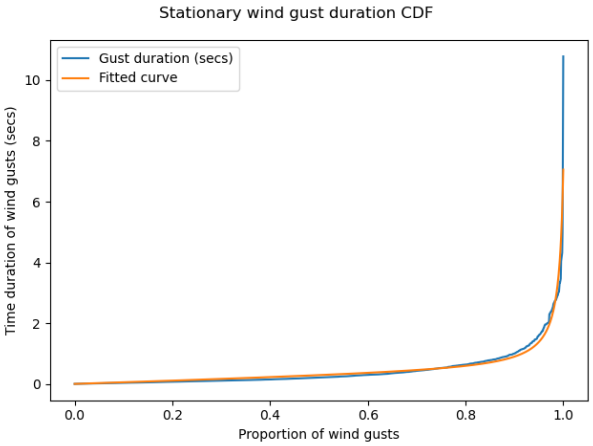
B.2 Fitting the CDF of Stationary Wind Gust Durations

As shown in Figure B.2, my wind gust duration and wind 'gap' duration (i.e. the time between wind gusts) cumulative distribution functions can be approximated using the equation:

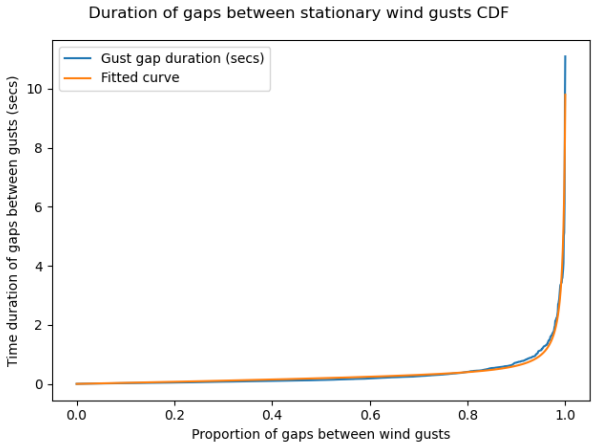
$$\mathrm{cdf}_{\mathrm{wind}}(x) \approx 1/1 - x.$$

Therefore, using SciPy's `curve_fit` function, I fit each curve to the equation:

$$F(x) = ax + \left(\frac{1}{(b-x)c} - \frac{1}{bc}\right),$$



(a) A graph illustrating the cumulative distribution function of the lengths of wind gusts.



(b) A graph illustrating the duration of gaps between wind gusts.

Figure B.2: Cumulative distribution functions mapping the duration of wind gusts and gaps between wind gusts in my stationary wind noise.

where a, b, and c are constants and $\text{cdf}_{\text{wind}}(0) = 0$. This gives the fit

$$D_{gust}(x) = 0.242x + \left(\frac{1}{(0.4646 - x) \cdot 17.10} - \frac{1}{1.0088 \cdot 17.10} \right)$$

for the duration of stationary wind gusts; and

$$D_{gap}(x) = 0.307x + \left(\frac{1}{(1.0041 - x) \cdot 25.52} - \frac{1}{1.0041 \cdot 25.52} \right)$$

for the time between stationary wind gusts in seconds.

Appendix C

Failed Improvements to the Nelke-Vary Algorithm

C.1 Attempts to Remove Wind Peaks from the PSD

I attempted three different methods of detecting and removing wind peaks with limited success.

My first attempt exploited the short-term nature of wind noise to detect peaks which only last one frame. This method compared the frequency values of sets of three adjacent frames, using a threshold to compare the ratios of power for each frequency between adjacent frames, and attenuated frequencies whose ratio exceeded the threshold. Unfortunately, whilst removing the vast majority of wind noise peaks, this also eliminated short-term peaks in the audio, notably consonants (affecting speech coherency [53]).

Another attempt used PCA to remove short-term artefacts from the output. This worked by first creating PCA components for clean speech output, and *for the error* ($output_{dirty} - output_{clean}$). Two sets of components were made (one roughly for each gender of voice), with the SSC 'centre-of-gravity' used to select the components used. I then performed PCA, attenuating error components on reconstruction. This leads to a cleaner, albeit muffled, low-pitched output signal.

My final attempt used speech PCA components but utilised the error between the original output and the PCA-reconstructed output to detect and remove artefacts (by attenuating frequencies where this error exceeded a threshold) – this created 'jittery' artefacts that further masked the speech present.

Appendix D

Additional Evaluation Results

D.1 Human Testing MOS Scores

	Algo wind – 0.2		Real wind – Strong		Algo wind – 0.8	
	Mean	Variance	Mean	Variance	Mean	Variance
Original Signal	2.86	0.551	1.79	0.883	1.43	**0.531**
Nelke-Vary Original	**4.21**	0.883	4.5	0.393	**4.29**	0.776
Nelke-Vary Adapted	3.93	**0.352**	**4.64**	**0.372**	4.14	1.12
Lee-Theunissen Original	3.07	0.495	2.64	0.515	2.57	0.959
Lee-Theunissen Adapted	3.21	0.74	2.71	1.2	2.71	0.776
My Solution	3.79	0.454	3.71	0.776	3.64	0.944

Table D.1: Human MOS scores for wind noise.

	Algo wind – 0.2		Real wind – Strong		Algo wind – 0.8	
	Mean	Variance	Mean	Variance	Mean	Variance
Original Signal	3.5	0.821	2.57	1.67	2.64	1.23
Nelke-Vary Original	1.93	1.49	1.5	0.821	1.57	1.24
Nelke-Vary Adapted	2.64	0.944	1.36	**0.801**	1.64	1.23
Lee-Theunissen Original	3.14	**0.694**	2.71	1.35	**3.14**	**0.265**
Lee-Theunissen Adapted	3.21	0.74	**3**	1	3	0.429
My Solution	**3.79**	0.74	2.86	0.98	2.93	1.07

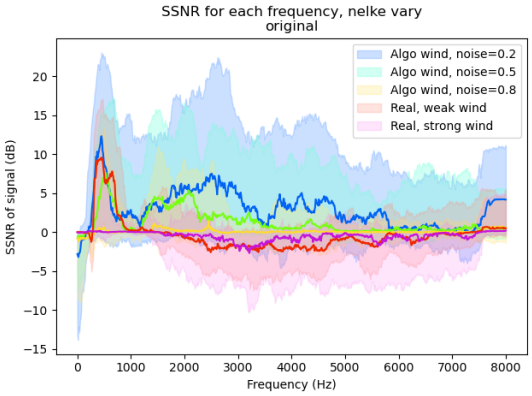
Table D.2: Human MOS scores for artefacts.

	Algo wind – 0.2		Real wind – Strong		Algo wind – 0.8	
	Mean	Variance	Mean	Variance	Mean	Variance
Original Signal	3.64	1.23	3.07	0.638	3.71	0.49
Nelke-Vary Original	2.64	0.372	2	1	2.36	1.09
Nelke-Vary Adapted	3.43	0.673	1.64	**0.23**	2.71	1.06
Lee-Theunissen Original	3.57	0.388	3.14	0.694	3.71	0.776
Lee-Theunissen Adapted	4	0.429	**3.71**	0.633	3.5	**0.393**
My Solution	**4.43**	**0.245**	3.5	0.964	**4**	0.857

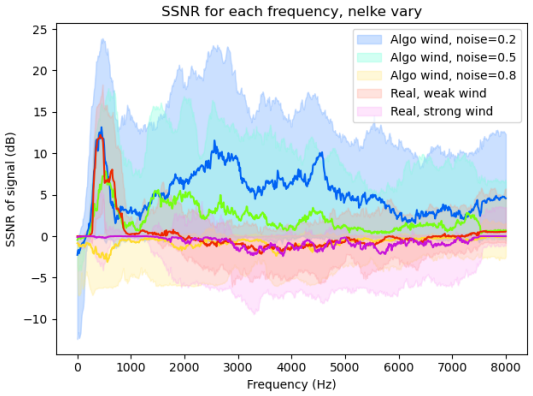
Table D.3: Human MOS scores for speech.

My MOS scores show a similar trend to my DCR scores, with the Nelke-Vary algorithm performing best at removing wind noise, the Lee-Theunissen algorithm introducing the least number of artefacts, and my solution offering the best clarity of the resulting speech signal. However, due to confusion of participants in the test on how to answer these questions, I have relegated these results to the appendix, since I feel they are less reliable than the corresponding DCR scores.

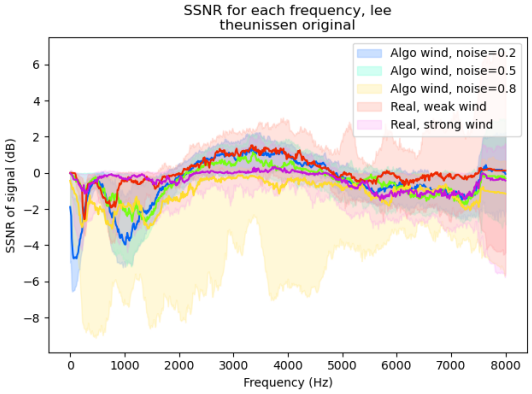
D.2 Individual Analysis of Each Algorithm using a Variety of Metrics



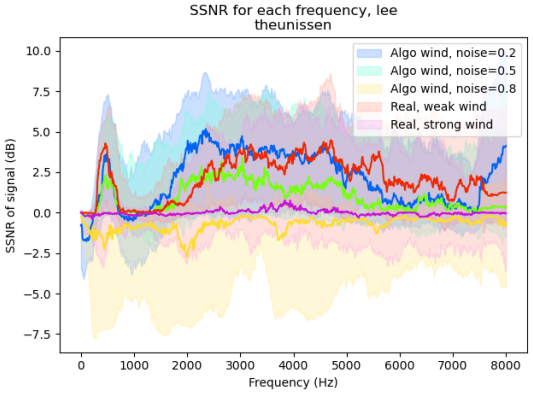
(a) The segmented SSNR of my original Nelke-Vary algorithm, plotted for different strengths of wind noise. Note the poor signal de-noising performance at frequencies > 1500Hz.



