

**The Development of a
Parametric Real-Time Voice Source Model for
use with Vocal Tract Modelling Synthesis on
Portable Devices**

Jacob Harrison
MSc by Research
University of York
Electronics
November 2014

Abstract

This research is concerned with the natural synthesis of the human voice, in particular, the expansion of the LF-model voice source synthesis method. The LF-model is a mathematical representation of the acoustic waveform produced by the vocal folds in the human speech production system. Whilst being used in many voice synthesis applications since its inception in the 1970s, the parametric capabilities of this model have remained mostly unexploited in terms of real-time manipulation. With recent advances in dynamic acoustic modelling of the human vocal tract using the two-dimensional digital waveguide mesh (2D DWM), a logical step is to include a real-time parametric voice source model rather than the static LF-waveform archetype.

This thesis documents the development of a parameterised LF-model to be used in conjunction with an iOS-based 2D DWM vocal tract synthesiser, designed with the further study of voice synthesis naturalness as well as improvements to assistive technology in mind.

Table of Contents

Abstract	i
List of Figures	v
List of Tables	vii
List of Accompanying Material	viii
Acknowledgements	ix
Author’s Declaration	x
1. Introduction	1
1.1 Thesis Overview	1
1.2 Thesis Structure	3
2. Literature Review	5
2.1 The ‘Source + Modifier’ Principle	5
2.2 Physiology of the Vocal Folds	8
2.3 Voice Types	11
2.4 Modelling the Voice Source	16
2.4.1 Existing Methods for Voice Source Synthesis	17
2.4.2 The Liljencrants-Fant Glottal Flow Model.....	20
2.5 Vocal Tract Modelling	28
2.6 ‘Naturalness’ in Speech Synthesis	37
3. A Parametric, Real-Time Voice Source Model	44
3.1 Motivation for Design	45
3.2 Specifications	47
3.3 Design	48
3.3.1 Choice of Parameters.....	48
3.3.2 Wavetable Synthesis.....	53
3.3.3 Voice Types.....	55
3.3.4 iOS Interface	58
3.3.5 Final Design	59
3.4 Implementation	61
3.4.1 Implementation in MATLAB	62
3.4.2 Implementation in iOS	69

3.5 System Testing	80
3.5.1 Waveform Reproduction	83
3.5.2 Fundamental Frequency	86
3.5.3 ‘Vocal Tension’ Parameters	86
3.5.4 Automatic Pitch-Dependent Voice Types	90
3.5.5 Automatic f0 Trajectory	95
3.6 Conclusions	97
4. Vocal Tract Modelling	98
4.1 Vocal Tract Modelling with the 2D DWM	98
4.2 Implementation of the 2D DWM in MATLAB	102
4.3 Implementation in iOS	105
4.4 System Testing	109
4.4.1 Formant Analysis	109
4.4.2 Multiple Vowels	111
4.4.3 System Performance.....	113
4.5 Conclusions	114
5. Summary and Analysis	115
5.1 Summary	115
5.2 Analysis	116
5.2.1 Voice Source Synthesis using the LF-Model	116
5.2.2 Extensions to the LF-Model	116
5.2.3 Use within 2D DWM Vocal Tract Model	119
5.2.4 Core Aims	119
5.3 Future Research	121
5.3.1 Issues within LF-Model implementation	121
5.3.2 Further Extensions to the Voice Source Model	122
5.3.3 Multi-touch, Gestural User Interfaces	124
5.3.4 Implementation of Dynamic Impedance Mapping within the 2D DWM	126
5.4 Conclusion	126
Appendix A – ‘LFModelFull.m’ MATLAB Source Code	128
Appendix B – ‘ViewController.h’ LFGGen App Header File	133
Appendix C – ‘ViewController.m’ LFGGen App Main File	134
Appendix D – ‘AudioEngine.h’ LFGGen App Header File	136

