



Audio Narration Application

by

Philip Halloran

This Report is submitted in partial fulfilment of the requirements of the Honours
Degree in Computer and Communications Engineering (DT081) of the Dublin
Institute of Technology

May 29th, 2017

Supervisor: Dr. Kevin Tiernan

DECLARATION

I, the undersigned, declare that this report is entirely my own written work, except where otherwise accredited, and that it has not been submitted for a degree or other award to any other university or institution.

Philip Ifalloran

May 29th, 2017

Acknowledgements

I would like to deeply thank Dr. Kevin Tiernan. Throughout the final two years at DIT, Kevin has been an inspirational lecturer in the field of DSP. I found Kevin to be an excellent communicator of fine details for difficult, sometimes abstract, concepts. Working with Kevin was always a pleasure, his guidance and mentoring as a project supervisor was invaluable.

Abstract

An audio narration application was proposed to assist in the creation of audio files that can be used to narrate training or lecture documents. Research into teaching methods has found that students with visual impairments or high-incidence cognitive disabilities such as dyslexia and reading difficulties can acquire information in a more efficient manner using audio-assisted reading rather than by conventional teaching methods [1]. The application will assist a teacher editing an audio file into sentences pages and paragraphs by creating a metadata file to accompany the narration file.

Key features of the project include spectral noise analysis to automatically set filtering bands to improve sound quality and set an active threshold for speech endpoint detection. The location of the starting points of each spoken sentence are automatically stored in a metafile. The developed application will also remove file redundancy, caused by the narrator pausing briefly, by extracting excessive long silences gaps. The software provides facilities for editing the metafile so that additional information about page and paragraph boundaries may be added.

The Python and C programming language were used to develop the project's underlying technology. Python has become a popular language for numerical analysis. It is a dynamically-typed, general purpose language. Python is supported with many libraries for array and matrix manipulation; image processing, digital signal processing, data exploration and visualization tools [2]. Python's strength lies in its ability to integrate scientific computing with general purpose computing. C was employed for time-critical parts of the project where Python's vector code proved to be infeasible. The C code was then interfaced to Python using the ctypes module.

Abbreviations

ADC Analogue to Digital Converter

F_s Sampling Frequency

FIR Finite Impulse Response

IIR Infinite Impulse Response

SNR Signal to Noise Ratio

V_{Ref} Reference Voltage

RMS Root Mean Square

ABS Absolute Value

CLI Command Line interface

GUI Graphical User Interface

.so Linux Share Object File

.dll Windows Dynamic Link Library

Table of Contents

Acknowledgements.....	ii
Abstract	iii
Abbreviations.....	iv
List of Figures	2
List of Tables	2
Chapter 1. Introduction	3
1.1 Project Scope	3
1.2 Ethical Framework	5
1.2.1 Accountability and Responsibility	5
1.2.2 Human Interaction and Professional Conduct	5
1.2.3 Respect for Property and the Environment	5
1.3 Project Management.....	5
1.3.1 Project Logbook.....	6
1.3.2 Software Requirements	6
1.3.3 Software Engineering Methodology & Deliverables	7
Chapter 2. Literature Review	8
2.1 Speech Endpoint Detection Algorithms and Techniques	8
2.1.1 Zero-Crossing Technique	9
2.1.2 Spectral Entropy Technique	10
2.3 Digital Filtering.....	11
2.3.1 Transfer Function using the Z Transform	12
2.3.2 FIR Filters	13
2.3.3 IIR Filters	13
2.4 Vocal Tract Anatomy and Speech Production	14
2.4.1 Phonemes	15
2.4.2 Vowel Formation	15
2.4.3 Fricative Formations	16
2.4.4 Stop Formations	16
Chapter 3. System Implementation.....	17
3.1 Command Line Interface	18
3.2. DSP System Design.....	20
3.3 Noise Analysis.....	21
3.3.1 Determining Voice Sounds Threshold Level	21
3.3.2 Design of noise reducing filter	22

3.3.3 Band Energy Segmentation	22
3.3.4 Block Processing Stages	23
3.4 Sentence Segmentation.....	25
3.3.1 Shared Object File.....	25
3.4.2 Start of spoken word – positive threshold transition.....	26
3.4.3 End of spoken word – negative threshold transition	27
3.4.4 Python Wrapper for Shared Object File and Duration analysis	27
3.5 Metafile Creation	30
Chapter 4. Testing and Results	32
4.1 Test environment.....	32
4.1.1 Software Dependencies	32
4.2 Noise characteristics	32
4.3 CLI Testing	33
4.4 Benchmarking.....	34
4.5 Sentence segmentation Testing	35
4.6 Block Processing	36
4.7 Application Limitations.....	37
4.8 Future Development	37
Chapter 5. Conclusion.....	38
References	39
Appendices	40
Appendix 1 Work Breakdown Structure.....	40
Appendix 2 Sampling	41
Appendix 3 Digital Filters	43
FIR Filters.....	43
IIR Filters	44
Appendix 4 CLI Screenshots	45
Start Screen	45
Main Menu	45
Play Time Segment.....	45
Play Sentence	46
Mark Page	46
Mark Paragraph	46
Purge Redundancy	47

List of Figures

Figure 1: Fricative and Stop Phoneme Durations	4
Figure 2: Thresholding and timing analysis on the absolute signal level.....	4
Figure 3: Prototyping model, each software development phase marks a project deliverable	7
Figure 4: Zero-Crossing For Different SNR.....	9
Figure 5: Elliptic Low-Pass Filter Response	11
Figure 6: Vocal Tract Structures (nidcd.nih.gov).....	14
Figure 7: Oral Speech Production Structures.....	15
Figure 8: Vowel formation samples.....	15
Figure 9: Fricative formations	16
Figure 10: Stop formations	16
Figure 11: Start screen	18
Figure 12: Main Menu	19
Figure 13: User input 3 selected – Play sentence	19
Figure 14: System Block Diagram For DSP Elements.....	20
Figure 15: Shows Band Energy Segmentation	22
Figure 16: DC offset time Domain	23
Figure 17: DC offset Frequency Domain.....	23
Figure 18: High-Pass filtering at different cut frequencies	24
Figure 19: Absolute value of audio signal before and after recursive filtering	24
Figure 20: Smoothed Signal with threshold	24
Figure 21: Extracting sentence endpoints and mapping back to audio signal	29
Figure 22: Block Processing Discontinuities	36

List of Tables

Table 1: Stop Durations	16
Table 2: CLI Test Plan	33
Table 3: Benchmarking Results	34
Table 4: Sentence Segmentation Durations	35

Chapter 1. Introduction

1.1 Project Scope

The purpose of the audio narration application is to assist a lecturer in the creation of audio files by creating a metafile with information on the end of sentences, paragraphs, and pages. Audio files that are long enough for lecturing or instructional documents contain large numbers of samples. Typical sample sizes for a test audio file with thirty-minute duration range from 14,400,000 samples at a sampling frequency of 8 kHz to 79,380,000 samples at 44.1 kHz. Such large datasets mean that processing large files as a single segment is problematic. A personal computer has finite resources with which to process data. Techniques such as caching and page tabling for virtual memory have been developed to enable large quantities of data to be processed but the level of support for these techniques depends on the cost of the underlying PC.

Block processing is a technique that is used to overcome resource limitations. The method processes data in blocks of a pre-specified length that divides equally into the full audio file. The technique employs a sequential processing of data that is analogous to a sliding window moving across the full length of the file. Block processing reads data directly from disk, thus removing excessive dependency on vital system resources such as RAM cache and CPUs. The implementation of block processing adds an extra layer of complexity to signal processing. The inter-block transitions cause discontinuities in the signal that manifest as an audible ‘click’ in the processed data. To eliminate the signal discontinuities, the final conditions of a processed block need to be provided as the initial conditions to the next block.

Endpoint detection is used in speech recognition for automotive control [3] and the efficient use of communications channels [4]. In this project, endpoint detection was used to segment sentences, which is a simpler subset of the general speech recognition problem of segmenting voice phonemes. Speech can be classified by labelling the boundaries (endpoints), or by modelling the signal as differing states (Voice Activity Detection) [4]. The approach used to perform sentence segmentation involves comparing the speech level to a threshold value. Signal levels above the threshold are identified as speech, levels below the threshold are identified as silence. The threshold is scaled from spectral energy analysis that is performed on a slice of audio data. The result from spectral analysis acts as a scaling factor on the mean value of the absolute signal level. Threshold scaling is kept intentionally low to ensure that

the maximum speech is extracted from the noise background. This is possible because of the pre-emphasis filtering stages in the block processing loop. Speech signals are highly complex to analyse. Stop and fricative phonemes result in reduced speech levels as air flow is stopped or constricted as illustrated. This can give rise to a false reading of speech boundaries.

To overcome the problem of false boundary detection a time duration analysis is performed. This augments the threshold level detection by measuring the duration of silences. Fricative durations range from 50 to 200 milliseconds whereas stop durations range from 12 to 74 milliseconds [5] so short silent regions are ignored in the sentence segmentation analysis. The timing analysis is additionally configured to extract inter-word periods below a specified duration. The indices of all threshold detected endpoints are measured against a minimum duration (350ms was found to perform best in testing See Table 4), if they are greater than the duration, they are kept. Otherwise they are discarded.

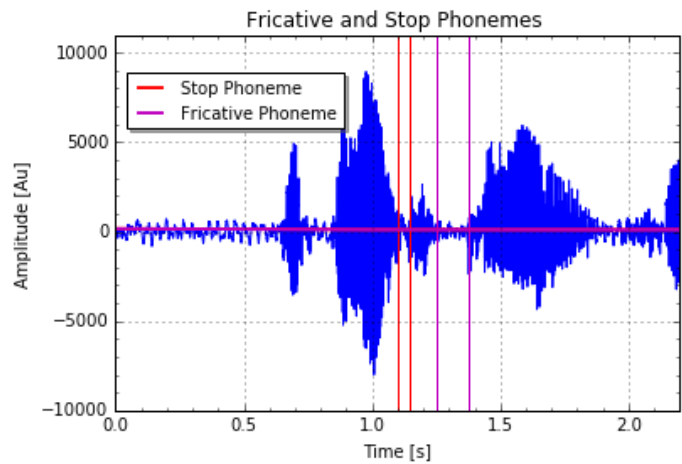


Figure 1: Fricative and Stop Phoneme Durations

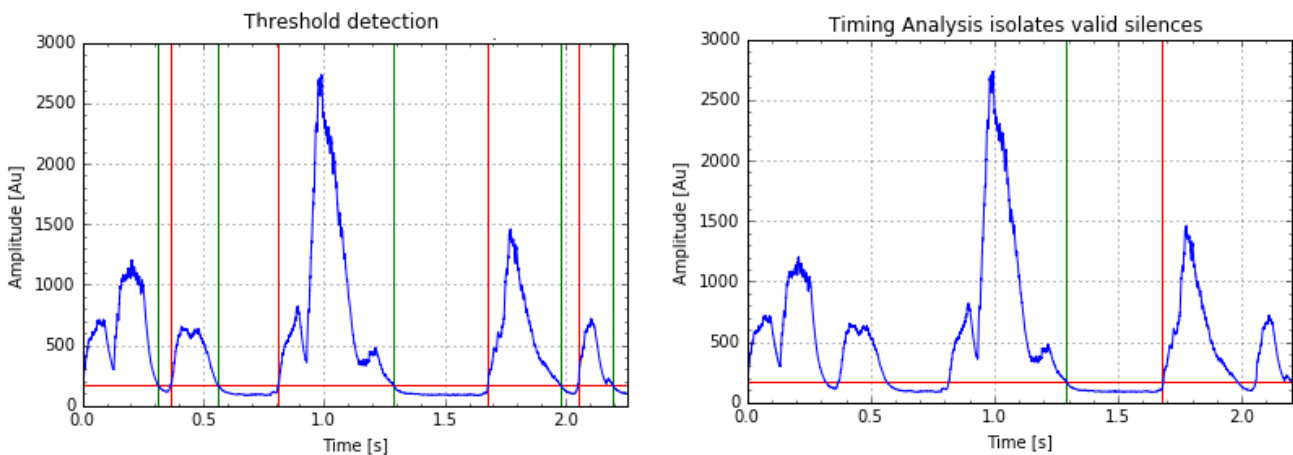


Figure 2: Thresholding and timing analysis on the absolute signal level

1.2 Ethical Framework

It is the author's responsibility as an engineering student, and as a future engineer, to act within an ethical framework. Consideration of the project's current and future implications was maintained throughout the development process. The codes of ethics from Engineers Ireland [6] and the Institute of Electrical and Electronic Engineers (IEEE) [7] served as a structure for the development of a code of ethics for this project and future projects in the author's career.

1.2.1 Accountability and Responsibility

The author ensured that industry standards are adhered to and patents are respected at all opportunities. The licensing of any technologies developed in this project would be restricted to companies that can display a responsible ethical structure. During the project, there was an ongoing evaluation to ensure that there will be limited scope for the misuse of the developed technology.