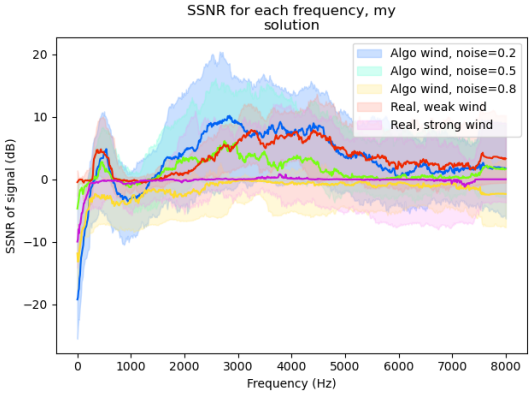
(b) The segmented SSNR of my adapted Nelke-Vary algorithm, plotted for different strengths of wind noise. Note the slightly better performance at higher frequencies (e.g. > 3000Hz).



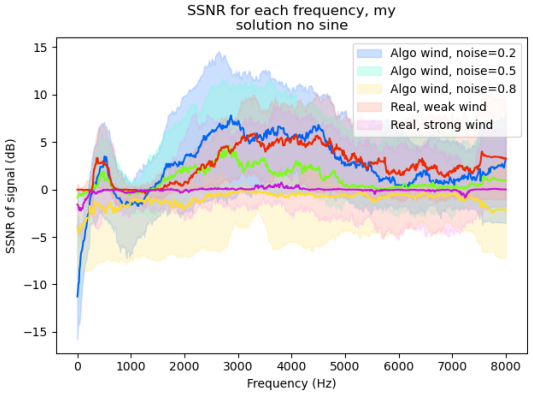
(c) The segmented SSNR of my original Lee-Theunissen algorithm, plotted for different strengths of wind noise. Note the poor performance for frequencies < 1500Hz and > 6000Hz.



(d) The segmented SSNR of my adapted Lee-Theunissen algorithm, plotted for different strengths of wind noise. Note the improved performance across all frequencies, particularly for frequencies < 1500Hz and > 5500Hz.



(e) The segmented SSNR for my solution, plotted for different strengths of wind noise. Note the overall improved noise removal compared to my 'no-sine' algorithm.



(f) The segmented SSNR for my solution, plotted for different strengths of wind noise. Note the improved performance for frequencies $500 - 1500$Hz compared to my other solution.

Figure D.1: The SSNR for each algorithm, segmented by wind noise strength.

	MSE (V^2)		SSNR (dB)		PESQ narrow-band		PESQ wide-band		STOI	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
0.2	7.89×10^5	4.36×10^{11}	6.73	11.6	2.24	0.155	1.24	0.0221	0.855	0.00338
0.5	1.47×10^6	9.24×10^{11}	4.47	5.47	1.79	0.0772	1.12	0.00501	0.789	0.00708
0.8	6.68×10^6	3.84×10^{13}	1.58	2.59	1.38	0.0404	1.05	0.00117	0.633	0.017
weak	2.46×10^6	1.03×10^{13}	4.36	9.71	1.56	0.0691	1.11	0.00795	0.749	0.0136
strong	3.13×10^6	5.22×10^{13}	4.28	9.93	1.6	0.09	1.11	0.00749	0.751	0.0144

Table D.4: All main metrics for the original Nelke-Vary algorithm.

	MSE (V^2)		SSNR (dB)		PESQ narrow-band		PESQ wide-band		STOI	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
0.2	7.51×10^5	3.63×10^{11}	7.1	10.1	2.33	0.164	1.27	0.026	0.877	0.00325
0.5	1.75×10^6	1.74×10^{12}	4.43	6.46	1.84	0.0917	1.14	0.00629	0.814	0.00687
0.8	8.79×10^6	6.36×10^{13}	1.15	4.75	1.39	0.044	1.06	0.00135	0.653	0.0192
weak	3.18×10^6	2.05×10^{13}	4.47	12.8	1.6	0.0798	1.13	0.011	0.782	0.0133
strong	4.12×10^6	8.28×10^{13}	4.17	11.8	1.64	0.101	1.13	0.0108	0.785	0.014

Table D.5: All main metrics for the adapted Nelke-Vary algorithm.

	MSE (V^2)		SSNR (dB)		PESQ narrow-band		PESQ wide-band		STOI	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
0.2	1.47×10^6	6.15×10^{11}	-2.05	0.668	2.39	0.179	1.4	0.0768	0.901	0.00312
0.5	2.24×10^6	1.65×10^{12}	-1.59	0.952	1.89	0.134	1.17	0.0279	0.845	0.00526
0.8	5.09×10^6	1.05×10^{13}	-1.29	1.63	1.45	0.0553	1.06	0.00119	0.688	0.0186
weak	2.9×10^6	7.65×10^{12}	-1.89	1.01	1.68	0.0793	1.18	0.0216	0.823	0.011
strong	3.14×10^6	1.81×10^{13}	-1.82	1.11	1.72	0.114	1.2	0.036	0.824	0.0115

Table D.6: All main metrics for the original Lee-Theunissen algorithm.

	MSE (V^2)		SSNR (dB)		PESQ narrow-band		PESQ wide-band		STOI	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
0.2	8.58×10^5	3.35×10^{11}	3.58	3.07	2.45	0.159	1.56	0.104	0.902	0.00224
0.5	1.61×10^6	1.37×10^{12}	2.31	2.87	1.98	0.118	1.26	0.039	0.857	0.00409
0.8	4.44×10^6	1.14×10^{13}	0.22	3.48	1.54	0.068	1.09	0.00365	0.727	0.0165
weak	2.38×10^6	8.56×10^{12}	2.4	5.68	1.87	0.108	1.3	0.0489	0.851	0.00652
strong	2.57×10^6	1.71×10^{13}	2.4	5.69	1.9	0.161	1.31	0.0765	0.849	0.00801

Table D.7: All main metrics for the adapted Lee-Theunissen algorithm.

	MSE (V^2)		SSNR (dB)		PESQ narrow-band		PESQ wide-band		STOI	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
0.2	6.6×10^5	4.09×10^{11}	5.84	11.5	2.82	0.165	2.11	0.151	0.909	0.00197
0.5	1.04×10^6	1.27×10^{12}	4.52	8.7	2.27	0.163	1.58	0.0845	0.861	0.0045
0.8	4.86×10^6	2.15×10^{13}	0.955	7.02	1.63	0.128	1.21	0.0308	0.702	0.0224
weak	1.94×10^6	9.84×10^{12}	4.37	15.2	2.11	0.222	1.62	0.155	0.843	0.00995
strong	2.48×10^6	3.63×10^{13}	4.59	14.2	2.16	0.26	1.65	0.165	0.846	0.0108

Table D.8: All main metrics for my solution.

	MSE (V^2)		SSNR (dB)		PESQ narrow-band		PESQ wide-band		STOI	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
0.2	7.03×10^5	3.46×10^{11}	4.48	7.09	2.8	0.199	1.96	0.173	0.917	0.00222
0.5	1.13×10^6	1.21×10^{12}	3.31	5.69	2.2	0.17	1.46	0.0744	0.865	0.00476
0.8	4.81×10^6	1.94×10^{13}	0.494	5.02	1.58	0.109	1.16	0.0145	0.704	0.0219
weak	1.99×10^6	9.1×10^{12}	3.23	10.0	1.98	0.179	1.47	0.112	0.847	0.0106
strong	2.44×10^6	3.07×10^{13}	3.42	9.71	2.03	0.222	1.51	0.135	0.849	0.0111

Table D.9: All main metrics for my solution (without sine activation function).

D.3 Nelke-Vary Additional Hyperparameter Analysis

D.3.1 Exploration of Hyperparameter Values for `f2`

	MSE (V^2)		SSNR (dB)		PESQ narrow-band		PESQ wide-band		STOI	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
300	5.61×10^6	1.21×10^{14}	4.32	**11.8**	1.77	0.222	**1.17**	0.0295	**0.804**	0.0167
400	4.35×10^6	7.78×10^{13}	**4.68**	13.1	**1.78**	0.222	**1.17**	0.0236	0.802	0.0165
500	3.83×10^6	5.69×10^{13}	**4.68**	13.4	**1.78**	0.213	1.16	0.0194	0.797	0.0162
600	3.59×10^6	4.67×10^{13}	4.56	13.2	1.77	0.204	1.15	0.017	0.792	0.0158
650	3.53×10^6	4.35×10^{13}	4.5	13.2	1.77	0.201	1.15	0.0167	0.789	0.0157
700	3.48×10^6	4.15×10^{13}	4.46	13.1	1.77	0.199	1.15	0.0165	0.787	0.0155
800	3.39×10^6	3.75×10^{13}	4.37	13.0	1.76	0.194	1.15	0.016	0.784	0.0152
900	3.33×10^6	3.47×10^{13}	4.31	12.8	1.76	0.19	1.15	0.0159	0.781	0.0148
1000	$\mathbf{3.28 \times 10^6}$	$\mathbf{3.2 \times 10^{13}}$	4.24	12.6	1.75	**0.184**	1.14	**0.0153**	0.778	**0.0145**

Table D.10: Metrics showing the different hyperparameter values for `f2`, for the Nelke-Vary algorithm.