Appendix E - 'AudioEngine.m' LFGGen App Main File	139
References.....	170

List of Figures

Figure 2.1	<i>The human vocal system</i>	7
Figure 2.2	<i>Cross-section of the human speech system</i>	9
Figure 2.3	<i>Glottal flow waveform and derivative</i>	10
Figure 2.4	<i>Comparison between F- and L- model waveforms</i>	22
Figure 2.5	<i>Annotated LF-model flow derivative waveform</i>	23
Figure 2.6	<i>Vocal tract represented as a series of tubes</i>	29
Figure 2.7	<i>1D Digital waveguide structure</i>	30
Figure 2.8	<i>Achieving a cross-sectional area function from MRI data</i>	31
Figure 2.9	<i>2D Digital waveguide mesh structure</i>	31
Figure 2.10	<i>Raised Cosine Function</i>	34
Figure 2.11	<i>2D and 3D DWM topologies</i>	36
Figure 2.12	<i>Wolfgang von Kempelen's 'Speaking Machine'</i>	39
Figure 2.13	<i>The 'Uncanny Valley' Effect</i>	42
Figure 3.1	<i>'Typical' LF waveform</i>	49
Figure 3.2	<i>'Typical' LF waveform with varying t_e value</i>	50
Figure 3.3	<i>'Typical' LF waveform with varying t_p value</i>	51
Figure 3.4	<i>'Typical' LF waveform with varying t_a value</i>	52
Figure 3.5	<i>LFGen app interface with CorePlot waveform display</i>	55
Figure 3.6	<i>'Breathy' voice waveform</i>	58
Figure 3.7	<i>Black box diagram for LFGen app</i>	59
Figure 3.8	<i>Software diagram for LFGen app</i>	70
Figure 3.9	<i>LFGen app interface (final version)</i>	79
Figure 3.10	<i>3D-printed vocal tract model</i>	81
Figure 3.11	<i>Modal voice type waveform and spectrum</i>	83
Figure 3.12	<i>Breathy voice type waveform and spectrum</i>	84
Figure 3.13	<i>Vocal Fry voice type waveform and spectrum</i>	84
Figure 3.14	<i>Falsetto voice type waveform and spectrum</i>	84
Figure 3.15	<i>'Typical' voice type waveform and spectrum</i>	85
Figure 3.16	<i>'Typical' voice type waveform with varying vocal tension</i>	87
Figure 3.17	<i>'Typical' voice type spectrum</i>	87
Figure 3.18	<i>'Typical' voice type waveform with minimum VT</i>	88
Figure 3.19	<i>'Typical' voice type waveform with maximum VT</i>	88

Figure 3.20	<i>‘Typical’ voice type waveform with varying t_a values</i>	88
Figure 3.21	<i>‘Typical’ voice type spectrum</i>	89
Figure 3.22	<i>‘Typical’ voice type spectrum with minimum t_a</i>	89
Figure 3.23	<i>‘Typical’ voice type spectrum with maximum t_a</i>	89
Figure 3.24	<i>Waveform of an f_0 sweep with ‘auto-voice’ enabled</i>	91
Figure 3.25	<i>Waveform between 24-52 Hz with ‘auto-voice’ enabled</i>	92
Figure 3.26	<i>Waveform between 52-94 Hz with ‘auto-voice’ enabled</i>	92
Figure 3.27	<i>Waveform between 94-207 Hz with ‘auto-voice’ enabled</i>	93
Figure 3.28	<i>Waveform between 207-208 Hz with ‘auto-voice’ enabled</i>	93
Figure 3.29	<i>Waveform above 288 Hz with ‘auto-voice’ enabled</i>	94
Figure 3.30	<i>Spectrogram of human /A/ vowel with varying f_0</i>	95
Figure 3.31	<i>Spectrogram of synthesised /A/ vowel with varying f_0</i>	96
Figure 4.1	<i>Spectrogram of synthesised /A/ vowel with ‘typical’ voice</i>	110
Figure 4.2	<i>English vowel chart</i>	111
Figure 4.3	<i>Xcode performance check</i>	113
Figure 5.1	<i>Idealised Vocal Fry waveform</i>	123
Figure 5.2	<i>HandSynth touchscreen interface</i>	125
Figure 5.3	<i>Proposed multitouch interface design</i>	125