1.2.2 Human Interaction and Professional Conduct

During the project development, the author sought to maintain a respectful engagement with all persons and did not engage in any discriminatory act. All persons were treated fairly and respectfully in accordance with their professional standing.

1.2.3 Respect for Property and the Environment

During the process of this project's development Dublin Institute of Technology (DIT) provided facilities including laboratories, books, and computing equipment. The author ensured that all equipment and work areas were left in an appropriate manner. The author ensured that the work that was undertaken in testing, development and in final design did not adversely affect the environment or the health and safety of other persons.

1.3 Project Management

In the preliminary stage of the application's development the author performed a provisional requirements analysis as part of an initial research program. This was undertaken to evaluate the individual aspects of the application development, the information obtained formed the basis for a work breakdown structure. The author specified that the duration of the sub-tasks defined in the Work Breakdown Structure were provisional as the application developer had limited experience in developing an end-to-end application of this nature. A work breakdown structure (See appendix 1) was used to decompose the project into a series of tasks and sub-

tasks. The top line of the structure is the project title; the second line is the task headings and the subsequent lines are the sub-tasks associated with the headings. Work breakdown structures are used to divide the project into modular sub-tasks that are inter-related. This information was then used to determine a timeline and predecessor tasks both within modules and between modules. The evaluation resulted in a Gantt chart which showed the sequencing of the tasks. The full extent of the work involved in the individual subtasks of the project wouldn't be known until further research and testing was carried out.

1.3.1 Project Logbook

During the project development, the author maintained a weekly progression log.

The log gave perspective and a direction of travel for the week ahead. In addition, the logbook was used as a reference for the final report; a record for ongoing analysis, and as a guideline for project management. The author recorded the minutes from the meetings with the project supervisor. The minutes were an invaluable resource for writing the final report.

1.3.2 Software Requirements

In the preliminary stages of this project, primary and secondary research uncovered several key elements to the design and implementation:

- High-speed, adaptable speech endpoint detection: The application must apply block processing techniques allied with strict memory management to efficiently process the large volumes of data in an acceptable timeframe.
- Automated file management and format conversion from various formats including wav, ogg, aac and mp3: The application should not be overtly restricted by file formatting constraints.
- A user-friendly interactive interface: A software application should be user-friendly and intuitive to use. When designing software applications, a conscious effort must be made to design and maintain an intuitive system that is acceptable for end users with varying levels of technical proficiency.

1.3.3 Software Engineering Methodology & Deliverables

The Prototyping model was selected for this project. The prototyping model is an iterative approach that delivers increased functionality in a series of development phases. This method is best suited to software development projects where the full requirements are not known in the initial design phase. The project required twelve prototyping loops to deliver a working system.

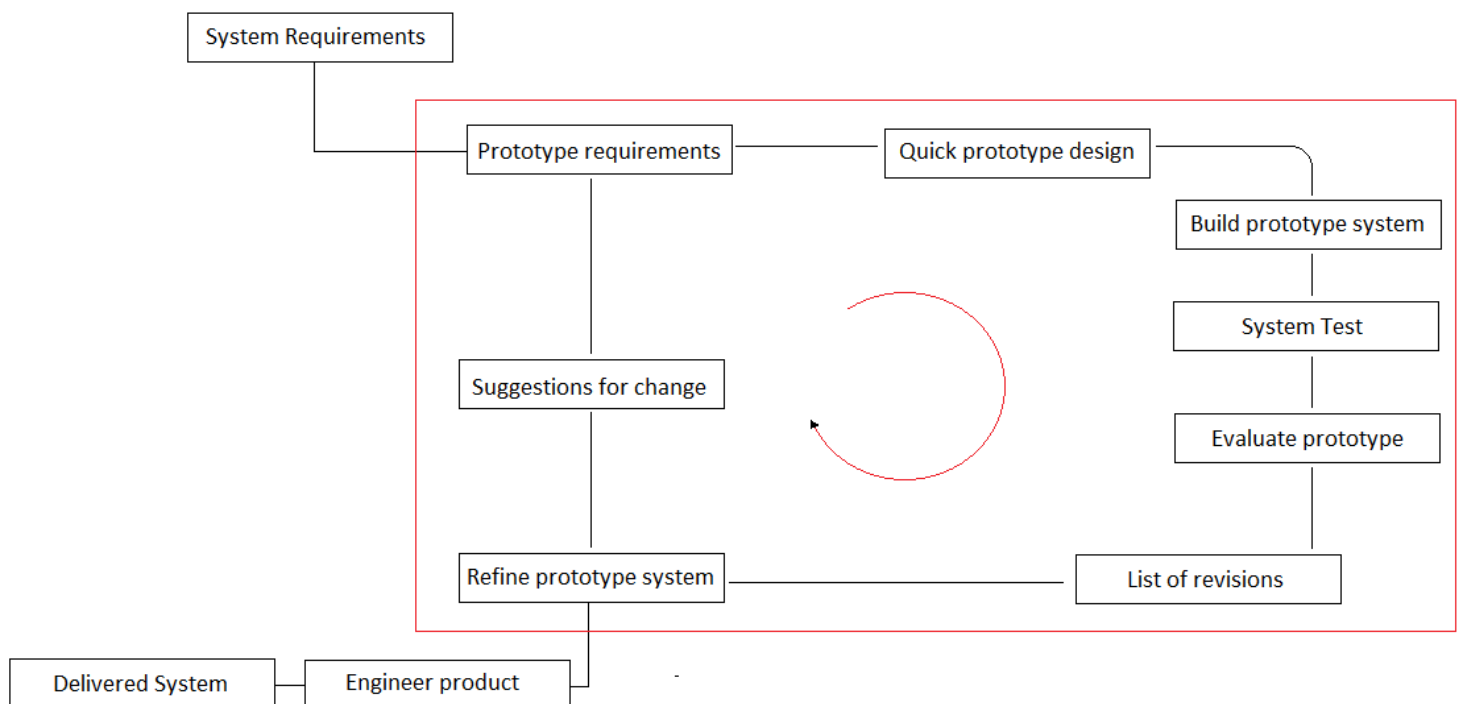


Figure 3: Prototyping model, each software development phase marks a project deliverable

This project was unique in the author's experience of software development. In the initial four phases, there was a need to develop a foundation of digital signal processing functions to calibrate plot functions with synthesised test signals. The initial phases caused a two-week delay in the start of the project, which was difficult to recover. Additional overruns included the investigation and testing of C shared object files to increase performance and development and testing for spectral noise estimation.

Chapter 2. Literature Review

To fully understand the requirements for producing an audio narration application it was necessary to investigate existing technologies and methods that are currently in use. This research focused on key areas that have been identified as being pivotal underpinnings of the application. The areas identified were

1. Speech endpoint detection algorithms.
2. Digital filtering techniques.
3. Vocal tract anatomy and speech production.

The research performed at this stage of the project would give a direction of travel and serve as a template for a development of understanding of the challenges that would need to be overcome to produce a working application.

2.1 Speech Endpoint Detection Algorithms and Techniques

Endpoint detection is the process used to determine the boundaries of speech and non-speech within an audio signal. Accurate endpoint detection has been an active research area over several decades [8]. The motivation for endpoint detection is primarily used for speech recognition in applications such as search engines and automotive infotainment systems. Numerous strategies have been developed to isolate speech utterances in background, noise-contaminated, silences. Zero crossing rate algorithms generally exhibit favourable performance in low noise level environments but their effectiveness drops off rapidly as the signal to noise ratio (SNR) reduces. Spectral entropy is a technique that attempts to rectify the performance limitations of zero crossing rate algorithms in low SNR environments. Sub-band, spectral entropy ratios are evaluated and applied to evaluate speech boundaries in noisy environments [9]. The author will explain these techniques to contextualise the approach developed throughout the project. The algorithms above are tested using at least two types of noise:

1. White noise
2. Pink Noise

White noise has equal energy across the full range of frequencies. As a result, white noise is difficult to filter using static filtering techniques.

Pink noise has higher energy at low frequencies. The power spectral density levels roll-off at a rate of $1/F$, thus the highest levels of noise power are found at lower frequencies. Pink noise responds better than white noise to static filtering techniques.

2.1.1 Zero-Crossing Technique

Zero-crossing rate is a technique that counts the number of times that the sampled audio file changes sign from positive to negative. The rate of change of sign gives an indication of the relative amplitudes of the signal. Speech will have a high relative amplitude and silence will have a low relative amplitude.

The mathematical description of the Zero-Crossing algorithm is defined as

$$\text{Rate}_{\text{ZC}} = \frac{1}{2} \sum_{n=0}^{N-1} |\text{sgn}[x(n)] - \text{sgn}[x(n-1)]|$$

Where

$$\text{sgn}[x(n)] = \begin{cases} 1, & x(n) \geq 0 \\ -1, & x(n) < 0 \end{cases}$$

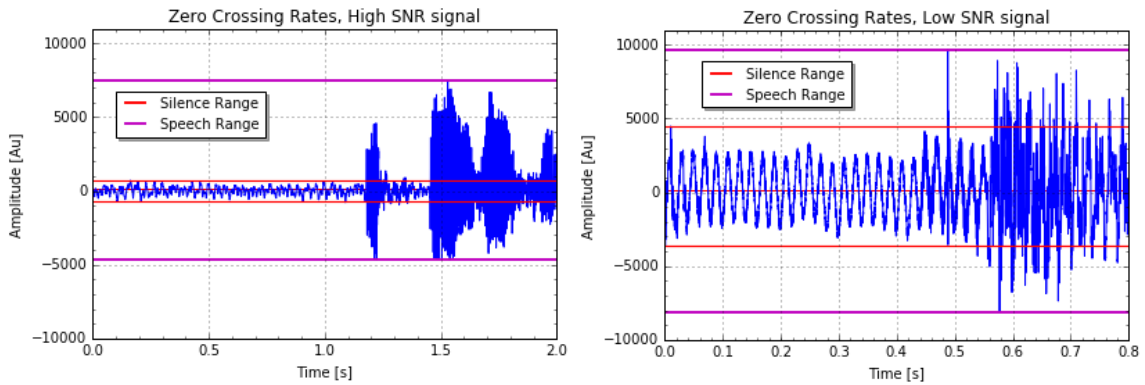


Figure 4: Zero-Crossing For Different SNR

Zero crossing rate works well for audio signal with a high Signal to Noise Ratio (SNR) but performance drops-off as the SNR decreases. The lack of performance is due to noise level approaching the level of the utterance, thus the crossing rates approach equality.

2.1.2 Spectral Entropy Technique

Signal entropy is a technique that is commonly used in information theory. In communication systems, there is a direct link between probability and information. If a symbol in a communication system is likely to occur that is considered to have little information and conversely, if a symbol is unlikely to occur it is considered as having a high level of information [10]. Spectral entropy is a technique that uses probabilities to discern between different levels of a signal's spectrum thus describing the complexity of a system. It can be used to capture the formants in the vocal distribution of speech signals.

The mathematical description of the Spectral Entropy algorithm is defined as

Calculate the signal segment spectrum using the Discrete Fourier Transform (DFT)

$$\mathbf{X}(\mathbf{k}) = \sum_{n=0}^{N-1} \mathbf{x}[n] e^{-j \frac{2\pi}{N} n \mathbf{k}}$$

Calculate the Power Spectral Density (PSD) of the audio signal segment, and normalise by the number of bins.

$$\mathbf{P_x}(\mathbf{k}) = \frac{1}{N} |\mathbf{X}(\mathbf{k})|^2$$

Normalise the PSD so it can be used as a Probability Density Function

$$\mathbf{P_i} = \frac{\mathbf{P_x}(\mathbf{k})}{\sum \mathbf{P_x}(\mathbf{k})}$$

The Power Spectral Entropy (PSE) is then ascertained from the general entropy equation

$$\mathbf{P_{SE}} = - \sum_{i=1}^n \mathbf{P_i} \ln \mathbf{P_i}$$

2.3 Digital Filtering

Digital filters are software defined structures that numerically transform a set of sampled input numbers, $x[n]$ (See appendix 2 - Sampling), to produce a filtered set of output numbers, $y[n]$. Digital filters are implemented in the form of a difference equations that evaluate to the weighed sum of the current and previous input numbers [12].

The digital filter is computed as a sum of products. Outputs $y[n]$ for inputs $x[n]$ and previous outputs $y[n]$.

$$y[n] = \sum_{i=0}^N b_i x[n-i] - \sum_{i=0}^N a_i y[n-i]$$

The values b, a represent the filter's coefficients. $b[0, \dots, N]$ are the non-recursive coefficients and $a[0, \dots, N]$ are the recursive coefficients. N represents the order of the filter, thus for larger N there is a large computational burden but a more favourable reduced transition band due to a steeper roll-off in the filter's frequency response [13].