The behaviour of `f2` is very similar to the hyperparameter values for `f1` – I note that both low and high values perform optimally for different metrics. Therefore, I again retain `f2`'s value of 650Hz, since it offers adequate performance in all metrics.

D.3.2 Exploration of Hyperparameter Values for $\mathbf{SSC_{max}}$

	MSE (V^2)		SSNR (dB)		PESQ narrow-band		PESQ wide-band		STOI	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
1000	$\mathbf{2.91 \times 10^6}$	$\mathbf{2.47 \times 10^{13}}$	3.98	14.5	1.6	**0.129**	1.13	**0.0112**	0.735	0.0178
2000	3.33×10^6	3.76×10^{13}	4.51	13.4	1.74	0.184	**1.15**	0.0152	0.781	0.0157
3000	3.53×10^6	4.35×10^{13}	4.5	13.2	1.77	0.201	**1.15**	0.0167	0.789	0.0157
4000	3.62×10^6	4.68×10^{13}	**4.53**	13.0	1.79	0.21	**1.15**	0.0178	0.795	0.0154
6000	3.73×10^6	5.07×10^{13}	**4.53**	12.7	1.8	0.218	**1.15**	0.0184	0.801	0.015
8000	3.81×10^6	5.26×10^{13}	4.51	**12.5**	**1.81**	0.221	**1.15**	0.0187	**0.803**	**0.0149**

Table D.11: Metrics showing the different hyperparameter values for $\mathrm{SSC_{max}}$, for the Nelke-Vary algorithm.

For larger values of $\mathrm{SSC_{max}}$ (or `SSC1MaxFrequency`), my estimate of the mean frequency of the signal will be more accurate, in turn giving us a better estimate of how much wind is present in my signal, and hence aiding wind removal. Smaller values will speed up computation time, through reducing the number of frequencies considered.

Once again, there is disagreement in the best value. For mean-squared error, we want to over-approximate wind noise (since wind noise is louder than speech, speech attenuation is less of an issue), hence 1000Hz is given as the best parameter. For speech, we want more accurate estimates of mean frequency, hence larger values appear to perform well in PESQ and STOI. Finally, SSNR indicates that 4000-6000Hz is the optimal frequency for $\mathrm{SSC_{max}}$.

Since my current value 3000Hz offers similar performance in all metrics to my value for SSNR, I will continue using this in my adapted algorithm. This is because any changes here (that otherwise improve my solution in isolation) may be detrimental when combined.

D.3.3 Exploration of Hyperparameter Values for PSD Sampling Locations

	MSE (V^2)		SSNR (dB)		PESQ narrow-band		PESQ wide-band		STOI	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
1 (31.25Hz)	6.95×10^6	1.47×10^{14}	4.25	13.3	1.77	0.226	1.17	0.0296	0.797	0.017
3 (93.75Hz)	4.27×10^6	7.56×10^{13}	**5.4**	16.7	**1.8**	0.234	**1.21**	0.0364	**0.798**	0.0171
5 (156.25Hz)	3.54×10^6	5.17×10^{13}	5.05	14.7	**1.8**	0.223	1.18	0.0239	0.797	0.0164
7 (218.75Hz)	$\mathbf{3.53 \times 10^6}$	4.35×10^{13}	4.5	13.2	1.77	0.201	1.15	0.0167	0.789	0.0157
9 (281.25Hz)	3.68×10^6	3.92×10^{13}	4.07	11.8	1.74	0.189	1.13	0.0125	0.781	0.0152
11 (343.75Hz)	3.74×10^6	3.74×10^{13}	3.76	11.4	1.7	0.176	1.11	0.00977	0.759	**0.0151**
13 (406.25Hz)	3.79×10^6	$\mathbf{3.54 \times 10^{13}}$	3.53	**11.3**	1.66	**0.164**	1.1	**0.00893**	0.74	0.0153

Table D.12: Metrics showing the different hyperparameter values for `placeToSampleNo1` for the Nelke-Vary algorithm.

	MSE (V^2)		SSNR (dB)		PESQ narrow-band		PESQ wide-band		STOI	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
20 (625Hz)	$\mathbf{3.41 \times 10^6}$	$\mathbf{4.02 \times 10^{13}}$	4.5	13.7	1.7	**0.175**	1.14	**0.0151**	0.765	0.0172
25 (781.25Hz)	3.51×10^6	4.25×10^{13}	4.42	13.1	1.74	0.192	1.14	0.0157	0.778	0.0162
30 (937.5Hz)	3.55×10^6	4.28×10^{13}	4.43	**13.0**	1.76	0.198	1.14	0.0158	0.785	0.0158
35 (1093.75Hz)	3.53×10^6	4.35×10^{13}	4.5	13.2	**1.77**	0.201	**1.15**	0.0167	**0.789**	**0.0157**
40 (1250Hz)	3.49×10^6	4.32×10^{13}	4.57	13.4	**1.77**	0.203	**1.15**	0.0174	**0.789**	0.0159
45 (1406.25Hz)	3.48×10^6	4.3×10^{13}	**4.6**	13.5	1.76	0.195	**1.15**	0.0169	0.787	0.0161
50 (1562.5Hz)	3.49×10^6	4.33×10^{13}	4.57	13.5	1.75	0.187	**1.15**	0.0164	0.781	0.0164

Table D.13: Metrics showing the different hyperparameter values for `placeToSampleNo2` for the Nelke-Vary algorithm.

Our sample locations offer interesting insights. Firstly, unlike a lot of the other metrics presented, my results do not vary linearly or quadratically with their value – that is, there is no single best value, nor any clear trend to follow. Instead, I see that (especially for the second sample location) my results vary relatively erratically. This is because I want to sample locations of the signal PSD which contain as little of the voice signal as possible. As a result, I want to choose locations which lie at frequencies between the harmonic frequencies of speech. This results in the values provided in Tables D.12 and D.13.

For `placeToSampleNo1`, I see that the optimal position to sample at varies again based on the metric. For mean squared error, a sample frequency of 218.75Hz is preferred; whilst for other metrics a sample position of 93.75 is optimal. This is set to the former in my original algorithm – I will continue to use this value, since it offers far better MSE performance, whilst achieving similar scores to 218.75Hz in PESQ and STOI.

For the second sample position, my current sample location of 1093.75Hz offers optimal performance in PESQ and STOI, whilst achieving respectable performance in other metrics. I note that there is less variance in the mean scores here – this is likely since the majority of each noisy signal occurs below $\sim$ 500Hz, where most fundamental frequencies of speech and most wind noise lies. Therefore, I will continue to use this value.

D.4 Lee-Theunissen Additional Hyperparameter and NN Architecture Analysis

D.4.1 Exploration of Hyperparameter Values for `numberFilters`

(`passband` - `numberFilters`)	MSE (V^2)		SSNR (dB)		PESQ narrow-band		PESQ wide-band		STOI	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
(256-112)	8.51×10^5	2.54×10^{12}	**9.15**	10.9	3.05	0.399	1.83	0.295	0.94	0.00126
(128-223)	$\mathbf{6.88 \times 10^5}$	$\mathbf{1.53 \times 10^{12}}$	7.39	8.53	**3.11**	0.411	**1.99**	0.326	**0.95**	0.000751
(96-298)	1.3×10^6	2.07×10^{12}	0.479	**1.34**	2.72	**0.349**	1.78	**0.212**	0.938	**0.000524**

Table D.14: All main metrics for the number of filters, in the Lee-Theunissen algorithm.

Our analysis here is slightly more complex. Since I only want to analyse the number of filters, I need to increase the size of my filters to cover the same frequencies. As a result, I change `passband` along with `numberFilters` such that `passband` $\cdot$ `numberFilters` remains constant.

As shown by all metrics (except SSNR), my current values for `passband` and `numberFilters` offer the best results, with notably better performance for MSE. However, more smaller filters offer reduced variance, whilst larger filters offer more accurate signal reconstruction in terms of SSNR.

D.4.2 Exploration of the Number of PCA Components for the 'Gains PCA' NN Architecture

	MSE (V^2)		SSNR (dB)		PESQ narrow-band		PESQ wide-band		STOI	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
PCAGains50	3.36×10^6	1.83×10^{13}	-1.86	1.08	1.83	0.23	1.23	0.064	0.814	0.0142
PCAGains100	3.38×10^6	1.87×10^{13}	-1.64	0.991	1.84	0.235	1.23	**0.0638**	0.816	0.014
PCAGains200	3.33×10^6	1.86×10^{13}	-1.25	**0.943**	1.84	**0.226**	1.24	0.0674	0.819	0.0133
PCAGains500	3.29×10^6	1.9×10^{13}	-0.524	1.11	1.88	0.229	1.26	0.0736	0.823	0.0126
PCAGains1000	$\mathbf{2.56 \times 10^6}$	$\mathbf{1.54 \times 10^{13}}$	**2.21**	5.55	**1.97**	0.242	**1.34**	0.103	**0.838**	**0.00943**

Table D.15: Comparing my 'gains PCA' architecture autoencoders using all main metrics, for the Lee-Theunissen algorithm. The number in the name indicates the number of PCA components used for the input of that autoencoder.

Once again, an increased number of PCA components leads to improved performance. I also note that this neural network (with gains PCA components) has marginally better performance than the original PCA components. Therefore, using PCA components derived from gains as my output does enable us to represent output gains with increased accuracy.

D.4.3 Exploration of the Number of PCA Components for the 'No PCA' NN Architecture

	MSE (V^2)		SSNR (dB)		PESQ narrow-band		PESQ wide-band		STOI	
	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance	Mean	Variance
noPCA50	$\mathbf{2.1 \times 10^6}$	$\mathbf{1.24 \times 10^{13}}$	**3.93**	10.8	**2.07**	0.305	1.41	0.137	**0.848**	0.00876
noPCA500	2.13×10^6	1.33×10^{13}	3.86	10.4	2.06	0.305	**1.42**	0.134	0.845	**0.00862**

Table D.16: Comparing my 'no PCA' architecture autoencoders using all main metrics, for the Lee-Theunissen algorithm. The number in the name indicates the number of nodes in each 'middle layer' of the architecture.