List of Tables

Table 2.1	<i>Four voice types and their corresponding waveforms</i>	12
Table 2.2	<i>Four voice types with spectra, pitch range and noise amount</i>	14
Table 2.3	<i>Four male voice types and their timing parameter values</i>	25
Table 3.1	<i>Five LFGGen voice types and their timing parameter values</i>	56
Table 4.1	<i>Synthesised formants vs average English male speech formants</i>	112

List of Accompanying Material

The following material can be found on the accompanying data CD:

1. A PDF of this document
2. 'Audio Examples' folder – synthesised voice types and 2D DWM vowels:
 - a. 'Breathy110Hz.wav' - breathy voice type at 110 Hz
 - b. 'Falsetto110Hz.wav' - falsetto voice type at 110 Hz
 - c. 'Modal110Hz.wav' - modal voice type at 110 Hz
 - d. 'Typical3-bird.wav' - typical voice type with /3/ vowel
 - e. 'Typical110Hz.wav' - typical voice type at 110 Hz
 - f. 'TypicalA-bart.wav' - typical voice type with /A/ vowel
 - g. 'TypicalAe-Bat.wav' - typical voice type with /Ae/ vowel
 - h. 'TypicalI-beet.wav' - typical voice type with /I/ vowel
 - i. 'TypicalQ-bod.wav' - typical voice type with /Q/ vowel
 - j. 'TypicalU-food.wav' - typical voice type with /U/ vowel
 - k. 'VocalFryFu110Hz.wav' - vocal fry voice type at 110 Hz
3. 'Code Listings' folder
 - a. 'LFGenMkVI.zip' - compressed folder containing xcode project for LFGen iOS app
 - b. 'LFModelF0Data.m' - matlab script for producing a synthesised vowel for a given voice type with an f0 sweep taken from a voice recording
 - c. 'LFModelFull.m' – matlab script for producing any voice type with options for pitch, amplitude, duration, breathiness and vocal tension.
4. Demonstration video – 'LFGenDemoVideo.mp4'

Acknowledgements

To my parents, thank you for your constant love, support and encouragement throughout this project.

To my supervisor David Howard, thank you for the inspiring supervisions and general advice during this project and others throughout my time at York.

To Steve, Amelia, Laurence, Becky, Andrew, Tom, Eyal, Frank, Jude and Helena, thank you for some truly memorable crossword sessions during the Audio Lab lunch breaks, and the near-constant supply of cake.

To Jiajun, Ed and Simon, your patience and understanding with the often-frustrating life of a post-graduate researcher made our house a pleasure to come back to after many late nights in the library.

To Dimitri, your expertise and willingness to teach iOS and Core Audio helped this project materialise at a crucial point in the development stages.

Special thanks to my friends on both sides of the country, especially Benedict, Sam, JP, Ben, Mike, Annie and Rosie.

Author's Declaration

The work presented in this thesis is entirely the author's own, with any substantial external influences attributed in the text. None of the content in this thesis has been published by the author in any form. This work has not previously been presented for an award at this, or any other, University.

1. Introduction

The title of this thesis is **The Development of a Parametric Real-Time Voice Source Model for use with Vocal Tract Modelling Synthesis on Portable Devices**. The research project described herein is concerned with digital modelling of the human voice source to help improve the naturalness of existing speech synthesis technology. This thesis contains an analysis of existing voice source models, followed by a description of the development of a voice source modelling application for iOS devices.

This chapter introduces the key themes of this research, and the motivation for this specific project. An overview of the remaining chapters is given in section 1.2.

1.1 Thesis Overview

The human voice is the most expressive and versatile instrument we possess. Whether delivering a public speech, singing in a church choir or having a private conversation, the sheer flexibility of the vocal instrument allows us to convey a huge spectrum of human emotion, with the subtlest of expressive touches. It is not surprising that a totally accurate reproduction of the human vocal system has not yet been achieved. Apple's Siri software [1] is capable of producing speech output that, on a casual listen, can sound indistinguishable from human

speech, however the software's vocabulary is limited to pre-recorded voice sounds. The DECTalk system (commonly associated with Stephen Hawking's communication aid) [2] is instantly recognisable as a computerised or 'robotic' voice and has an unlimited vocabulary, as it can produce any speech sounds. The compromises inherent in both these systems are informed by the context in which they are used – Siri users do not rely on the software to communicate, but might prefer a pleasant voice. Users of communication aids such as DECTalk rely on the ability to convey any information in an efficient and intelligible manner, with naturalness or realism being of lesser importance.