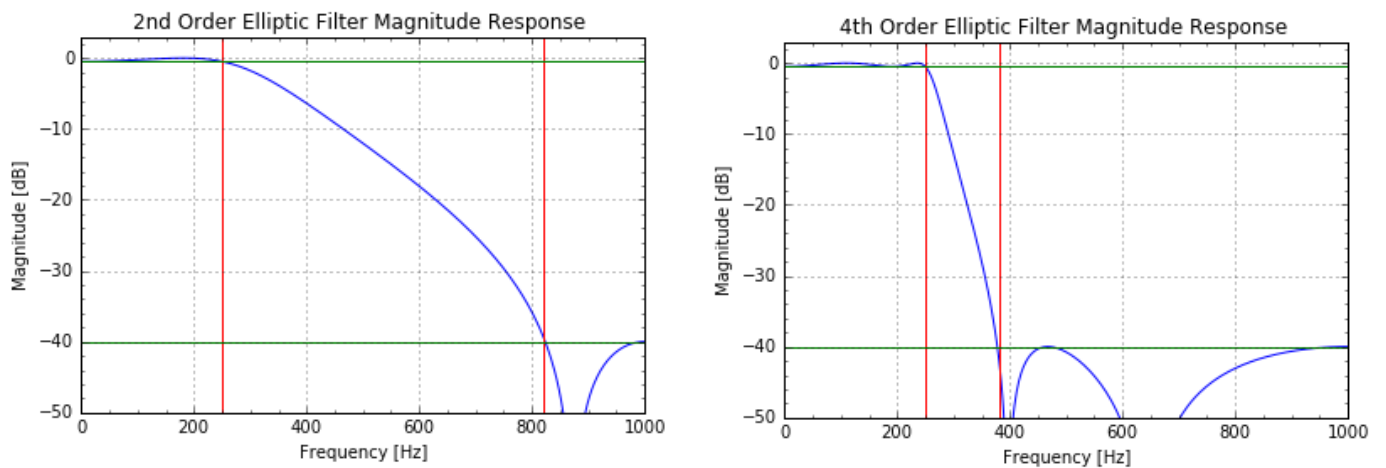


Figure 5: Elliptic Low-Pass Filter Response

2.3.1 Transfer Function using the Z Transform

To obtain the filter's transfer function the Z transform is used to transform a filter's difference equation.

$$\mathcal{Z}\{y[n]\} = \mathcal{Z}\left\{\sum_{i=0}^N b_i x[n-i] - \sum_{i=0}^N a_i y[n-i]\right\}$$

$$Y(z) = \sum_{i=0}^N b_i z^{-i} X(z) - \sum_{i=0}^N a_i z^{-i} Y(z)$$

Bring 'a' Summation to LHS

$$Y(z) + \sum_{i=0}^N a_i z^{-i} Y(z) = \sum_{i=0}^N b_i z^{-i} X(z)$$

Isolate Y(z) and X(z)

$$Y(z) \left\{1 + \sum_{i=0}^N a_i z^{-i}\right\} = X(z) \sum_{i=0}^N b_i z^{-i}$$

Rearranging gives

$$\Rightarrow \frac{Y(z)}{X(z)} = \frac{\sum_{i=0}^N b_i z^{-i}}{1 + \sum_{i=0}^N a_i z^{-i}} = H(z)$$

H(z) gives the digital filter's b & a coefficients.

1. There are two main categories of digital filter (See Appendix 3):
2. Finite Impulse Response (FIR)
3. Infinite Impulse response (IIR)

2.3.2 FIR Filters

FIR filters are non-recursive. Their output is a weighed sum of current and previous input samples. FIR filters exhibit true stability a perfect linear phase response. In addition, FIR filters exhibit greater coefficient sensitivity than IIR filters. Coefficient sensitivity is a phenomenon that results in the filter's frequency response deviating from their design specifications. It arises from the quantisation error from rounding the filter's coefficients.

$$H(z) = \sum_{i=0}^N b_i z^{-i}$$

FIR Transfer Function

2.3.3 IIR Filters

IIR filters are recursive, implementation is achieved through recursive difference equations. The impulse response is theoretically infinite. If a single sample is added to the IIR followed by a series of zeros, then theoretically, an infinite number of non-zero samples could be produced. IIR filters are more efficient than FIR filters. Signal filtering is achieved with less delay and specifications are met with lower order filters when compared with FIR filters. As a result, there is less computational overhead in IIR implementation.

$$H(z) = \frac{\sum_{i=0}^N b_i z^{-i}}{1 + \sum_{i=0}^N a_i z^{-i}} = \frac{B(z)}{A(z)}$$

2.4 Vocal Tract Anatomy and Speech Production

Speech production is a complex process involving many different structures within the mouth, throat and thoracic cavity. At a fundamental level, the mechanism for human speech can be modelled as a source of air pressure that is transformed by vocal structures called vocal folds in the Larynx and is subsequently altered by the structures within the mouth and nasal cavity.

The pressure source for sound production is delivered by the lungs as the diaphragm muscle, which separates the thoracic and abdominal cavities, pushes against the lungs. The intercostal muscles, that are located between the ribs contract, resulting in an exhalation of air through the trachea. Fine control of these structures regulates air pressure.

The vocal folds are controlled by a complex series of muscles which provide a mechanism for fine control of the folds. The vocal folds sit in the airflow expelled from the lungs. As the air passes through the folds they vibrate, the frequency of vibration is determined by the tension on the folds and the extent of how far they are open.

The vibrations from the vocal folds are periodic in nature and contain fundamental frequencies with integer value harmonics [14]. The vibrations from the vocal folds are then filtered through the nasal and oral cavities. If the vocal folds are held open, the waves are formed into specific sounds. To whisper, the vocal folds are held together so that they don't vibrate. Within the oral cavity, the main structures are the velum (soft palate), the tongue and the lips

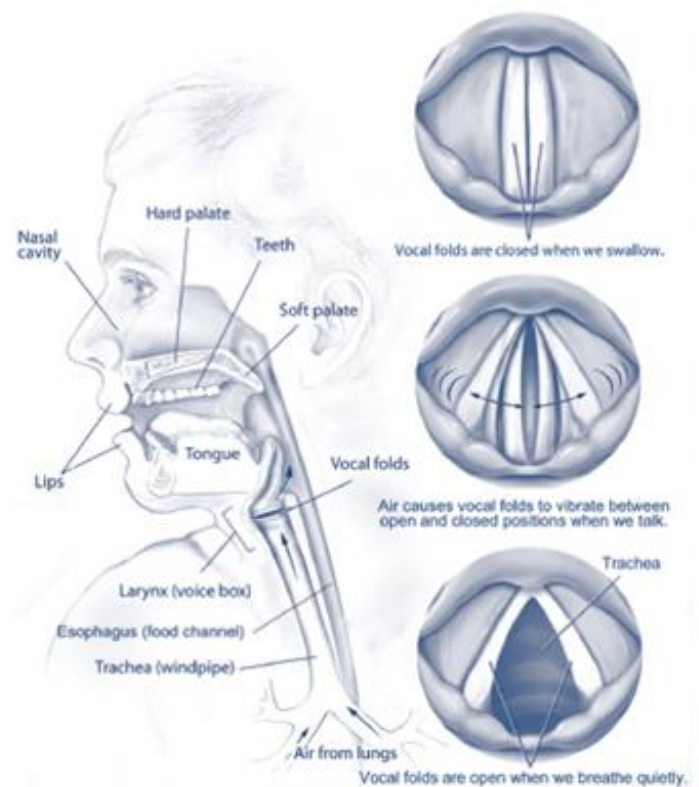


Figure 6: Vocal Tract Structures (nidcd.nih.gov)

2.4.1 Phonemes

Phonemes are the fundamental building blocks of linguistics. The English language contains approximately forty phonemes which can be categorised by the nature and location of articulation [5]. There are four places of articulation:

1. Labial
2. Dental
3. Alveolar
4. Palatal

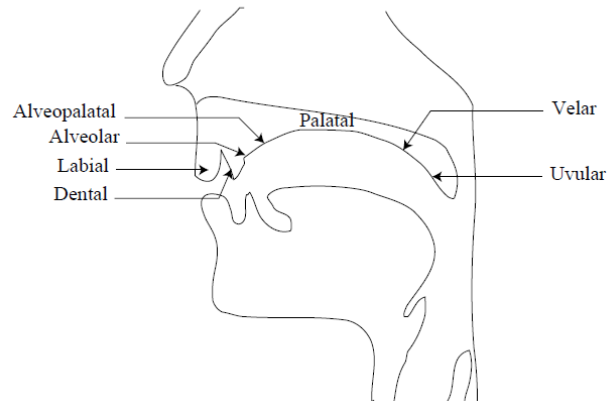


Figure 7: Oral Speech Production Structures

2.4.2 Vowel Formation

Vowel formations are produced with periodic vocal fold excitation. Vowels are characterised by the fact they there is no contraction of the vocal tract. Their acoustic characteristics are dependent on the articulation of the jaw, tongue and lips [5].

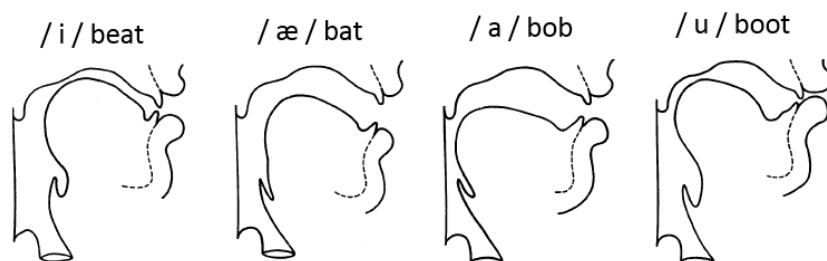


Figure 8: Vowel formation samples

2.4.3 Fricative Formations

Fricatives can be produced with periodic excitation. They result from turbulent airflow at a narrow constriction in one of the mouth's articulation structures. The acoustic characteristics are determined by the structure being constricted [5].

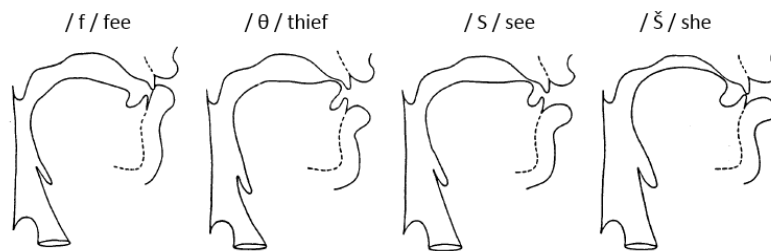


Figure 9: Fricative formations

2.4.4 Stop Formations

There are six stop consonants in the English language, three voiced stops and three unvoiced stops. Stops are characterised by a complete closure of the vocal tract to increase pressure followed by a sudden release of the articulation structure. The sudden release of pressure manifests as turbulent noise. Both voiced and unvoiced stop formations can exhibit periodic excitation during closure [5].

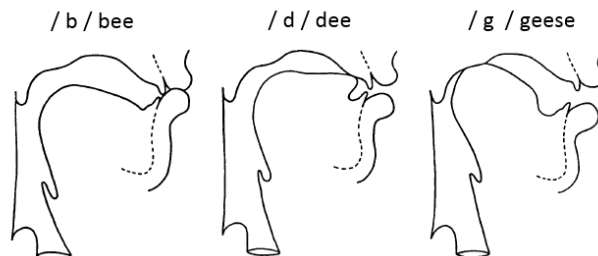


Figure 10: Stop formations

Table 1: Stop Durations

Type	Voiced	Stop durations (ms)	Unvoiced	Stop Durations (ms)
Labial	/b/ bee	12	/p/ pea	57
Dental	/d/ dee	18	/t/ tea	72
Velar	/g/ geese	30	/k/ key	74

The stop consonants are of significance for speech endpoint detection. A stop can be detected as an end of sentence segment if the stop duration exceeds either a timing or spectral threshold in an endpoint detection algorithm.

Chapter 3. System Implementation

Sample numbers for an audio file with thirty-minute duration range from 14,400,000 samples at 8 kHz to 79,380,000 samples at 44.1 kHz. Filtering and processing of large data sets in a single iteration would be prohibitively slow. In addition, it would place a maximum time duration constraint on the length of the audio file to be processed. To overcome the time delays associated with processing long files the author found it necessary to apply block processing techniques for the digital filtering stages where the audio file is processed in a series of smaller sized segments. In testing, this was found to give a considerable speed advantage for filtering and added flexibility by enabling the application to theoretically process audio files of any duration.

During development and evaluation, it was found that Python's execution speed for testing the signal array against the threshold value would add considerable latency to the application. Tests conducted on a 320,000-sample dataset returned an average timing of 325.3ms. As this was scaled up to 79,380,000 the rate of threshold testing exceeded the one-minute duration and the PC resources became overwhelmed. Because of this problem, the author began to research the possibility of using the C language to reduce the threshold evaluation latency. From information obtained during research the author discovered that a well-tested method was to use the Python *ctypes* interface library. The *ctypes* library allows Python scripts to call C functions for speed-sensitive applications. Python imposes strict C-typing on all variables and datatypes to achieve seamless calls to functions that are written as C shared object files. A shared object file is a Linux version of a dynamic link library. Unlike static libraries that are compiled with the main program at compile time, shared object files are pre-compiled and dynamically loaded at run-time as and when they are needed.