I see a noticeable improvement when removing the PCA from the neural network, with more nodes in the middle layer actually detrimenting performance. This may be since my smaller

(50 nodes each in the middle layers) network is able to better generalise my wind noise, and that my larger network is overfitting the data.

This may indicate that a different architecture, or different PCA components from differently structured input data, would improve performance. This is likely possible since a linear mapping from my input to a reduced-dimension representation has been learnt by my 'no PCA' architecture, with less error than its equivalent PCA counterpart.

Appendix E

Consent Form

Consent Form

Title of Research Project: An exploration of removing wind noise from mono speech input using DSP and AI approaches

Researchers: ----, Dr Dong Ma, Dr Jing Han (University of Cambridge)

Name of Participant: Click or tap to enter your name here.

Purpose of Study:

This project aims to explore existing solutions to remove wind noise from a mono input source. Wind noise is caused through air vortices forming on the surface of the microphone, in turn causing wind gusts to be amplified.

You (the participant) will be asked to listen to a selection of wind noise audio clips, and to fill out corresponding questions about these on a Google form (https://forms.gle/srLRoB6xDTFxE7W38). You will be asked in this form to provide your age anonymously, to see if this has a factor in perception. Three sample voice signals will be presented, both with wind noise, and having been ran through five algorithms which aim to remove this wind noise from the audio.

Due to COVID-19, this experiment will be completed online. As a result, additional precautions will be made to ensure the participant's safety when completing this experiment.

Procedure:

Before data collection occurs, time will be spent verbally briefing the participant on the contents of this consent form. In addition, links to the online form will be provided, and guidance will be provided on how to set up the participant's headphones and their test environment. In particular, the participant will be guided through setting up their volume, including reducing the volume, playing a loud tone, and increasing the volume until this tone is loud but comfortable to listen to for a prolonged duration.

Once the experiment has been set up, the participant will be asked to complete the following actions:

1. Listen to a selection of audio clips – these are either voice and wind mixed signals, or voice signals where wind noise has been removed using an algorithm.
2. For each, provide three scores between 1 and 5, signifying how much wind has been removed, how much noise/artefacts are present in the signal, and how clear the voice in the signal is.
3. For each, to offer a very brief description of how each signal sounds, including how well it removes noise and retains speech in our signal.
4. For wind-removed signals, to offer three scores between 1 and 5, comparing these signals to the original 'speech-and-wind' signal in wind removal, presence of noise/artefacts, and clearness of voice.

For each *voice*, six *audio* clips will be provided, one with wind noise present, and five audio clips where wind noise has been removed using an algorithm.

Each clip is between 10 and 30 seconds in length, with the approximate time to complete each set of clips (six clips per set) being 5-10 minutes each. The expected duration of each test will be approximately 30-40 minutes.

Participants will be free to withdraw at any time, without providing an explanation. All participants will be debriefed at the end of the experiment.

Confidentiality:

All information collected in this study is confidential. user data will be recorded, except for age. All data collected will be stored securely offline, with only the listed researchers having access to this data. Once the data is analysed, all data will be destroyed, with only aggregated data being used for any publications. As per GDPR rules, participants have the right to request the data collected by this experiment, and/or to have this data deleted once collected.

Consent:

I have had the opportunity to discuss this study with an investigator, and I am satisfied that my questions have been answered. I voluntarily consent to participate in this study with the understanding that I can withdraw at any time.

Participant Signature: Click or tap to enter your name here.

Investigator Signature: ----

Date: Click or tap to enter a date.

Appendix F

Ethics Form

Ethics Form

Title of Research Project: An exploration of removing wind noise from mono speech input using DSP and AI approaches

CRSID of overseers: mgk25, lp15

CRSID of supervisors: dm878, jh2298

CRSID: ----

Name: ----

Email: ----

Project Start Date: 13/04/21

Project End Date: 31/04/21

Type of study: Controlled experiments for technology evaluation

Funding body: N/A

Brief description:

This project aims to explore existing solutions to remove wind noise from a mono input source. Some research focus has already been given to removing this type of noise from audio, due to its prevalence in microphone input – air vortices form on the surface of the microphone, causing wind gusts to be amplified. Removing such noise has direct applications in both outdoor sports/recreation and aviation, where wind feedback could affect awareness of surroundings and mask other noises (such as traffic or plane damage), hence potentially reducing safety.

My solution will involve analysing and modelling wind noise, as well as reimplementing two existing noise-removal algorithms, one utilising a DSP solution (Nelke and Vary, https://ieeexplore.ieee.org/document/6854970) and one using an AI autoencoder (Lee and Theunissen, https://royalsocietypublishing.org/doi/pdf/10.1098/rspa.2015.0309). These solutions will then be evaluated, adjusted to see how various parameters affect results, and a new solution created using all the above, hopefully achieving comparable if not better results on wind noise.

I will then test my solution in a few ways – methodologies such as PESQ, WER and SNR will be used to determine how recognisable speech is before and after wind is removed, and to quantitatively assess the quality of our audio output for each algorithm. Artificially generated test data will be used to enable us to adjust the amount of wind present and remove wind noise completely for training and testing purposes – this will consist of a single voice reading passages from audiobooks, with generated wind noise added over the top.

I will also use 10-20 human participants to assess the output of each algorithm quantitatively and qualitatively. This is required since audio quality is up to human perception – for example, different frequency artefacts may be perceived as more or less distracting to the overall noise-removed signal; or more quieter artefacts may be preferred to fewer loud artefacts. Each participant will be

selected to ensure a mix of age, from a group of friends of the investigator. Each participant will be briefed about the research before the tests, and how any collected data will be processed.

Due to the Coronavirus pandemic, remote testing will be required. However, for our purposes this should not pose too much of an issue, since no specialised equipment is required – users can listen to audio clips remotely and fill out an online questionnaire to give their responses. The assessor will read out each question in order via an online video call, guiding the participant if they have any questions and both briefing and debriefing the participant at the start and end of the test, respectively. The following steps will be taken (following the necessary parts of ITU-T P.800.2 as much as possible):

- Participants will first open a pre-prepared Google questionnaire containing questions and accompanying audio clips (https://forms.gle/srLRoB6xDTFxE7W38). Each participant will download the audio clips before playing, to ensure buffering does not affect results. Briefing and debriefing information will also be provided in written form and will be repeated verbally before the test starts/ends.
- Firstly, participants will play a signal frequency audio signal equal to the maximum volume of the test data presented – each participant will be asked to turn down their volume and turn this up until it is at a loud but comfortable volume. This ensures that the headphones are at a suitable volume for each participant. Since these experiments will be done remotely (and hence each participant may use assorted brands and types of headphones) this will also ensure that volume cannot be a factor in affecting our MOS scores – the participant will adjust their volume based on this test signal to ensure it is clearly audible but does not hurt their ears.
- Next, we provide participants with several audio clips, based on the same voice signal. For each voice signal:
    - One clip is of this voice signal with wind present
    - All other clips (five clips) are of this same voice signal, after wind has been removed using an algorithm.

  The order of each clip (and hence the order each algorithm is presented) is randomised, to prevent bias.

  For each voice, we present six audio clips, one with voice and wind present, and the other five coming from three algorithms:

    - My own algorithm
    - An AI algorithm, both as published, and with my own alterations to improve performance
    - A DSP algorithm, both as published and with my own alterations to improve performance.

- Based on each clip, we ask the participant for:
    - A MOS score, between 1 and 5, signalling how much wind noise has been removed from the signal

- A second MOS score for how many artefacts / noises are left behind by the algorithm once wind noise is removed.
- A third MOS score for how decipherable/clear the speech is to understand in the signal
- A qualitative description of how these samples sound, including what artefacts are present, both in the sound of the voice and any background artefacts
- Three DCR scores for the latter two signals (with wind noise not present / removed) - this is an indicator of how the signal sounds compared to the signal with wind noise present (both in terms of cleanness and decipherability). This is especially important given remote testing, since it helps to alleviate differences in listening conditions through comparing to a standardised control signal.

Each result will be recorded from the applicants and will be investigated to validate the feasibility and performance of each algorithm. The test will contain 18 clips, at an average length of 18s each, and 126 short questions – this is to enable testing at different intensities of speech, both from stationary and in-motion poses. Hence, the expected duration of each test will be <= 40 mins. Results will be stored securely and anonymised (except for age) to comply with GDPR laws.

Precautions:

- Participants will be volunteers, each signing a waiver to give consent. Information will be provided as to how data will be used and processed, and what the test will involve.
- We will ask participants if they have any temporary or permanent conditions which affect hearing. If this is the case, they will not be recruited.
- The experiment will be (where possible) conducted remotely, with the participant in a room of volume 20-100m^3 (this to follow ITU P.800 – headphones are used hence reverberations are much less important to the results obtained).
- The participant will be asked if they expect interruptions during the test, and if their current environment is free of background noises that might otherwise affect results. If this is not the case, the test can be paused during interruptions, and be made to run longer to accommodate this.
- Where possible, over-the-ear headphones will be used for the test – if not, then in-ear headphones should be used instead. In each case, I will ask the participant if the fit of their headphones lets in any ambient / external noises – if the headphones do let in ambient noise, we will ask the participant to either get better fitting headphones, to ensure their environment is free of ambient noise, else the participant will be rejected.
- Initial volume adjustment will be performed by starting at the lowest possible volume and increasing this until the highest volume is clearly audible. This avoids any possibility of ear damage due to the volume of the audio clips presented.
- A rough transcript will be taken of each test, including scores given and key words from their qualitative descriptions of each noise. This transcript is provided by Google Forms when the participant fills in their online form, as well as from supplementary notes made by me during the test (regarding queries or difficulties faced). No audio/video recording will be made of each test. A private video chat will be used for the tests to ensure no additional people can join the call.

- Only the age of each participant will be recorded, with all other parameters anonymised. Each participant will have the right for their data to be removed from the test.
- Participants will be briefed and made aware of the aims of the test, along with the fact they can withdraw at any time from the experiment.
- Participants will not be subject to physical or mental harm and will not be deceived.
- All participants will be debriefed, and care will be made to ensure this is sufficient given the remote nature of these tests.

Glossary

AI Artificial Intelligence. ii, 1–3, 7, 25, 27, 30, 68

DCR Degradation Category Rating. 27–30, 51, 69

DFT Discrete Fourier Transform. 46

DNN Deep Neural Network. 1

DSP Digital Signal Processing. ii, 1–3, 7

FFT Fast Fourier Transform. iv, 7, 13, 15, 24, 25, 36, 47

IDE Integrated Development Environment. 4

IFFT Inverse Fast Fourier Transform. iv, 15, 24, 25

MOS Mean Opinion Score. ii, iii, vi, 1, 27, 28, 40, 51, 69

MSE Mean Squared Error. ii, 27, 32, 35, 36, 38, 55, 56

NN Neural Network. v, vi, 2, 7, 18, 21, 22, 24, 25, 35, 38, 56

PCA Principal Component Analysis. vi, 2, 5, 7, 18–20, 22, 23, 32, 38, 39, 50, 56, 57, 67–70

PESQ Perceptual Evaluation of Speech Quality. 4, 27, 31, 32, 36, 54, 55

PSD Power Spectrum Density. iv, vi, 1, 2, 6, 12–16, 24, 25, 33, 37, 40, 55

SNR Signal-to-Noise Ratio. 27

SOTA State of the Art. ii, 2

SSC Spectral Sub-Band Centroids. 13, 14, 50, 67

SSNR Segmented Signal-to-Noise Ratio. ii, 27, 31, 32, 36–38, 40, 52, 54, 56, 68

SST Speech-to-Text. 7

STDR Spectro-Temporal Detection-Reconstruction. 18

STOI Short-Term Objective Intelligibility. 4, 27, 31, 32, 35, 36, 54, 55

WER Word Error Rate. ii, 1, 7, 27, 32, 33

List of Figures

1.1 The data collection and training pipeline of the 2hz.ai noise removal algorithm [16]. 2

2.1 The spiral model of software development, used in my project [24]. 3
2.2 The Hann window, used in my solution both for windowing and as an equal-voltage cross-fade function [40] 5
2.3 A comparison of an autoencoder with PCA for dimensionality reduction. Note both feature spaces are roughly the same [39] 5
2.4 Wind noise spectrum of an omnidirectional microphone at three wind speeds [42]. Note that the spectrum is both asymmetrical and irregular in shape, owing both to differences in the geometry of the microphone at different angles and the unpredictable nature of turbules. 6
2.5 A graph showing the relationship between wind speed, height of the sensor and the source region-inertial subrange frequency boundary [10]. 6

3.1 Graphs illustrating the process of producing my algorithmic bike wind noise. . . 9
3.2 The spectrogram of a wind noise segment captured (a) from a stationary pose at night, (b) on a bike, (c) from a stationary pose during the day; and (d) the spectrogram of isolated male speech. 10
3.3 The mean volume, max volume and variance in the volume of each type of wind gust/voice. 11
3.4 A graph of my wind gust detection method, with both time-domain and indicator function views. 11
3.5 Cumulative distribution functions illustrating (a) the duration of wind gusts and (b) the duration of gaps between wind gusts, for bike wind noise. 11
3.6 A graph showing the periodograms of speech and wind signals, as well as coloured noise approximations to the wind power spectra. Each periodogram is created using 90 minutes of each signal. 12
3.7 A diagram of the workflow of the Nelke-Vary algorithm. Algorithms specific to this algorithm are included in grey[1]. 12
3.8 Graphs showing the process of estimating our noise-removing gains. (a) A graph of the wind noise estimated by the Nelke-Vary algorithm, before and after my improvements (Section 3.6). (b) A graph of the SSC of a noisy speech frame. . 14
3.9 Graphs detailing the improvements made to the Nelke-Vary algorithm. 16
3.10 A diagram of the workflow of the Lee-Theunissen algorithm. Algorithms specific to this algorithm are included in grey [2]. 18
3.11 Graphs illustrating the various stages of preprocessing within the Lee-Theunissen algorithm. 19
3.12 Graph illustrating the spectrogram creation process within the Lee-Theunissen algorithm. 19
3.13 A diagram showing the basic architecture of the Lee-Theunissen autoencoder, as it appears in the original paper [2]. Here, N is the number of windows in my signal; F_N is the number of bandpass filters applied to my signal; C is the number of PCA components each from my speech and wind components; S_N is the number of samples in each window in my signal; and W_N is the number of windows passed at a time to the NN, to retrieve additional temporal information. 20

3.14 Spectrograms of the input gains, showing the difference between the (a) optimal and (b) predicted spectrograms. note that the optimal gains have either near-zero or near-one gains (with few in between), whilst our predicted gains features more intermediate gains values, indicating uncertainty. Also note the horizontal and vertical banding. 21
3.15 A diagram showing the architecture of my 'gains PCA' autoencoder architecture. Separate PCA components are used for the output, based on a development set of optimal gains. 22
3.16 A diagram showing the architecture of my no-PCA autoencoder architecture. Here, L is the number of nodes per input frame in the middle layer. 23
3.17 A diagram of the workflow of my solution. Algorithms specific to this algorithm are included in grey. 24
3.18 A diagram showing the architecture of my solution's AI autoencoder. I note that f_s is the sampling frequency of my signal, T is the length (in seconds) of each window, and D is the division factor used for my middle layers (here, this is set to 4). 25
3.19 Graphs showing the operation of my solution. 26

4.1 The SSNR of each algorithm, averaged over all audio files, as a function of frequency. 31
4.2 A set of spectrograms highlighting the various upsides and downsides of each algorithm. (a) and (b) show the original noisy signal, and the optimal clean speech signal; our other spectrograms show our actual output from denoising using the (c) original and (d) adapted Nelke-Vary algorithms, the (e) original and (f) adapted Lee-Theunissen algorithms, and my solution (g) with and (h) without the sine activation function present. 34
4.3 The SSNR for my values of `aMin` and `aMax` 37

A.1 A simple mathematical model for a neuron [66]. Note that $a_0 = 1$, meaning that $w_{0,j}$ is the bias for the neuron. 47

B.1 The mean amplitude, max amplitude and variance in the amplitude of each type of wind gust/ voice. 48
B.2 Cumulative distribution functions mapping the duration of wind gusts and gaps between wind gusts in my stationary wind noise. 49

D.1 The SSNR for each algorithm, segmented by wind noise strength. 52

List of Tables

4.1 Human DCR scores for wind noise. . . . 28
4.2 Human DCR scores for artefacts. . . . 29
4.3 Human DCR scores for speech. . . . 30
4.4 Main metrics comparing each algorithm presented. . . . 31
4.5 The mean word error rate of each algorithm, plus the corresponding input and optimal output signals. . . . 33
4.6 Metrics showing the different hyperparameter values for `f1`, for the Nelke-Vary algorithm. . . . 35
4.7 Metrics showing the different hyperparameter values for window size (in the form 2^n), for the Nelke-Vary algorithm. . . . 36
4.8 Metrics showing the different hyperparameter values for aMax for the Nelke-Vary algorithm. . . . 36
4.9 Metrics showing the different hyperparameter values for `aMin` for the Nelke-Vary algorithm. . . . 36
4.10 All main metrics for the size of each time-domain window, for the Lee-Theunissen algorithm. . . . 38
4.11 All main metrics for the number of overlapping windows (and size of each window), for the Lee-Theunissen algorithm. . . . 38
4.12 Comparing my original PCA architecture autoencoders using all main metrics for the Lee-Theunissen algorithm. The number in the name indicates the number of PCA components used (each from speech and wind) for the input of that autoencoder. . . . 38

D.1 Human MOS scores for wind noise. . . . 51
D.2 Human MOS scores for artefacts. . . . 51
D.3 Human MOS scores for speech. . . . 51
D.4 All main metrics for the original Nelke-Vary algorithm. . . . 53
D.5 All main metrics for the adapted Nelke-Vary algorithm. . . . 53
D.6 All main metrics for the original Lee-Theunissen algorithm. . . . 53
D.7 All main metrics for the adapted Lee-Theunissen algorithm. . . . 53
D.8 All main metrics for my solution. . . . 53
D.9 All main metrics for my solution (without sine activation function). . . . 53
D.10 Metrics showing the different hyperparameter values for `f2`, for the Nelke-Vary algorithm. . . . 54
D.11 Metrics showing the different hyperparameter values for SSC_{max}, for the Nelke-Vary algorithm. . . . 54
D.12 Metrics showing the different hyperparameter values for `placeToSampleNo1` for the Nelke-Vary algorithm. . . . 55
D.13 Metrics showing the different hyperparameter values for `placeToSampleNo2` for the Nelke-Vary algorithm. . . . 55
D.14 All main metrics for the number of filters, in the Lee-Theunissen algorithm. . . . 56
D.15 Comparing my 'gains PCA' architecture autoencoders using all main metrics, for the Lee-Theunissen algorithm. The number in the name indicates the number of PCA components used for the input of that autoencoder. . . . 56