Like-for-like testing on the same 320,000-sample dataset revealed that there was a reduction from 312.2ms to 2.2ms showing that the shared object file was running at greater than 140 times the speed of the python implementation (see chapter 4). In addition, the C implementation was capable of processing audio data for durations up to two hours without placing excessive loads on the PC's system resources.

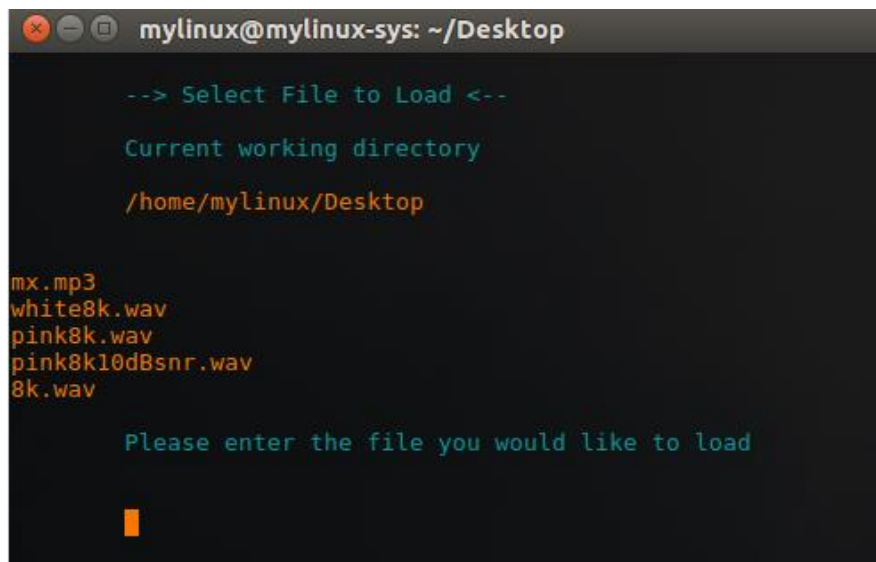
3.1 Command Line Interface

A command line interface (CLI) is a fast, lightweight interface to a computer system. CLIs were widely used in the early days of computing, when interactions with a computer meant sitting down in front of a terminal and typing commands to the screen. These days most human /computer interactions are through a graphical user interface (GUI). Recently, the CLI has made a comeback because of the success of microcomputers like the Raspberry Pi and the increasing popularity of Linux. This application utilised a simple command line interface as a user menu. The interface provides numeric selection that continually runs in an event loop. The application consists of four files (See disk attached for full code):

1. main.py
2. cli.py
3. myDspFuncts.py
4. myDspSo.so

On start-up, the user selects the audio file to process the program and enters an event loop following a phase of spectral noise estimation performed on the selected file.

to access the application functionality.



```
mylinux@mylinux-sys: ~/Desktop

--> Select File to Load <--

Current working directory

/home/mylinux/Desktop

mx.mp3
white8k.wav
pink8k.wav
pink8k10dBsnr.wav
8k.wav

Please enter the file you would like to load

█
```

Figure 11: Start screen


```
mylinux@mylinux-sys: ~/Desktop
```

```
Audio Narration Application, DIT SEEE 2017  
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  
  
Please select from the following options..?  
  
> [1] Search Audio Files  
> [2] Play Time Segment  
> [3] Play Sentence  
> [4] Mark Paragraph  
> [5] Mark Page  
> [6] Purge Redundancy  
> [7] Exit
```

ed in 'cli.py' called `cli`.

```
mylinux@mylinux-sys: ~/Desktop
MODE: [3] Play Sentence

Start of Sentence 1 = 4.61 [s]   End of Sentence 1 = 8.23 [s]
Start of Sentence 2 = 8.23 [s]   End of Sentence 2 = 11.98 [s]
Start of Sentence 3 = 11.98 [s]  End of Sentence 3 = 15.81 [s]
Start of Sentence 4 = 15.81 [s]  End of Sentence 4 = 19.30 [s]
Start of Sentence 5 = 19.31 [s]  End of Sentence 5 = 23.06 [s]
Start of Sentence 6 = 23.06 [s]  End of Sentence 6 = 27.03 [s]
Start of Sentence 7 = 27.03 [s]  End of Sentence 7 = 31.33 [s]
Start of Sentence 8 = 31.33 [s]  End of Sentence 8 = 35.14 [s]
Start of Sentence 9 = 35.14 [s]  End of Sentence 9 = 39.02 [s]

please enter start sentence number -->
```

3.2. DSP System Design

The application requires thirteen individual operations to produce the metafile with sentence, paragraph and page information. The thirteen elements can be allocated to five larger groupings:

1. Spectral noise energy analysis.
2. Block processing for the filtering stages.
3. Speech endpoint detection.
4. Mapping endpoints back to the audio file to locate the sentences.
5. Selecting paragraphs and pages to be written to the metafile.

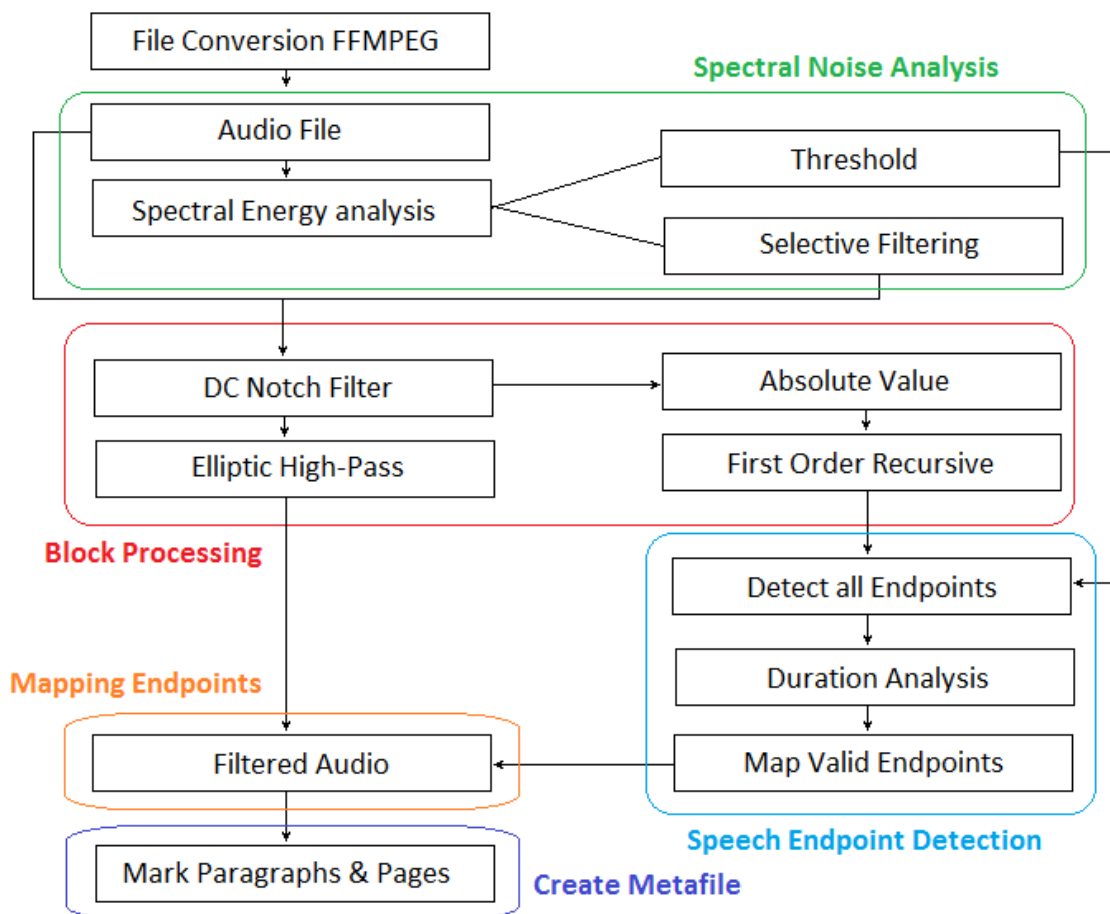


Figure 14: System Block Diagram For DSP Elements

3.3 Noise Analysis

The software begins by taking a snapshot of the audio background noise, this performs a spectral estimation of the noise energy. The spectral energy calculation was performed using Parseval's theorem. This states that the energy of an N-sample signal is numerically equivalent to 1/N times the signal's Discrete Fourier Transform (DFT) spectral energy [13]

$$\sum_{n=0}^{N-1} x[n]^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X(k)|^2$$

Parseval's Theorem can be used to measure the signal energy within spectral bands by summing over the appropriate range of DFT coefficients. The data from the estimation sets two key parameters:

1. The threshold level for speech endpoint detection.
2. A filter flag which switches different filtering stages on or off depending on the noise levels within the spectral bands.

3.3.1 Determining Voice Sounds Threshold Level

Due to some speech sounds having low levels it is important to set the threshold level separating speech from noise as low as possible so as not to mislabel those softer voice sounds. Conversely if the threshold were set too low then background noise in non-speech regions would be wrongly identified as speech. Based on experimentation the threshold level was set in the following manner:

- if low band energy was high, threshold level was set 0.63 times the measured rms noise level.
- if low band energy was low and high band energy was low, threshold level was set 0.775 times the measured rms noise level.
- if low band energy was low and high band energy was high, threshold level was set 0.25 times the measured rms noise level.

3.3.2 Design of noise reducing filter

A noise reducing filter is used to improve reliability of the speech segmentation. The frequency range up to 300 Hz is significant as this is typically where much background noise occurs and it is also outside the most important speech perception range. A high-pass filter (HPF) with cut-off frequency of up to 300 Hz is designed to reduce noise levels. A compromise exists when increasing the HPF cut-off frequency, i.e., noise levels are lowered but low frequency speech quality is adversely affected. Parseval's theorem is used to determine the effectiveness of the HPF at different cut-off frequencies with the objective of selecting the lowest cut-off frequency that is effective at reducing noise while still maintaining best bass sound quality possible.

3.3.3 Band Energy Segmentation

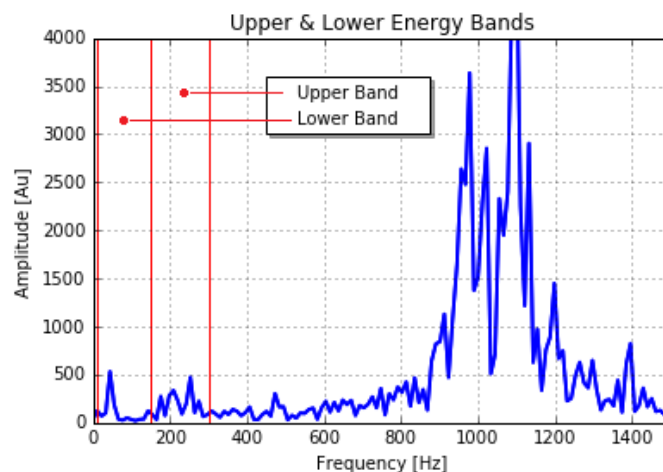


Figure 15: Shows Band Energy Segmentation

The spectral characteristics of the signal noise are divided into two bands:

- 10 - 150 Hz for low-band noise.
- 150 – 300 Hz for higher band noise.

The HPF noise performance is determined by the estimated reduction in power between its input and output noise signals.

A 150 Hz passband edge filter is used if the 300 Hz HPF does not provide significantly improve noise power reduction (> 1 dB) due to its improved low frequency bass sounds.

Also, if there were no significant noise within this lower band as well just a D.C. notch filter is used to remove any D.C offset on the recorded signal.

Calibration data for the threshold and filtering flags was obtained from testing with white and pink noise sources with SNR levels of up to 10dB.

3.3.4 Block Processing Stages

The block processing loop contains three filtering stages

1. DC Notch.
2. Elliptic High-Pass.
3. First-Order Recursive.

The DC notch filter is used to remove DC components from the audio signal. DC bias extends the cone of a speaker and causes high levels of spectral energy in the lower end of the spectrum. It has the following transfer function:

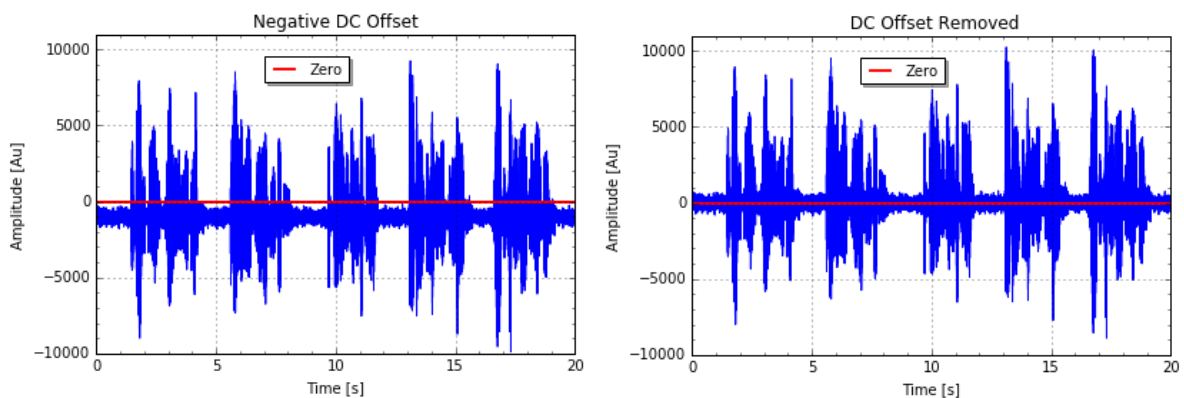


Figure 16: DC offset time Domain

In the time domain, a DC offset appears as a level shift. In the case above (left), The DC offset is negative. The signal has an overall negative bias. The DC notch filter removes that bias, restoring the signal to a level zero median. In the frequency domain, the DC offset is visible as a large spike at 0 Hz

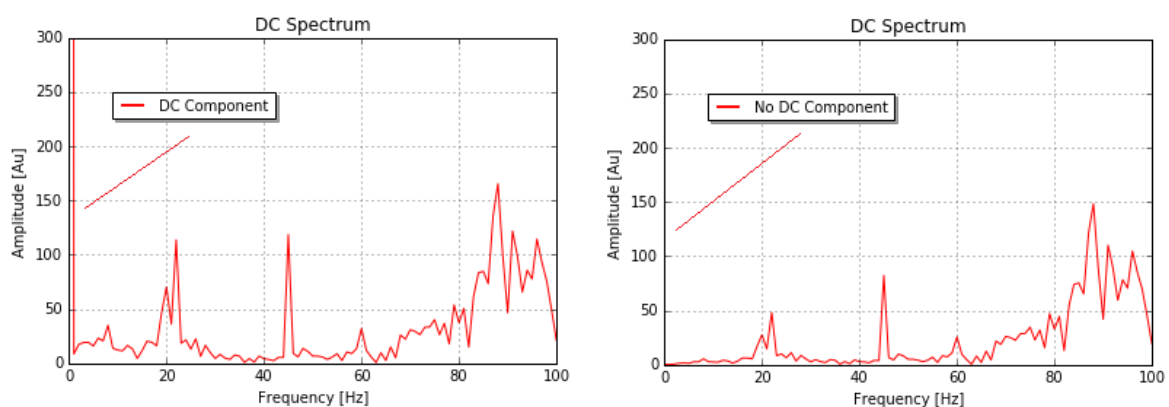


Figure 17: DC offset Frequency Domain

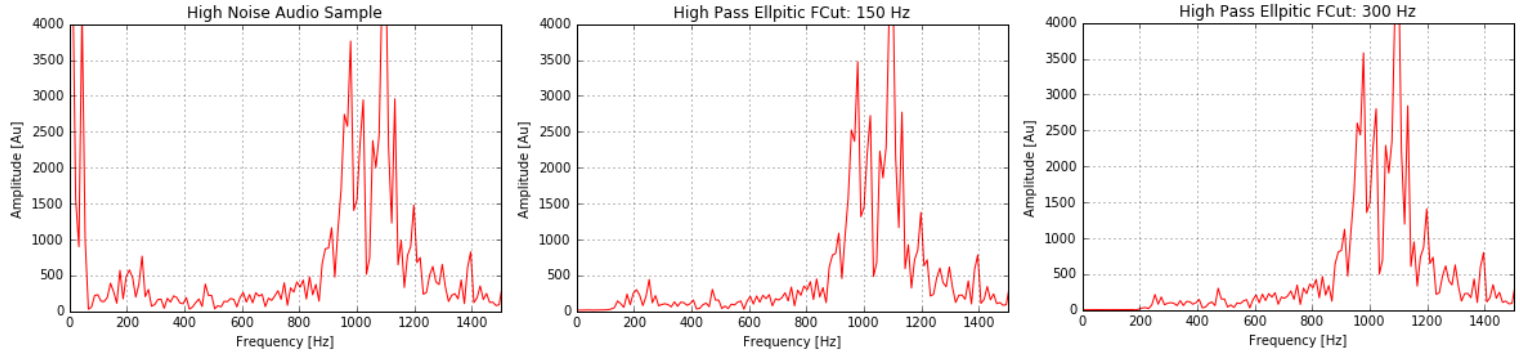


Figure 18: High-Pass filtering at different cut frequencies

The diagram (above) shows a high energy, low-band noise, audio signal being cut at two different frequencies. During the block processing phase the blocks of processed data are loaded into two arrays. One of the arrays stores the filtered audio file and the other array performs further processing necessary for setting the speech threshold.

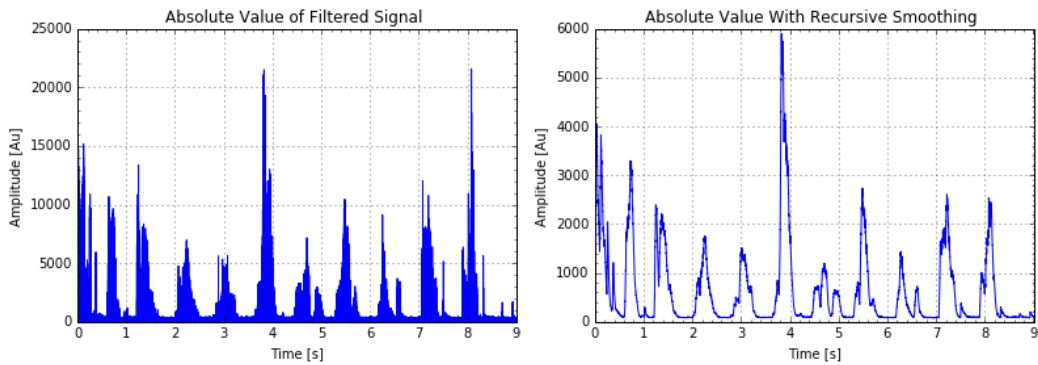


Figure 19: Absolute value of audio signal before and after recursive filtering

The signal level effectively converts the audio signal into a DC signal. Following this, the signal is smoothed with a first-order recursive filter. The smoothing filter reduces sudden transitions in the signal level by averaging the levels between adjacent signal indices. To set the speech endpoint threshold the mean value of the signal is first computed. This value is then scaled by data acquired from the spectral energy analysis. The threshold level should be kept as low as possible but it should always stay above the noise floor.

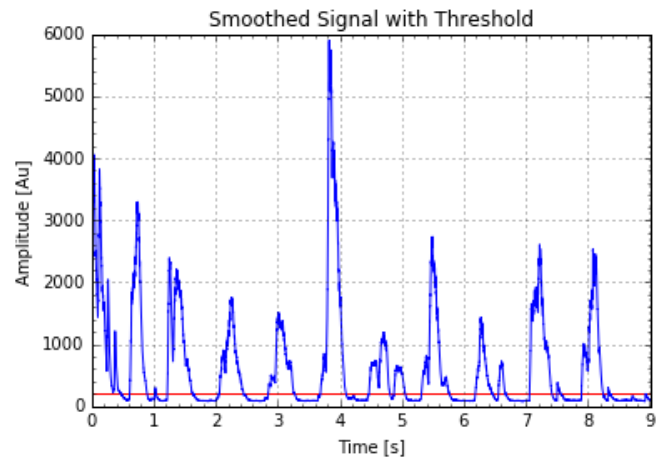


Figure 20: Smoothed Signal with threshold

3.4 Sentence Segmentation

During the development and testing of the speech endpoint detection section of the project the author discovered that the speed of Python for large dataset calculations would be a considerable limiting factor. Following research, the author decided to write the main endpoint detection loop in the C language as a shared object file that could be used by the main Python script. This would be augmented with a secondary time-based analysis in Python on a reduced dataset of pre-detected endpoints from the shared object file. The interaction between the Python program and the .so file was handled by a Python library called *ctypes*. The shared object file is wrapped in a Python function; this function also performs the time-based analysis.

3.3.1 Shared Object File

```
#include<stdio.h>

// compile directive: gcc -shared -Wl,-soname,myDspSo -o myDspSo.so -fPIC myDspSo.c
void detectEndPoints(int frameLength, double *smoothedSignal, double *resultVector, float speechThreshold,
float Fs)
{
    int sampleCntr, zeroCntr, loadCntr = 0;

    for (sampleCntr = 0; sampleCntr<frameLength; ++sampleCntr)
    {
        // startpoint of spoken word
        if ((*smoothedSignal > speechThreshold) && (*(smoothedSignal - 1) < speechThreshold))
        {
            //printf("\n Startpoint at %f s", (sampleCntr/Fs));
            *resultVector = sampleCntr / Fs;
            resultVector += 1;
            loadCntr++;
        }
        // endpoint of spoken word
        if (((*smoothedSignal) < speechThreshold) && (*(smoothedSignal - 1) > speechThreshold))
        {
            //printf("\n Endpoint at %f s", (sampleCntr/Fs));
            *resultVector = sampleCntr / Fs;
            resultVector += 1;
            loadCntr++;
        }
        // increment pointer
        smoothedSignal += 1;
    }

    // Add break condition to the array
    for (zeroCntr = loadCntr; zeroCntr<loadCntr + 2; zeroCntr++)
    {
        *resultVector = 0;
        resultVector++;
    }
}
```

The code featured above shows the C function that is compiled to a shared object file. The use of pointers between Python and the C shared object file enables the shared object file to work directly on the block processed metadata array, this removes the need to copy large arrays. The Python wrapper then has immediate access to the processed array on completion of the endpoint process.

The shared object file is passed five arguments:

1. The number of data samples in the audio array.
2. A void pointer to the first index of the audio array.
3. A second void pointer to the first index of the audio array.
4. The speech threshold calculated from the spectral analysis
5. The audio array's sampling frequency.

The detection of endpoint indices and data transfer of each is performed on the metadata signal in the following manner:

- Pointer one and two are pointing to the first index of the metadata array.
- Pointer one is used to iterate over the metadata array.
- Pointer two is used to write back the index of the endpoints divided by the sampling frequency i.e. the time of the occurrence back in to the previously tested indices of the same array.
- As these indices have already been tested, the values that they contain are redundant. Writing the results back into the same array is a more efficient use of system memory.

3.4.2 Start of spoken word – positive threshold transition.

Pointer one loops through the array and tests each index to see If the data value stored at this index is greater than the silence threshold and the data value in the previous index is less than the threshold.

- When the condition above is met, write the index of the array that is greater than the threshold divided by the sampling frequency i.e. the time of the occurrence back in to location one of the same array.
- increment pointer two.
- Continue to loop pointer one looking for the first instance of the end of the word.

3.4.3 End of spoken word – negative threshold transition

- Continue to loop through the data array and test each index to see If the data value stored in this index is less than the silence threshold and the data value in the previous index is greater than the threshold.
- When the condition above is met, write the index of the array that is less than the threshold divided by the sampling frequency i.e. the time of the occurrence back in to the next location of the same array.
- Increment the pointer two.
- Continue to loop pointer one looking for then next instance of the start of word.

The program continues to repeat this process until the end of the array has been reached. Each start-of-sentence time being stored in an even index and each end-of-word time stored in an odd index in the original array.

End of data array – insert break condition by appending two zeros to the last index of endpoints. During the time duration analysis, the Python wrapper function will loop through the array and calculate the silence durations for the for the second stage of endpoint detection, the zero value acts as a break condition for the loop.

3.4.4 Python Wrapper for Shared Object File and Duration analysis

The duration analysis is performed to augment the threshold evaluation of the shared object file. The shared object file returns all endpoints detected from the active threshold. The duration analysis tests the duration between endpoints and discards the endpoints less than a specified duration, typically 350ms.

```
def getEndpoints(dataSignal, Fs, Nd, duration, speechThreshold):
```

```
    """ C shared object file (dll) """
    myDspSo = CDLL('./myDspSo.so')
    """ assign array to pass to dll """
    c_smoothedSignal=dataSignal

    """ call dll to detect endpoints efficiently """
    myDspSo.detectEndPoints(c_int(Nd),c_smoothedSignal.ctypes.data_as(c_void_p),c_smoothedSignal.ctypes.data_as(c_void_p),c_float(speechThreshold),c_float(Fs))

    """ initialise lists for dynamic sizing """
    sentenceEnd=[]; sentenceStart=[];
    i=0
    """check first transition positive or negative, configure counter accordingly"""
    if(dataSignal[0]>speechThreshold):
        i=1
    else:
        i=0

    while(c_smoothedSignal[i]!=0):

        """ test endpoints for duration > 350ms, end of sentence boundaries """
        if(c_smoothedSignal[i+1] - c_smoothedSignal[i]> duration and i%2 !=0):

            """ dynamically size sentence start & end indices to numpy arrays """
            sentenceStart.append(c_smoothedSignal[i])
            sentenceEnd.append(c_smoothedSignal[i+2])

        i+=1

    """ convert arrays back to numpy arrays """
    sentenceStart=np.array(sentenceStart)*Fs
    sentenceEnd=np.array(sentenceEnd)*Fs
    """ convert lists to numpy arrays """

    """ return sentence endpoints """
    return(sentenceStart, sentenceEnd)
```

The Python function featured performs two operations:

- Wraps the Shared object file in a Python Function
- Performs duration analysis with the endpoints returned from the shared object file
-

The lines featured control access to the shared object file. The Python ctypes library is used to convert the Python datatypes to C data types.