D.16 Comparing my ‘no PCA’ architecture autoencoders using all main metrics, for the Lee-Theunissen algorithm. The number in the name indicates the number of nodes in each ‘middle layer’ of the architecture. 56

List of Algorithms

1 Resample a signal $x(t)$ using sinc interpolation from a sample rate of A Hz to B Hz 8
2 Remove integer overflows/underflows (clicks) from a signal array $x(t)$ 17

Appendix K

Project Proposal

An exploration of removing wind noise from mono speech input using DSP and AI approaches

Computer Science Tripos Part II Project
Draft Proposal

13th May 2021

Supervisors:	Dr Dong Ma and Dr Jing Han
Director of Studies:	Dr Robert Harle
Overseers:	Dr Laurence Paulson and Dr Markus Kuhn
Project Originator:	Self originated
Will human participants be used?	Yes

Motivation

When cycling or walking, especially in windy conditions where the wind speed varies heavily, the input audio recorded from a pair of headphones or lavalier microphone is often heavily obscured and indecipherable, especially for speech. This occurs due to a phenomenon whereby air vortices form on the microphone's surface, greatly amplifying the sound of the wind itself [1] [2]. In particular, since some headphones provide microphone feedback when on voice calls, wearing headphones can reduce awareness, since other environmental noises, such as vehicle engines and horns, pedestrian voices and footsteps, are masked by this added wind noise. The presence of wind noise does not provide any additional safety benefit (except maybe to indicate speed) – therefore removing this noise in any capacity should improve safety, especially when reacting to changing road conditions when cycling or walking.

For most consumer microphones, wind noise is a major issue, which is commonly only counteracted using physical means, such as wind-cancelling foam or artificial fur applied to the microphone's surface. However, adding fur greatly increase the bulk of a built-in microphone, and can appear very unsightly to a consumer – hence a software alternative would have some clear benefits here.

Finally, my solutions could apply to many other fields where wind noise is found – for example, pilots often are affected by wind noise whilst flying, especially due to turbulence when taking off and landing. The vast majority of pilots already wear active noise- cancelling headphones, and so a computationally-light algorithm that outperforms existing noise-cancelling algorithms for noise would have a direct application in this sector [3]. Similarly, other modes of transport where wind noise is prevalent – convertible cars or motorbikes for example – could also benefit from these algorithms.

Description of the Work

This project aims to explore existing solutions to remove wind noise from a mono input source, by implementing solutions from research papers, and testing these with a combination of wind and speech noise. Evaluation will also play a key part in this project, with a combination of evaluation methods – including human MOS testing, speech recognition accuracy using WER, SNR, MSE, as well as other sample-wise comparisons – being available to compare each algorithm, and perhaps discover advantages and disadvantages of each.

As a main extension, I will try and create my own model for a wind noise removal algorithm, and then implement this algorithm to hopefully improve accuracy over existing systems.

Description of Data Generation

To reduce the scope of this problem, I will only focus on removing wind noise from speech (from Talkbank's repository of phone call speech) [4]. Wind noise will be collected myself using a dictaphone at night to reduce the amount of environmental noise in this dataset. As an extension, I could also try to selectively remove wind noise from a variety of speech and environmental noises, with the latter obtained from the TAU Urban Acoustic Scenes [5] dataset.

I will then assume that wind noise is additive for the scope of this project, unless there is a clear pattern to suggest otherwise when examining real wind noise (this will allow me to use SNR/MSE easily to determine accuracy). Using this, I can combine these datasets to hopefully mimic a single walker/cyclist taking a phone call outside on a windy day. The piecemeal nature of generating my own wind noise gives me the ability of generating large amounts of training, test and validation data (so long as my artificial test data closely mimics the behaviour of real wind noise on a microphone input). In addition, noise-free audio can also be collected incredibly easily, as removing the wind noise from the audio can be done whilst generating it. Clearly, the input for such an algorithm would be the generated wind noise audio, and the output would be evaluated against this wind noise-removed audio.

Description of Algorithm Implementation

I will aim to implement two particular research papers, one focusing on a DSP solution to wind noise removal, and one focusing on a more AI oriented solution.

The DSP solution I will recreate is from Nelke and Vary [6]. This solution uses wind noise signal centroids (a weighted mean of frequency bands) to extract features and detect the amount of wind noise, calculated using the FFT of the noisy input. The denoised output frequencies are then retreived by weighting frequency bands used for wind noise by the predicted proportion of wind noise present in the signal, and performing a IFFT/overlap-add to retrieve the denoised signal. Although there is a variety of different algorithms and techniques used in this method I feel that the paper itself is clear enough to replicate this method sufficiently, and the method should be simple enough to implement in a reasonable amount of time. In addition, this algorithm performs surprisingly well, and is designed to be computed in real-time.

The AI solution I will implement is from Lee and Theunissen [7]. This solution also uses a similar method of seperating out bands of frequencies (this time using a Gaussian filter on each band of frequencies), then preprocessing these values before inserting them into a shallow three-layer autoencoder. The autoencoder is then trained to produce the desired output. This is a somewhat similar solution to that used in Rhodes [8] except this uses a dual microphone setup, and a different set of preprocessing and optimisation steps. Although it seems harder to reproduce than the above DSP solution, I still feel confident that I can implement this in a reasonable amount of time.

Description of Analysis of Algorithms

This section is split into two parts – the first aspect being to analyse the impact of making changes to the above solutions (such as increasing the number of bands of frequencies, sample rate / sample quantisation, adding layers to the neural network or retraining to work on general environmental noise, as an extension). This will require me to be mindful of how I code my solution, so as to make later adjustments to the architecture as painless as possible (such as by using variables where possible to enable changes to be made in just one place). In addition, I will examine the outputs of each algorithm to determine if any trivial changes can be made to improve accuracy / WER or perceived quality of output.

The second section is taking these algorithms and evaluating their performance, using a variety of evaluation strategies. I have chosen the following evaluation methods to focus on in this project:

- Since I will be analysing speech, I have decided to first use WER with a speech recognition API, on the clean, noisy and the output of each algorithm. This way, I can use the clean recognition accuracy as an upper limit, the noisy recognition accuracy as a lower limit, then compare each algorithm in relation to these factors.
- I will also focus on MOS, a technique using human participants and requesting them to score solutions in terms of audio quality and speech perceptibility/intelligibility. This will be carried out following necessary risk assessments and ethics plans, and (where possible) I will try and follow ITU-T P.800 defining MOS for speech and audio quality estimation, in order to ensure reproducibility.
- Finally, I will use SNR and MSE to compare the clean and algorithm output signals directly, as well as plotting accompanying graphs to detail where these errors occur.
- As an extension, I am also considering using PESQ and PEAQ. PESQ is an algorithm designed to evaluate speech audio quality algorithmically, providing an objective output; PEAQ is an equivalent algorithm that assesses general audio quality. Due to PESQ requiring a license, I will need to use an open-source implementation, which may give me unreliable results compared to other research papers.

Starting Point

Current research

As of present, many denoising algorithms and techniques have been proposed, investigated, and achieved appealing performance in many areas. For example, 2hz.ai have used DNNs on algorithmically-generated noisy speech to remove all background noises from speech, seeing an 1.4 point increase in MOS score over the noisy speech input [9]. Another alternative solution is that employed by Haque and Bhattacharyya, which utilised a variety of linear filtering techniques to remove background noise from speech, with mean squared errors at ≈4% [10]. However, these approaches aim to remove all background noises, rather than specifically targeting a single type of noise.

To remove wind noise in industry (for example, live news shoots) multiple microphone systems are used, with one microphone near the presenter, and a second microphone further away to capture ambient noise. The second microphone's output is then inverted, and added to the first microphone's output, giving an approximation for $(external\ noise + presenter) - (external\ noise)$. This has the disadvantages of requiring multiple microphones in different locations, as well as removing all noise and leaving a less natural clean speech output. Often, fake ambient noise can then be added back into this output to leave a much more aesthetically pleasing output, which by no means represents the real audio being captured on-location. Multiple research papers do focus on multiple-microphone wind noise removal (mainly for industry applications), with Nelke and Vary exploiting the differences in phase spectral density for voices and wind noise to remove wind noise [11]; and researchers at KAIS using Wiener filters to characterise wind noise individually using a second microphone, and using this to remove wind noise from a noisy input[12]. AI approaches have also been explored thoroughly in research, with Keshavarzi et. al. using a recurrent neural network to reduce wind noise in low-power hearing aids [13], and Rhodes using a combined DSP and AI approach to remove wind noise in low power wearables [8]. As for single microphone wind noise removal, Nelke and Vary utilised a DSP approach, through modelling wind noise using signal centroids, and seperating these by comparing their different spectral energy distributions [6]. Nemer and Leblanc also used a DSP approach, instead choosing seperate algorithms and parameters of wind noise removal based on the perceived wind noise resonance [14]. As mentioned in my description above, Lee and Theunissen [7] have also used a similar approach to Rhodes [8], but with a single microphone methodology – utilising DSP preprocessing and a shallow autoencoder architecture.

Current skills

I come to this project having had some experience with music production and audio editing, and thus do have some experience with removing noise from audio. However, I have little experience with applying code to such a problem.

I am already very familiar with Python, having used it in both GCSE and A-level Computer Science, as well as in the Scientific Computing course in Part IA.

As part of my group project in Part IB, I also helped to produce a system in Python which performed optical word recognition on logbooks supplied by the Zoology Museum in Cambridge. I mainly focused on preprocessing photos of logbook pages, which involved binarisation of images, perspective transforms using matrices, and a combination of histogram analysis and a variety of window functions to detect logbook lines (and hence cells), through exploiting the larger number of black pixels in a column/row line. I also helped with converting this data into a format which could be fed into a CNN performing word recognition. As a result, this project familiarised me with Numpy, Matplotlib and (to a much lesser degree) Scipy and Tensorflow. Otherwise, I have only followed rudimentary tutorials on Tensorflow and audio libraries such as librosa – therefore, I expect that some time at the start of the project will be spent becoming acquainted to these tools.

I will also need to do some research on PCA, which is used in the AI architecture I will be implementing [7].

Finally, I will be taking the Digital Signal Processing and Deep Neural Networks courses this year, with the first occuring in Michaelmas and the second in Lent. Therefore, I may also need to spend time reading ahead (or researching topics in each of these courses further) in order to complete this project.

Substance and Structure of the Project

Success Criteria

My project will involve several main parts, from which my core success criteria for this project is formed. These are:

- To examine collected wind noise audio, and to examine several features and data formats of this noise – for example, the shape of waveforms, frequency responses, or spectrograms – to determine what characteristics wind noise possesses.
- To implement two existing wind noise removal techniques, with one using a more traditional predominantly DSP methodology, and one using an AI methodology. My criteria for success here is to achieve $> 60\%$ accuracy with WER, with 100% being set as the score achieved by the clean audio signal.
- To experiment with these existing solutions, by changing parameters and determining their effect (for example, adjusting the sample rate of input files, reducing or increasing the number of layers in a NN to improve performance / accuracy, etc.). This will require me to write very extensible code for both above solutions, such that these changes are easy to perform and reverse.
- To evaluate the above methods, using MOS with human participants, as well as MSE/SNR comparing to clean audio, and WER using a speech recognition AI on noisy speech; and to ensure that my testing methods are detailed to ensure reproducibility of my results.

Optional Extensions

I expect that the core success criteria will be completable within the time provided for this project, and also feel that I will have time to explore additional extension tasks. These are listed here, with particular emphasis on the first starred bullet point.

- ⋆ My main extension aim is to use the techniques and features learned from my research above to create a new similar architecture to remove wind noise, hopefully improving on the accuracy of these above solutions. This is likely to use parts from both implementations explored in my main project, plus any additional features either from existing research or gleamed from features of the wind noise itself. Effort will be made to determine which parts of wind noise can easily be removed using DSP methods, and if there are any other more complex components which might require a shallow neural network to remove.

- I could try and generalise my approach to removing wind noise from a combination of mixed noises, such as environmental noise or white noise in addition to speech. Unlike speech on its own, which can be detected relatively easily by its power spectrum density [6], environmental noises will have a variety of frequencies included in them, shifting the problem from enhancing speech output by removing wind noise, to removing the wind noise without damaging any other parts of the spectrum.

- Another option is to improve the performance of either the existing solutions, or my own solution, such that it can be performed in 'real-time', considered as < 2 seconds; or further still, to try and ensure that lower-power processors (such as those in phones) can compute the algorithm in real-time. This could be done by reducing the number of layers or connections in the NN used, parallelising portions of the DSP algorithms further by utilising more vectorized operations, or by trimming insignificant weights as the network is trained (to obtain a sparse network with similar performance to a dense network) [15].

- Since I am mainly focusing on speech, I could try to optimise my model for voice assistants, by increasing the intelligibility of speech in my output. This may be at the cost of removing more noise than is necessary, or harming output when additional noise is added to the input.

- A final idea is to focus on wind speed. I could collect wind noise along with an anemometer, to record the corresponding wind speeds. With this, I could improve the performance of my algorithm through selecting different algorithms based on speed, or passing approximated wind speed into the algorithm/neural network itself. Since (in the latter case) I would only be adding parameters to my model, the performance of my solution can only improve on the training data provided.

Timetable and Milestones

Below is my projected timeline to complete this project. This timeline takes into account workloads in both Michaelmas and Lent terms, as well as revision time in both Michaelmas and Lent vacations. In addition, it provides some buffer time in the event of me falling behind schedule: due to the uncertainty surrounding Coronavirus; or due to balancing workloads between the project, supervisions and other courses. I have also decided to try and write the dissertation partially as I work – this ensures that I remember to elaborate on any decisions made in the preparation and implementation phases, which as a result should make any evaluation much easier to write. If this ends up not being possible, then I can utilise some of the extension time left near the end of the project for finishing the dissertation itself.

Slot 1 - *6th July – 25th September*

Initial preparation:

- Brainstorming project ideas
- Finding supervisors and exploring other project ideas
- Starting preliminary research on wind noise, determining if the idea is feasible to implement in the time provided, finding prior research and ensuring that this problem has been tackled before in some capacity
- Finalising the core topic of my project – wind noise removal using DSP/AI
- Finding supervisors for my project idea
- Finalising choice of supervisors

Milestones: Finalising project idea, finalizing choice of supervisors, ensuring feasibility of project.

Slot 2 - *26th September – 9th October*

Preparation:

- Discussions with supervisors on scope of project
- Additional research on previous solutions to project

Important dates and deadlines

- Upload preliminary project information onto Moodle – Friday 9th October, 12 noon.

Slot 3 - *10th October – 23rd October*

Writing project proposal:

- Write 100 words on the core idea of my project

- Work on, and finish core project proposal
- Submit ethics forms and complete necessary risk assessments to enable me to use human MOS testing in my project
- Produce a clean list of papers and websites which will aid the implementation and preparation sections of my project
- Familiarise myself with all prepatory knowledge for my project, including empirical and quantitative evaluation methods, and any additional DSP/AI strategies that may help with implementation

Important dates and deadlines

- Phase 1 Project Selection Status Report due – Monday 12th October, 3:00pm.
- Draft Project Proposal due – Friday 16th October, 12 noon.
- **Project Proposal deadline – Friday 23rd October, 12 noon.**

Milestones: Completing and submitting project proposal and accompanying paperwork, broadening knowledge of project area ready for analysis and implementation.

Slot 4 - *24th October – 6th November*

Setup, preparation and analysis:

- Learn how to use Tensorflow, numpy, scipy and any required audio libraries (audioread, librosa, etc.)
- Practice implementing basic DSP and AI models to improve my skills
- Collect ≈2 hours of real wind noise audio using two microphones, one with wind-shielding fur and one without – these correlated audio streams can then be examined and compared to spot features more easily
- Use correlation to sync up these two separate wind noise audio files ready for analysis
- Get the difference of the raw wind noise audio and wind-shielded audio, to get wind noise (and any artefacts) in isolation
- Begin analysing this wind noise data to spot patterns and features which could be exploited in a wind-removal algorithm – for example, analysing the phase spectrum density, amplitude levels, wave shapes, average duration of gusts, maximum amplitudes of wind gusts etc.
- Collect a large amount of pure wind noise audio for use in creating the test/training data

Milestones: Competency in Tensorflow and audio processing libraries, acquired real wind noise, aim to provide diagrams and rough analysis of wind noise to supervisors by the end of this period.

Slot 5 - *7th November – 20th November*

Analysis and preparation section writeup:

- Continue to, and finish, analysing the raw wind noise
- Produce a clear set of features which could be effectively exploited to remove wind noise
- Begin to write the preparation section by adding any evaluation methods, AI/DSP terms and assumed knowledge, ethical or legal concerns including licenses on datasets etc. which directly impact my project, as well as including the features of wind noise / speech explored above

Milestones: Begin writing the preparation section in LaTeX, produce a theoretical model for wind noise based on observed features.

Slot 6 - *21st November – 4th December*

Writeup and begin implementation of DSP algorithm

- Write an algorithm which automatically generates datasets with and without wind noise, given datasets for wind noise and speech
- Generate ≈30 hours of wind noise in preparation for training neural networks – split these into separate test, training and validation datasets
- Complete a first draft of the preparation section of the dissertation
- Begin implementing the first DSP algorithm, utilising vectorised operations where possible to improve performance

Milestones: Implement artificial wind noise generation and generate test data, continue to write up findings, begin to write first DSP algorithm.

Slot 7 - *5th December – 18th December*

Complete implementation of DSP algorithm and implement evaluation algorithms

- Finish implementing the DSP algorithm, test using artificially generated audio informally to ensure that this has been implemented correctly
- Write up any equations used in the DSP algorithm within LaTeX
- Write all core evaluation code, including a reproducible test platform for MOS tests with human participants, an implementation of WER using Google's Speech To Text APIs, and SNR/MSE implementations, each with graph printing functionality

Milestones: Completed DSP algorithm, finished evaluation algorithms.

Slot 8 - *19th December – 1st January*

Begin implementing the AI algorithm plus catchup **(slack weeks due to Christmas and New Year)**

- Catch up on any work not yet done from previous weeks
- Begin implementing the AI algorithm, whilst making notes of weights, training functions, and any data preprocessing required for input into the neural network

Milestones: Progress has been made on the AI algorithm

Slot 9 - *2nd January – 15th January*

Finish AI algorithm, add to dissertation **(slack weeks to catch up from previous weeks)**

- Finish the AI algorithm implemented here, and continue to make notes during this process
- Add to (and complete) the core implementation section to discuss changes made here

Milestones: Both core solutions are completed, implementation draft is complete.

Slot 10 - *16th January – 29th January*

Test my solution quantitatively against existing solutions, begin writing up evaluation section of project.

- Compare each of the three implemented algorithms, plotting graphs and recording results for each evaluation method used
- Analyse the output audio signals of each algorithm, in a similar way to the original wind noise – determine if there are any features present to the errors and if they can easily be filtered or removed to easily improve accuracy
- Make any changes which trivially improve accuracy to our algorithms
- Begin to write up the evaluation section of the project, making both quantitative and qualitative deductions based on the data collected

Milestones: The evaluation section is on the way to completion, testing is mostly complete (ignoring MOS testing).