The code featured performs the duration analysis by testing if the difference between adjacent endpoints are greater than the duration specified for a sentence break in the audio file. This loop continues until it meets the break condition entered by the C loop. i.e. the zero-value appended to the last endpoint.

The diagram featured below shows the mechanism for extracting the timing data from the modified audio file and how it is mapped back on to the audio signal. The ABS (Audio Signal) array is the array from which the threshold is derived. It contains the absolute signal levels.

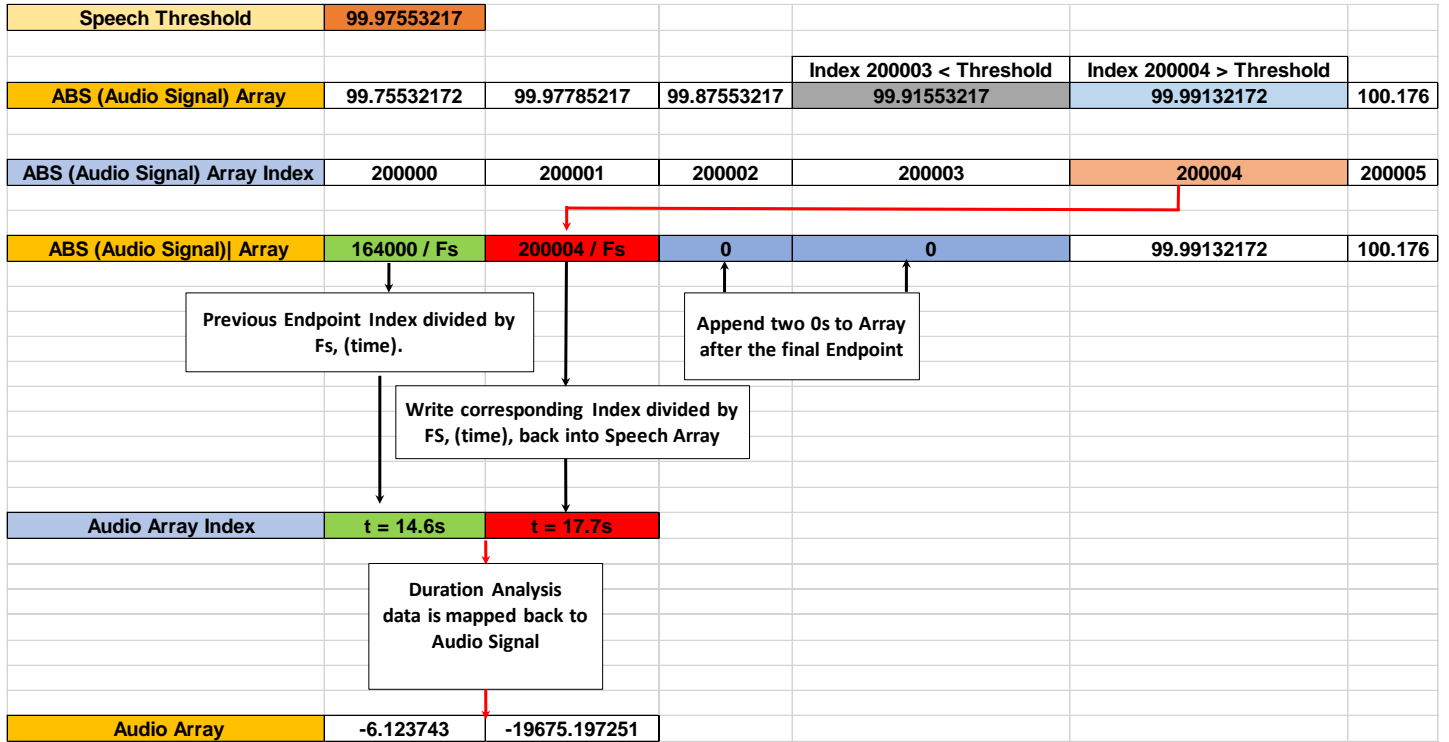


Figure 21: Mapping sentence endpoints to audio signal

3.5 Metafile Creation

The metafile is a file that is used to store the marked pages and paragraphs specified by the user. The file is created using the sentence indices returned by the sentence segmentation phase of the application. It takes the indices of the start and end points of the sentence arrays and asks the user for input to select the appropriate indices for paragraph and page delimiters. The resulting data is then passed to a function which writes those values to a csv file using the Python *pandas* library.

```
def markParagraph(filteredAudio,startOfSentence,endOfSentence,Fs,fileName):
```

```
    paragraphs=[]
    for i in range (len(startOfSentence)):
        startPoints=convertTime((startOfSentence[i]/Fs))
        print("Start of Sentence %d = %s"%(i+1,startPoints)),

        endPoints=convertTime((endOfSentence[i]/Fs))
        print("End of Sentence %d = %s"%(i+1,endPoints))
        print""
        choice='a'
        while(1):

            print("\nplease enter starting sentence number --> ",
                  a=input(""))

            print("\nplease enter ending sentence number --> ",
                  b=input(""))

            startParagraph = (startOfSentence[a]/Fs)
            #print startParagraph
            paragraphs.append(startParagraph)
            endParagraph = (endOfSentence[b]/Fs)
            #print endParagraph
            paragraphs.append(endParagraph)

            print("\n\nEnter Q to quit or any other key to continue marking"
                  choice=raw_input(""))
            if(choice=='q'):
                break

    return (paragraphs)
```

When the function (above) is called it prints all sentence indices to the screen. The lines featured ask for user input for start sentence and end sentence indices to delimit a paragraph. Each index entered by the user is divided by the sampling frequency of the audio file to give the corresponding time for that sentence. The loop continues until the user inputs 'Q' to quit. The program then returns to the main event loop. The resulting data is loaded into a dynamically sizeable list and returned to the calling function. Mark page is a similar function that returns an array of pages (See disk for full program code).

```
def saveData(data,title,blockType):

    start=[]; end=[]

    for i in range(len(data)/1):
        if(i%2==0):
            start.append("%.2f"%data[i])
            print("\n Start at index %d, A = %.2f"%(i,data[i]))
        if(i%2!=0):
            end.append("%.2f"%data[i])
            print("\n End at index %d, A = %.2f"%(i,data[i]))
    index=np.arange(1,len(data)/2 +1,1)
    dataFrame = {
        'Title':title,
        'Index':index,
        'T_Start':start,
        'T_End':end,
        'Type':blockType
    }

    df = pd.DataFrame(dataFrame, columns = ['Type', 'Index', 'T_Start', 'T_End', 'Title'])

    df.to_csv(str(title)+''.csv',mode='a')
```

The function featured above is called by the main event loop after paragraphs or pages have been marked. It takes three arguments:

1. A Python list containing the indices of the start and end time of each of the paragraphs.
2. The title of the audio file that is being processed
3. The type of data that is being sent (paragraph/page).

The three data elements are then inserted into a Python dictionary which acts as a data frame. Each paragraph or page is given a specific index e.g. page 1, page 2. In addition, a title frame is written to the head of the csv file. If this is the first write to the csv file, a file with a name that matches the audio file is created. If the write is a subsequent write, the data is appended to the existing file. The corresponding times are written to the csv file with the index, the csv file can take both pages and paragraphs.

Chapter 4. Testing and Results

4.1 Test environment

All development and testing was undertaken with a Linux operating system (Ubuntu 16.04 LTS). The personal computer was a Lenovo T410 Laptop with the following hardware specifications:

Architecture:	x86_64
CPU(s):	2
Model name:	Intel(R) Core(TM) i5 CPU M 520 @ 2.40GHz
CPU MHz:	2394.014
RAM:	DDR3 4GB
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	3072K

Specifications obtained from Linux command: `lscpu | grep <cache,cpu etc>`

4.1.1 Software Dependencies

The software dependencies required to run the application are:

- FFMPEG - Open-source Audio and visual file conversion software. Require for file conversion from mp3 to wav.
- python-dev - required for Python integration with C packages.

4.2 Noise characteristics

Testing of the Audio narration application was performed using white and pink noise sources. These sources were selected because they are used to evaluate existing speech boundary algorithms [8] as they are more difficult to remove using static filtering techniques. Pink noise exhibits high power at low frequencies that rolls-off at a rate of $1/F$. White noise exhibits equal power at all frequencies. Typically, the noise characteristics of audio recordings would be low due to attenuation of higher frequencies by the surrounding environment.

4.3 CLI Testing

Time constraints brought on by unscheduled delays precluded a comprehensive testing schedule. As a result, the CLI wasn't fully tested. Many areas require refactoring to eliminate discrepancies.

A basic test plan was devised for the main menu inputs. The plan tested for valid and invalid inputs. The results revealed the need for refactoring for selected functionality.

Table 2: CLI Test Plan

Test	Input	Expected Output	Actual output	Error	Result
Loading mp3 files	mx.mp3	wav file	wav file	N/A	Successful
Loading wav files	8k.wav	wav file	wav file	N/A	Successful
Showing both Audio formats on main menu request	User	mp3, wav	mp3, wav	N/A	Successful
Play Time Segment	Time range 0 to file duration	Play segment of specified duration	Play segment of specified duration	N/A	Successful
Play Time Segment	Time range beyond file duration	Instruct user to re-enter correct value	Crash	Program Crash	Unsuccessful
Play sentence	Sentence range of file duration	Play segment of specified duration	Play segment of specified duration	N/A	Successful
Play sentence	Sentence range beyond file duration	Instruct user to re-enter correct value	Index wrap around	Sentence indices wrap around to start indices	Unsuccessful
Mark Paragraph	Sentence Indices	csv file with paragraphs marked	csv file with paragraphs marked	N/A	Successful
Mark Paragraph	Sentence range beyond file duration	Instruct user to re-enter correct value	csv file with paragraphs marked with invalid times	Invalid Data	Unsuccessful
Mark Page	Sentence Indices	csv file with pages marked	csv file with pages marked	N/A	Successful
Mark Page	Sentence range beyond file duration	Instruct user to re-enter correct value	csv file with pages marked with invalid times	Invalid Data	Unsuccessful
Purge Redundancy	wav file	wav file with redundancy removed	wav file with redundancy removed	N/A	Successful
Exit	User	Clean Termination	Clean Termination	N/A	Successful

4.4 Benchmarking

Benchmarking of a 320,000-sample 8kHz dataset was performed to ascertain the performance gains from using a shared object file to perform the speech endpoint detection calculations. Like-for-like test results revealed that the shared object, C, implementation was greater than 140 times faster than the equivalent Python implementation. This included the time penalties incurred by accessing the shared object file. As this was scaled-up to larger files the performance gains increased for the C implementation. Monitoring system resources on the System Performance GUI indicated that Python's use of memory resources was less efficient than that of C. Following research into the underlying mechanism, it was found that the C implementation had less computational overhead than that of Python. The system could better exploit the principle of locality of reference. Specifically, spatial locality of reference. Spatial locality of reference allows blocks of adjacent memory to be read into cache in a single read. Cache has a higher access speed than RAM thus allowing the CPU to access the data faster.

Table 3: Benchmarking Results

Test #	Audio File	Execution time [ms]	File Duration [s]	Number of Samples
C Implementation				
1	8k_tst.wav	2.367	40	320000
2	8k_tst.wav	2.098	40	320000
3	8k_tst.wav	2.31	40	320000
4	8k_tst.wav	1.945	40	320000
5	8k_tst.wav	2.392	40	320000
Average Timing		2.2224		
Python Implementation				
1	8k_tst.wav	315.463	40	320000
2	8k_tst.wav	329.205	40	320000
3	8k_tst.wav	345.887	40	320000
4	8k_tst.wav	323.153	40	320000
5	8k_tst.wav	312.81	40	320000
Average Timing		325.3036		

4.5 Sentence segmentation Testing

Testing was undertaken to find the minimum stop duration that would accurately detect the end of sentences for files by three narrators. A duration of 350 milliseconds was found to have the best performance. The table below illustrates the test data for a forty second test file containing ten sentences.

Table 4: Sentence Segmentation Durations

Sentences In File	Threshold Level	Stop Duration (ms)	Detected	Missed
10	Low	40	Voiced Stops, Unvoiced Stops inter- word periods, sentences	Fricative Durations
10	Low	80	Unvoiced Stops. inter- word periods, sentences	Fricative Durations, Voiced Stops
10	High	40	inter-word periods, sentences.	Fricative Durations. Voiced Stops, Unvoiced Stops.
10	High	80	Unvoiced Stops. inter- word periods, sentences.	Fricative Durations, Voiced Stops, Unvoiced Stops.
10	Active	200	Inter-word periods, sentences.	Fricative Durations. Voiced Stops, Unvoiced Stops.
10	Active	350	Long duration inter- word periods. All sentences	Fricative Durations. Voiced Stops, Unvoiced Stops. Short inter-word periods
10	Active	400	Long duration sentence pauses	Fricative Durations. Voiced Stops, Unvoiced Stops. Most inter-word periods. Short duration sentence pauses

4.6 Block Processing

Block processing is a technique that is used in real-time data acquisition and processing applications, it is implemented like a sliding window across the full data range. In this application, the technique is employed because of the large data sets involved. The block size is determined by the length of an audio file being processed. For a test file of forty seconds, experimental results for this application showed that the fastest execution time was obtained from a block size that was approximately one twelfth of the file duration. For longer, more realistic lecture durations of thirty to sixty minutes it was found that the number of blocks required was between 192 and 256

Testing revealed Block processing discontinuities. If the initial conditions aren't provided, there are discontinuities between each block as the next block starts from a zero-value initial condition. The discontinuities give rise to an audible 'clicking' as the signal steps up from a zero-value to the first value in the block array. The problem is visually evident in the diagrams featured above. The diagram shows a segment of the metadata from an audio signal processed into six blocks (fig 22). The signal is pulled to a zero-value initial condition. There is a visual step in the signal as it transitions from block to block. To overcome this problem, the final conditions of block $[n]$ need to be provided as the initial conditions to block $[n+1]$.