Slot 11 - *30th January – 12th February*

Finish evaluation section, create MOS consent forms and copy dissertation to a presentation format using LaTeX or Powerpoint **(slack weeks due to catchup time)**

- Try to complete a first draft of the dissertation paper, including drafts of the preparation, implementation and evaluation sections
- Polish any test platforms for human MOS tests, and create consent forms / briefing sheets in preparation
- Transfer existing quantitative evaluation notes onto a presentation for Progress Report Presentations

Important dates and deadlines

- **Progress Report Deadline – Friday 5th February, 12 noon.**
- Progress Report Presentations – Thursday 11th or Friday 12th February, 2pm

Milestones: Dissertation first draft is completed, presentation is ready and/or has been completed.

Slot 12 - *13th February – 26th February*

Catch up period, work on extensions and polishing dissertation

- Continue to write the dissertation, actively seeking review cycles from supervisors
- Finalise all paperwork required to begin human trials, set a time to complete human trials
- **At some point before Easter vacation, perform human trials for MOS and add these results to the evaluation section**
- Begin working on extension tasks if on schedule

Important dates and deadlines

- Progress Report Presentations – Monday 15th or Tuesday 16th February, 2pm.

Milestones: Beginning to work on extension tasks, MOS trials are in progress

Slot 13 - *27th February – 12th March*

Finalise introduction and preparation section, continue to work on extension tasks and MOS

- Finalise introduction and preparation section and deliver to supervisors for comments on these sections
- Continue to work on extensions (if on schedule), and add to dissertation sections where appropriate
- **At some point before Easter vacation, perform human trials for MOS and add these results to the evaluation section**

Milestones: Working on extension tasks, introduction and preparation sections of dissertation complete

Slot 14 - *13th March – 26th March*

Finalise implementation section, continue to work on extension tasks and MOS

- Finalise implementation section and deliver to supervisors for final comments (unless added to due to extensions)
- Continue to work on extensions (if on schedule), and add to dissertation sections where appropriate
- **At some point before Easter vacation, perform human trials for MOS and add these results to the evaluation section**

Milestones: Working on extension tasks, implementation section of dissertation complete

Slot 15 - *27th March – 9th April*

Finalise evaluation and conclusions section, continue to work on extension tasks, begin to polish entire dissertation

- Finalise evaluation and conclusions section and deliver to supervisors for final comments (unless added to due to extensions)
- Continue to work on extensions (if on schedule), and add to dissertation sections where appropriate

Milestones: Working on extension tasks, dissertation core draft finished (except for extensions)

Slot 16 - *10th April – 23rd April*

Finish extension tasks, aim to finish dissertation

- Continue to work on extensions (if on schedule), and add to dissertation sections where appropriate
- Finish any outstanding extensions, and add any new comments to the corresponding sections in the dissertation
- Clean up any source code that is untidy or unclear as to its true meaning
- Polish entire dissertation as one large unit, ensuring that each section links and flows, and that all references are attributed correctly; seek a review of my dissertation from all parties, and make alterations according to feedback

Milestones: Finished extension tasks, polish dissertation.

Slot 17 - *24th April – 7th May*

Finalise dissertation and source code

- Finalise dissertation, address comments from supervisors/overseers and request further comments from supervisors
- Finish cleaning up any source code that is untidy or unclear

Milestones: Dissertation and source code completed

Slot 18 - *8th May – 14th May*

Catch up time, submit dissertation and source code.

- Finalise dissertation implementation and evaluation sections

Important dates and deadlines

- **Dissertation Deadline – 14th May, 12 noon.**
- **Source Code Deadline – 14th May, 5:00pm.**

Milestones: Dissertation and source code submitted

Resources Declaration

I will require the following software and libraries in order to successfully complete this project:

- Python 3.4+ for the main implementation of my solution
- Tensorflow/Keras for coding the neural networks
- Scipy, Numpy, librosa, and perhaps other audio processing libraries in order to perform digital signal preprocessing on the input stream
- MatPlotLib to plot graphs and help evaluate my test system
- Any other libraries required for implementation or (if I decide to use PESQ/PEAQ as an extension) evaluation
- PyCharm IDE to help me debug, implement and test my solution
- $<$ 20GB of storage in order to store generated test data, as well as any collected wind noise data and any noise banks (i.e. wind noise, traffic noise, voices) which will be used to generate the test data in the first place
- CUDA and associated libraries to dramatically improve the speed of my neural network training
- Two lavalier microphones (which I already own) to record side-by-side wind noise data, with and without wind cancelling foam applied to the microphone. This could be used both for testing, as well as evaluating my solution in a more complex real-world scenario. In addition, this will give me the data required to analyse wind noise in isolation, and determine its properties to generate artificial wind noise

As an extension, I may also need an anemometer (also already purchased) to record wind noise speeds, if I go forward with using wind speed data to improve my model.

(I accept full responsibility for this use of any specialist hardware (e.g. anemometer, external lavalier microphones) and have made contingency plans to protect myself against hardware failure.)

Due to the libraries and training time that may be required for this project, I am planning on using my own computer, which has the following specifications: This will allow me

Operating system:	64-bit Windows 10
CPU:	Intel Core i7 7700k (4 cores, 8 threads)
GPU:	Nvidia GeForce GTX 1080Ti
RAM:	16 GB
Storage:	3.5 TB SSDs ($\approx$2 TB free)

to much more easily implement and test my solution, using CUDA to train my neural network solution efficiently.

(I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.)

I also have a secondary laptop PC, whose specifications are:

Operating system:	64-bit Windows 10
CPU:	Intel Core i7 8550u (4 cores, 4 threads)
GPU:	Nvidia GeForce MX150
RAM:	16 GB
Storage:	512GB SSD (≈150 GB free)

(I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.)

In the case of data loss, I will be keeping backups of my work on two separate external HDDs, as well as making cloud backups to GitHub, as well as another cloud service (such as Google Drive). This will also give me version control in the scenario where I am working on both computers. This has already been set up in preparation for the project.

In the case of software failure, I will either move onto my laptop, or can restore my computer from regular backups on an additional 4TB external HDD.

In the case of hardware failure, I could either use my laptop, or will have the financial means to either replace failed hardware components or purchase a new computer to similar or better specs to my current desktop.

References

[1] Chris Woolf. "Characterization and Measurement of Wind Noise around Microphones". In: *Audio Engineering Society Convention 140*. 2016. URL: http://www.aes.org/e-lib/browse.cfm?elib=18194.

[2] J. A. Zakis. In: *J Acoust Soc Am* 129.6 (2011), pp. 3897–3907.

[3] FAA. *HEARING AND NOISE IN AVIATION*. URL: https://www.faa.gov/pilots/safety/pilotsafetybrochures/media/hearing.pdf (visited on 16/10/2020).

[4] TalkBank. *CABank*. URL: https://ca.talkbank.org/access/ (visited on 16/10/2020).

[5] DCASE. *DCASE 2019 Challenge*. URL: http://dcase.community/challenge2019/task-acoustic-scene-classification (visited on 16/10/2020).

[6] Christoph Matthias Nelke et al. "Single Microphone Wind Noise PSD Estimation Using Signal Centroids". In: *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*. 2014. URL: http://ikspub.iks.rwth-aachen.de/pdfs/nelke14.pdf.

[7] Tyler Lee and Frédéric Theunissen. "A single microphone noise reduction algorithm based on the detection and reconstruction of spectro-temporal features". In: *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 471.2184 (2015), p. 20150309. DOI: 10.1098/rspa.2015.0309. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rspa.2015.0309. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rspa.2015.0309.

[8] Anthony Rhodes. "Real-Time Wind Noise Detection and Suppression with Neural-Based Signal Reconstruction for Multi-Channel, Low-Power Devices". In: (Sept. 2017). URL: https://arxiv.org/ftp/arxiv/papers/1710/1710.00082.pdf.

[9] Davit Baghdasaryan. *Real-Time Noise Suppression Using Deep Learning*. 2018. URL: https://developer.nvidia.com/blog/nvidia-real-time-noise-suppression-deep-learning/ (visited on 16/10/2020).

[10] Minajul Haque and Kaustubh Bhattacharyya. "Speech Background Noise Removal Using Different Linear Filtering Techniques". In: Jan. 2018, pp. 297–307. ISBN: 978-981-10-8239-9. DOI: 10.1007/978-981-10-8240-5_33.

[11] Christoph Matthias Nelke and Peter Vary. "Dual Microphone Wind Noise Reduction by Exploiting the Complex Coherence". In: *ITG-Fachtagung Sprachkommunikation* (2014).

[12] Jinuk Park et al. "Coherence-based Dual Microphone Wind Noise Reduction by Wiener Filtering". In: Nov. 2016, pp. 170–172. DOI: 10.1145/3015166.3015206.

[13] Mahmoud Keshavarzi et al. "Use of a Deep Recurrent Neural Network to Reduce Wind Noise: Effects on Judged Speech Intelligibility and Sound Quality". In: *Trends in Hearing* 22 (2018), p. 2331216518770964. DOI: 10.1177/2331216518770964. eprint:

https : / / doi . org / 10 . 1177 / 2331216518770964. URL: https://doi.org/10.1177/2331216518770964.

[14] Elias Nemer and Wilf Leblanc. "Single-microphone wind noise reduction by adaptive postfiltering". In: Nov. 2009, pp. 177 –180. DOI: 10.1109/ASPAA.2009.5346518.

[15] *Trim insignificant weights: TensorFlow Model Optimization*. URL: https://www.tensorflow.org/model_optimization/guide/pruning.