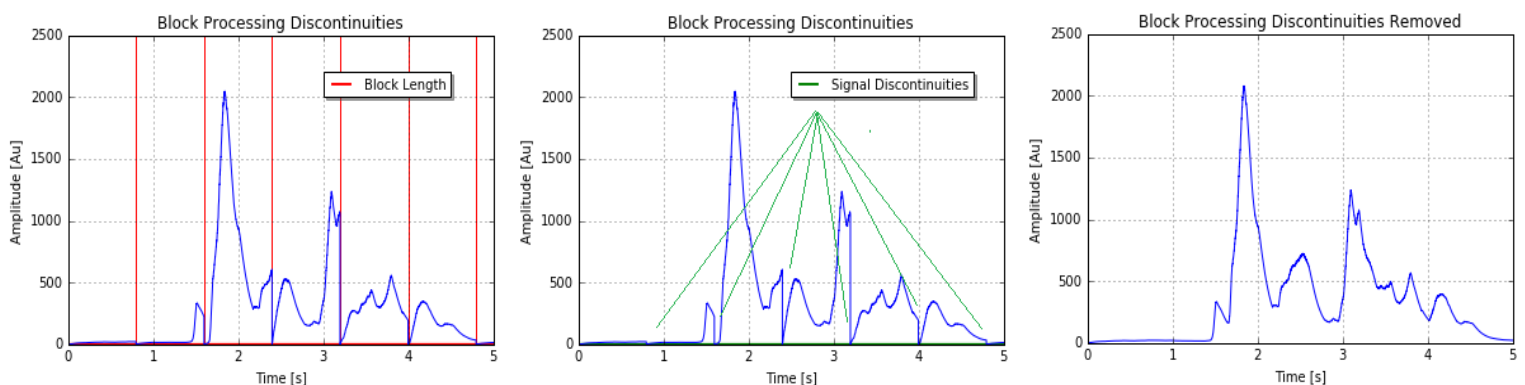


Figure 22: Block Processing Discontinuities

4.7 Application Limitations

During testing the performance of the application to detect all sentences dropped-off as the audio file SNR went below 10dB. The author finds that a more detailed study into the relationship between the noise and active thresholding is required to maximise system performance. Furthermore, the author would investigate the integration of adaptive filtering using a dual-microphone system for narration recording to increase sound quality in noisy environments.

The application was designed and tested in a Linux environment. For the program to run in a Windows environment the C shared object file would need to be compiled to a Dynamic Link Library (dll). Compiling to dll format would require minor changes to the structure of the C file for the compilation.

4.8 Future Development

Python dictionary data structures allow data to be stored in attribute value pairs which consist of an attribute that describes the data type and the data associated with the description. During development, the author designed the function for writing data to the csv file in dictionary format to enable easy integration with a MongoDB database. Using MongoDB as part of the application would allow a comprehensive record of all data processed by the application to be retained. The application would be augmented with a function to add querying functionality.

Chapter 5. Conclusion

Speech endpoint detection is an active field of research with applications ranging from automotive control to speech powered search engines. Research uncovered numerous solutions to the problem, all of which had strengths and weaknesses. This project, and the research performed before and throughout its development, investigated the viability of the Python programming language to create a responsive “Audio Narration Application”. Audio narration software has many applications, such as teaching children with visual impairments or high-incidence cognitive disabilities; staff training and audio guides for museums and galleries. The application used spectral noise analysis to perform active thresholding and selective digital filtering to obtain reliable speech segmentation in noise levels up to 10dB. The segmentation was used to create a metafile on the syntactic structure of a narrated text and remove redundancy from audio files.

Python was used for most of the application but performance became a problem when audio files exceeded five minute durations. To remedy this, the design leveraged the C language to create a shared object file which brought about a measured 140 times increase in speech endpoint detection. These results provided an insight into the performance metrics between statically typed, compiled languages like C and dynamically typed, interpreted, scripting languages like Python. Overall, Python was found to be a capable language for signal processing and general purpose computing. Python has an abundance of libraries for scientific computing including pandas for data structures and analysis; IPython for an interactive console; Numpy for N-dimensional array computations and Matplotlib for two-dimensional plotting.

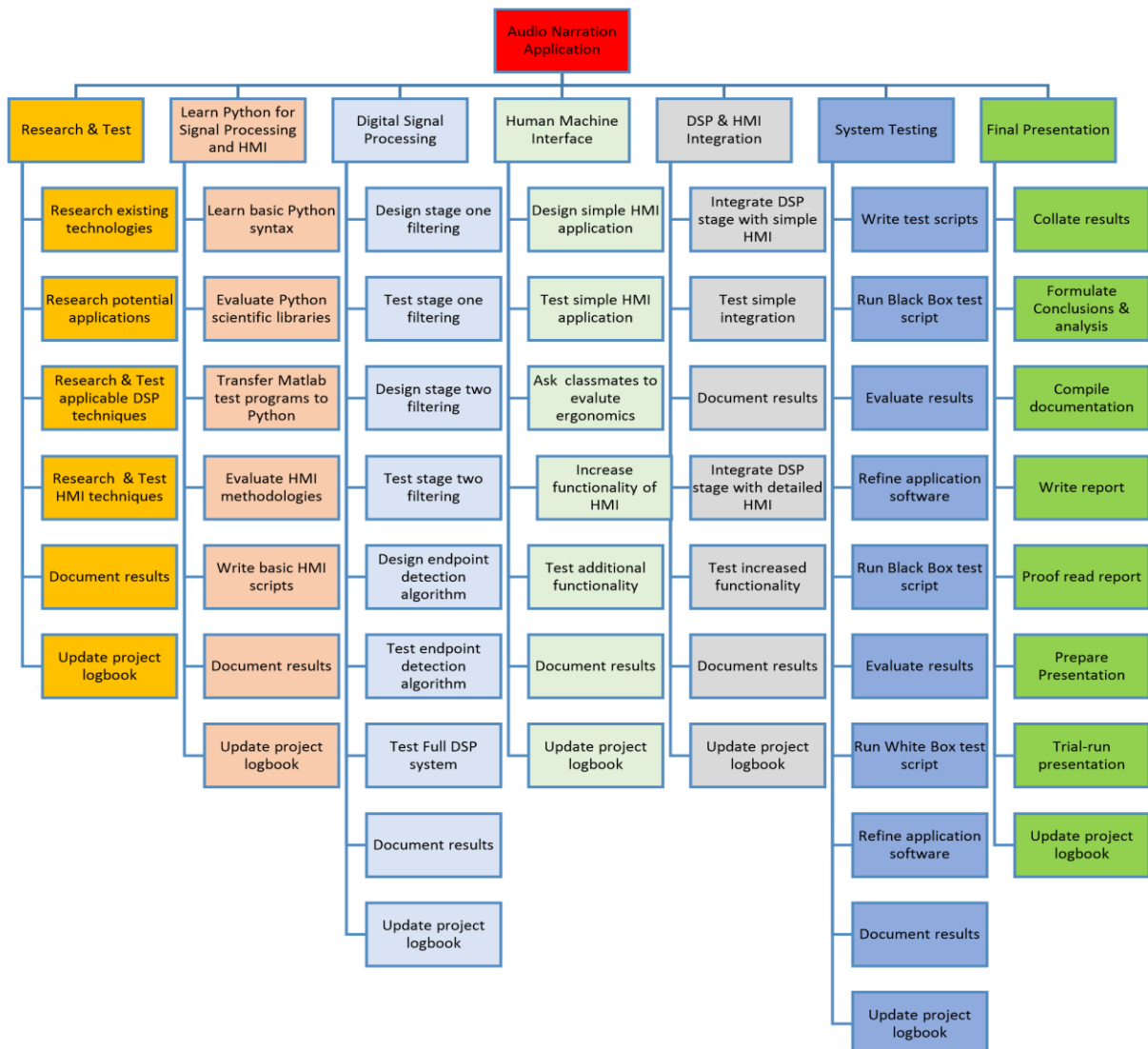
Going forward, the application would require further development in areas of filtering and usability. Testing on signals with pink and white noise contamination of 10dB signal to noise ratio revealed that accurate segmentation was durable but there was a shortfall in audio quality. To overcome this problem an investigation into the integration of adaptive filtering using dual microphones for recording would be required. Furthermore, for the application to be a fully usable system for real-world applications, additional functionality would be required to read the metafiles it creates and map the data contained onto the audio tracks they are derived from.

References

- [1] E. Boyle, M. Rosenberg, V. Connelly, S. Washburn, L. Brinckerhoff and M. Banerjee, "Effects of Audio Texts on the Acquisition of Secondary-Level Content by Students with Mild Disabilities", *Learning Disability Quarterly*, vol. 26, no. 3, pp. 203-214, 2003. [Accessed: 27-4-2017].
- [2] E. W. Anderson, G. A. Preston and C. T. Silva, "Using Python for Signal Processing and Visualization," in *Computing in Science & Engineering*, vol. 12, no. 4, pp. 90-95, July-Aug. 2010. [Accessed: 27-4-2017].
- [3] K. Yamamoto, F. Jabloun, K. Reinhard and A. Kawamura, "Robust Endpoint Detection for Speech Recognition Based on Discriminative Feature Extraction," 2006 IEEE International Conference on Acoustics Speech and Signal Processing Proceedings, Toulouse, 2006, pp. I-I. [Accessed: 22-4-2017].
- [4] G. Evangelopoulos and P. Maragos, "Multiband Modulation Energy Tracking for Noisy Speech Detection", *IEEE Transactions on Audio, Speech and Language Processing*, vol. 14, no. 6, pp. 2024-2038, 2006.
- [5] MIT EECS, *Acoustic Properties of Speech & Sound*. Cambridge Ma: MIT, 2000.
- [6] Engineers Ireland, *Code of Ethics* in *Code of ethics*. 2010. [Online]. Available: <http://www.engineersireland.ie/EngineersIreland/media/SiteMedia/about/Engineers-Ireland-Code-Of-Ethics-2010.pdf>. [Accessed: Jan. 30, 2017].
- [7] "IEEE IEEE code of ethics," 2017. [Online]. Available: <http://www.ieee.org/about/corporate/governance/p7-8.html>. Accessed: Jan. 30, 2017.
- [8] Li Yi and F. Yingle, "A Novel Algorithm to Robust Speech Endpoint Detection in Noisy Environments," 2007 2nd IEEE Conference on Industrial Electronics and Applications, Harbin, 2007, pp. 1555-1558. [Accessed: 7- 5- 2017].
- [9] Y. Zhang, K. Wang and B. Yan, "Speech endpoint detection algorithm with low signal-to-noise based on improved conventional spectral entropy," 2016 12th World Congress on Intelligent Control and Automation (WCICA), Guilin, 2016, pp. 3307-3311
- [10] A. Kelly, "Digital Communications ", Dublin Institute of Technology, Kevin St, 2016.
- [11] P. Cha and J. Molinder, *Fundamentals of signals and systems*, 1st ed. Cambridge: Cambridge University Press, 2006.
- [12] K. Tiernan, "Digital Signal Processing ", Dublin Institute of Technology, Kevin St, 2016.
- [13] H. Baher, *Analog & digital signal processing*, 2nd ed. Sussex, UK: Wiley & sons, 2002.
- [14] UNSW, "Voice Acoustics: an introduction to the science of speech and singing", [Newt.phys.unsw.edu.au](http://newt.phys.unsw.edu.au), 2017. [Online]. Available: <http://newt.phys.unsw.edu.au/jw/voice.html#filter>. [Accessed: 25- May- 2017].

Appendices

Appendix 1 - Work Breakdown Structure



A work breakdown structure was used to decompose the project into a series of tasks and sub-tasks. The top line of the structure is the project title; the second line is the task headings and the subsequent lines are the sub-tasks associated with the headings. Work breakdown structures are used to divide the project into modular sub-tasks that are inter-related. This information was then used to determine a timeline and predecessor tasks both within modules and between modules.

Appendix 2 - Sampling

Digital signal processing is a branch of electrical engineering that is concerned with digital signals from electronic devices and the digitisation of analogue signals. Analogue signals are signals that vary continuously in both amplitude and time [7]. Examples include electrical signals from the human heart, radar signatures and seismic waves. Digital signals are numerical signals that have discrete amplitude and time with finite precision in both the time and amplitude axes [11], digital signals include the signals used in computer and telecommunications systems.

Sampling an analogue signal is restricted to finite numerical values. A digital signal representation of an analogue signal has a minimum granularity on both the x and y axes. The sampling frequency determines the discrete slices of the x-axis resolution. The bit-depth of the analogue to digital converter that is sampling the digital signal determines the discrete levels of the y-axis. A 16-bit analogue to digital converter (ADC), will have 2^{16} discrete levels. For a voice signal this will range from -32678 to +32678. The relationship between the level of the incoming voice signal and the discrete level is obtained from the following equation:

$$\text{Voltage granularity} = \frac{V_{Ref}}{2^n - 1}$$

V_{Ref} is the reference voltage of the microphone. For a computer with a 5 volt reference the discrete levels in terms of voltage would be

$$\frac{5.0}{2^{16} - 1} = 76.3 \mu V$$

This value is the difference between two adjacent levels.

This in turn leads to a problem called quantisation error. If we used an ADC with a low bit-depth e.g. 3-bit, that would give $2^3 = 8$ levels. The signal level measurement will have a granularity of

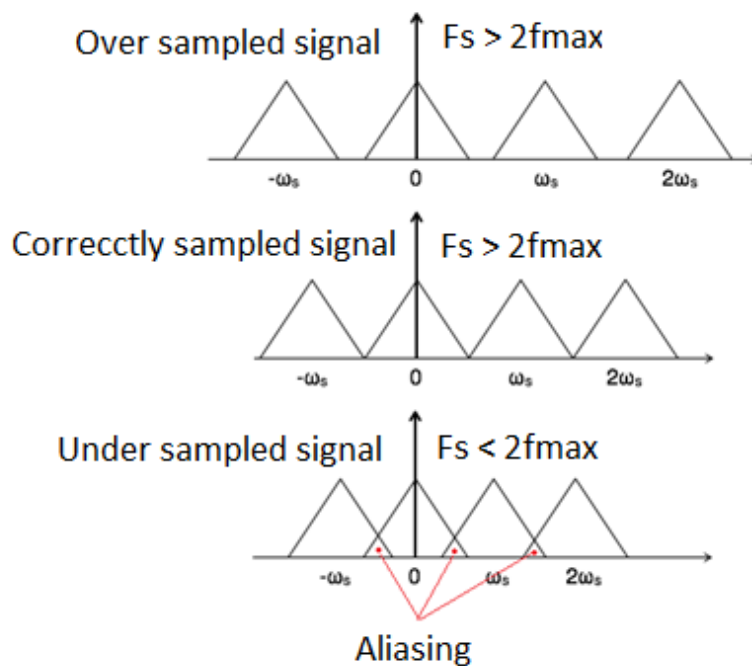
$$\frac{5.0}{2^3 - 1} = 714 mV$$

This means that the minimum resolution with which the ADC could measure a continuous signal level is 714 mV. An ADC should be chosen with a sufficient bit-depth for the application in which it is being used.

The Sampling theorem, developed by Shannon & Nyquist¹ states that all information is retained for the reconstruction of a signal if the time domain signal $x(t)$ is band limited and sampled at $2f_{\text{Max}}$ where f_{Max} is the highest frequency component of $x(t)$.

$$X_s(f) = F_s \sum_{k=-\infty}^{\infty} X(f - kf_s)$$

The expression states that the sampled spectrum is a sum of series of shifted images of a series $X(f)$. $X_s(f)$ is the spectrum of the sampled signal sampled at F_s Hertz [12].



Spectral characteristics of sampling

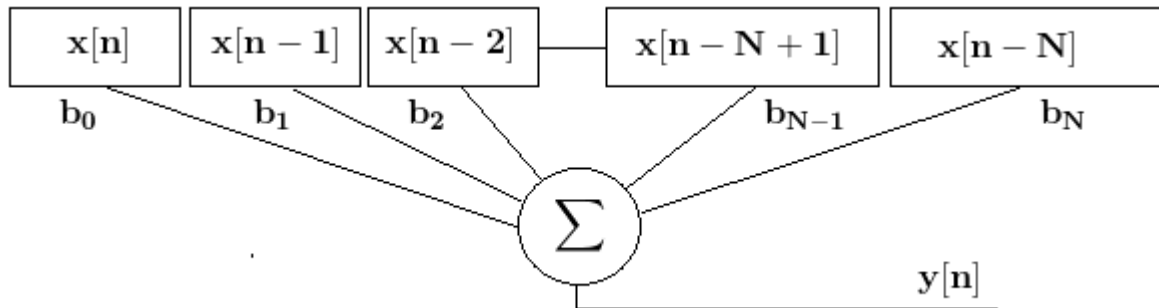
If $x(t)$ is sampled at less than $2f_{\text{Max}}$, information in the signal will be lost. This results in a phenomenon called aliasing.

If $x(t)$ is sampled at a rate greater than $2f_{\text{Max}}$, information is retained. This process is sometimes used to remove the need for an anti-aliasing filter benefit.

Appendix 3 - Digital Filters

FIR Filters

FIR filters are non-recursive. Their output is a weighed sum of current and previous input samples.



FIR Filter Structure

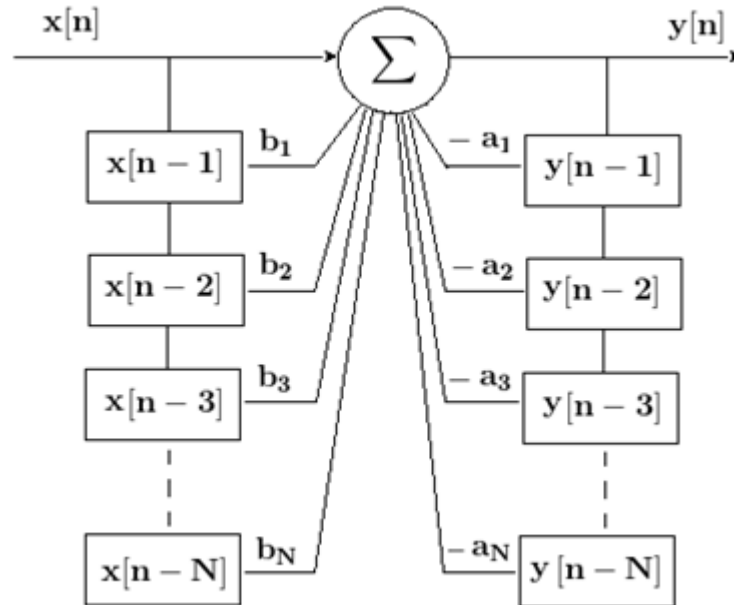
FIR filters have true stability a perfect linear phase response. In addition, FIR filters exhibit greater coefficient sensitivity than IIR filters. Coefficient sensitivity is a phenomenon that results in the filter's frequency response deviating from their design specifications. It arises from the quantisation error from rounding the filter's coefficients.

$$H(z) = \sum_{i=0}^N b_i z^{-i}$$

FIR Transfer Function

IIR Filters

IIR filters are recursive, implementation is achieved through recursive difference equations. The impulse response is theoretically infinite. If a single sample is added to the IIR followed by a series of zeros, then theoretically, an infinite number of non-zero samples could be produced.



IIR Filter Structure

IIR filters are more efficient than FIR filters. Signal filtering is achieved with less delay and specifications are met with lower order filters when compared with FIR filters. As a result, there is less computational overhead in IIR implementation.

$$H(z) = \frac{\sum_{i=0}^N b_i z^{-i}}{1 + \sum_{i=0}^N a_i z^{-i}} = \frac{B(z)}{A(z)}$$


IIR Transfer Function

Appendix 4 - CLI Screenshots

Start Screen

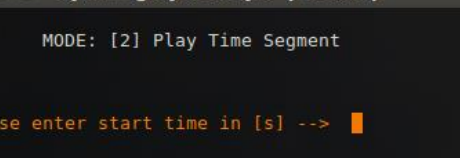
```
mylinux@mylinux-sys: ~/Desktop
--> Select File to Load <--
Current working directory
/home/mylinux/Desktop
mx.mp3
white8k.wav
pink8k.wav
pink8k10dBsnr.wav
8k.wav
Please enter the file you would like to load
```

[Main Menu](#)

```
 mylinux@mylinux-sys: ~/Desktop
```

```
Audio Narration Application, DIT SEE 2017  
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  
  
Please select from the following options..?  
  
 > [1] Search Audio Files  
 > [2] Play Time Segment  
 > [3] Play Sentence  
 > [4] Mark Paragraph  
 > [5] Mark Page  
 > [6] Purge Redundancy  
 > [7] Exit
```

Play Time Segment



A terminal window titled "mylinux@mylinux-sys: ~/Desktop" with standard window controls. The terminal output shows "MODE: [2] Play Time Segment" and a prompt "please enter start time in [s] --> " with a red cursor.

```
mylinux@mylinux-sys: ~/Desktop
MODE: [2] Play Time Segment
please enter start time in [s] --> 
```

Play Sentence

```
mylinux@mylinux-sys: ~/Desktop
MODE: [3] Play Sentence

Start of Sentence 1 = 4.61 [s] End of Sentence 1 = 8.23 [s]
Start of Sentence 2 = 8.23 [s] End of Sentence 2 = 11.98 [s]
Start of Sentence 3 = 11.98 [s] End of Sentence 3 = 15.81 [s]
Start of Sentence 4 = 15.81 [s] End of Sentence 4 = 19.30 [s]
Start of Sentence 5 = 19.31 [s] End of Sentence 5 = 23.06 [s]
Start of Sentence 6 = 23.06 [s] End of Sentence 6 = 27.03 [s]
Start of Sentence 7 = 27.03 [s] End of Sentence 7 = 31.33 [s]
Start of Sentence 8 = 31.33 [s] End of Sentence 8 = 35.14 [s]
Start of Sentence 9 = 35.14 [s] End of Sentence 9 = 39.02 [s]

please enter start sentence number --> █
```

Mark Page

```
mylinux@mylinux-sys: ~/Desktop
MODE: [4] Mark Paragraph

Start of Sentence 1 = 4.61 [s] End of Sentence 1 = 8.23 [s]
Start of Sentence 2 = 8.23 [s] End of Sentence 2 = 11.98 [s]
Start of Sentence 3 = 11.98 [s] End of Sentence 3 = 15.81 [s]
Start of Sentence 4 = 15.81 [s] End of Sentence 4 = 19.30 [s]
Start of Sentence 5 = 19.31 [s] End of Sentence 5 = 23.06 [s]
Start of Sentence 6 = 23.06 [s] End of Sentence 6 = 27.03 [s]
Start of Sentence 7 = 27.03 [s] End of Sentence 7 = 31.33 [s]
Start of Sentence 8 = 31.33 [s] End of Sentence 8 = 35.14 [s]
Start of Sentence 9 = 35.14 [s] End of Sentence 9 = 39.02 [s]

please enter starting sentence number --> █
```

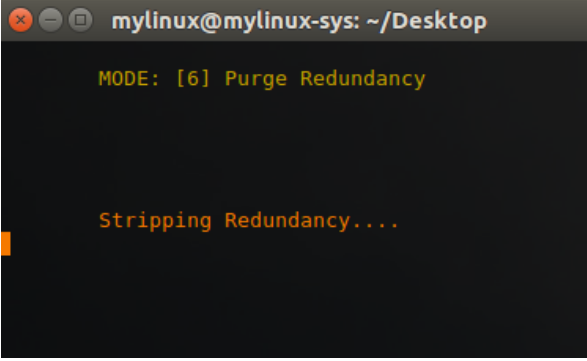
Mark Paragraph

```
mylinux@mylinux-sys: ~/Desktop
MODE: [5] Mark Page

Start of Sentence 1 = 4.61 [s] End of Sentence 1 = 8.23 [s]
Start of Sentence 2 = 8.23 [s] End of Sentence 2 = 11.98 [s]
Start of Sentence 3 = 11.98 [s] End of Sentence 3 = 15.81 [s]
Start of Sentence 4 = 15.81 [s] End of Sentence 4 = 19.30 [s]
Start of Sentence 5 = 19.31 [s] End of Sentence 5 = 23.06 [s]
Start of Sentence 6 = 23.06 [s] End of Sentence 6 = 27.03 [s]
Start of Sentence 7 = 27.03 [s] End of Sentence 7 = 31.33 [s]
Start of Sentence 8 = 31.33 [s] End of Sentence 8 = 35.14 [s]
Start of Sentence 9 = 35.14 [s] End of Sentence 9 = 39.02 [s]

please enter starting sentence number --> █
```

Purge Redundancy

A terminal window with a dark background and orange text. The title bar shows 'mylinux@mylinux-sys: ~/Desktop'. The terminal displays 'MODE: [6] Purge Redundancy' followed by 'Stripping Redundancy...'. A small orange vertical bar is visible on the left side of the terminal window.

```
mylinux@mylinux-sys: ~/Desktop  
MODE: [6] Purge Redundancy  
  
Stripping Redundancy...
```