The work described in this thesis is concerned with the idea of contributing to a voice synthesis system that is both *versatile* and *expressive*. This work takes into account the importance of the voice source (discussed in Chapter 2) in human speech production, and aims to look at ways in which a more sophisticated voice source model can be incorporated in existing speech synthesis applications.

The motivation for this research comes from two places of interest. Firstly, natural voice synthesis provides a fascinating research area, with inspiration from and implications for a variety of disciplines such as engineering, psychoacoustics, linguistics, voice pathology and even philosophy. The software developed for this work was designed predominantly as a research tool that could be used in any of these fields, as an input source for a new vocal tract model, for example, or as a means of exploring the nature of voice source variation in the perception of synthesised voices.

As well as a general interest in voice synthesis, the impact of related software for assistive technology applications is considered a major motivation for improving the technology in this field. This partly informed the decision to focus on portable devices such as tablets and smartphones, which, for some users of assistive technology, have become useful and often essential items [3] [4]. Whilst the goal of this work was never to develop a fully formed communication aid, it is hoped that the research and software described herein will contribute to future developments for such an application.

1.2 Thesis Structure

Chapter 2 provides a summary of the existing literature on topics related to this work. First, the ‘source + modifier’ model of speech production and voice synthesis is explained, followed by a description of voice source physiology. The main ‘voice types’ are then introduced, and an overview of voice source modelling is given. Vocal tract modelling techniques are then described, including a description of the digital waveguide mesh, which is used to model the vocal tract in this work. Finally, previous research projects on the subject of ‘naturalness’ are recounted to set the work in context.

Chapter 3 describes the majority of the development process for a parametric, real-time voice source model. The general motivation for this design is given as well as a technical specification. The design of the software is described, followed by an implementation report and system testing results.

Chapter 4 documents the process of porting an existing 2D digital waveguide mesh model of the vocal tract first to MATLAB and then iOS. Chapter 5 concludes the work, with an analysis of the project as a whole, followed by a brief exploration of potential future work on the subject.

2. Literature Review

This chapter summarises the key themes of the research undertaken, and discusses existing literature on the subject. The impetus for this research came from the conclusions from two earlier research projects [5] which dealt with the concept of ‘naturalness’ in voice synthesis, and made attempts to improve or explore this notion through real-time control.

During the initial stages of the current project, it was concluded that a different approach should be taken, namely improving the synthesis engine, rather than its interface. For the sake of completeness, and to place this work in context, a brief summary of these earlier studies and related literature is included. The relevant literature can, therefore, be split into four key areas:

- voice source physiology and acoustics
- speech synthesis and vocal tract modelling
- ‘naturalness’ in speech synthesis
- voice source modelling

The latter (voice source modelling) is the primary research area.

2.1 The ‘Source + Modifier’ Principle

Before discussing the physiology and acoustics of the voice source, it is necessary to define what is meant by the ‘voice source’ in relation to speech production as a

whole. It is widely understood that an appropriate analogue of the human speech system is its description consisting of a **sound source** with **sound modifiers**. Howard and Murphy [6] provide a detailed introduction to voice science, which encompasses everything from speech system physiology to speech and singing recording techniques. This book includes another component to the 'source and modifier' analogue: the 'power source', being the lungs. It is important to include the power source when considering human speech production, however in synthesised speech, the airflow from the lungs is (usually) not incorporated into the synthesis engine, so a single 'voice source' can be considered as an approximation of the waveform created when the airstream resulting from lung pressure acts on the vocal folds. The sound modifiers are the acoustic cavities between the glottis and the lips (the vocal tract), and the articulators (the tongue, lips and jaw), which modify the voice source signal by acoustically filtering certain frequencies, and creating speech components such as consonants. Figure 2.1 displays a cross-section of the human vocal system, detailing power and noise source, compared with the voice source waveform created when they act together.

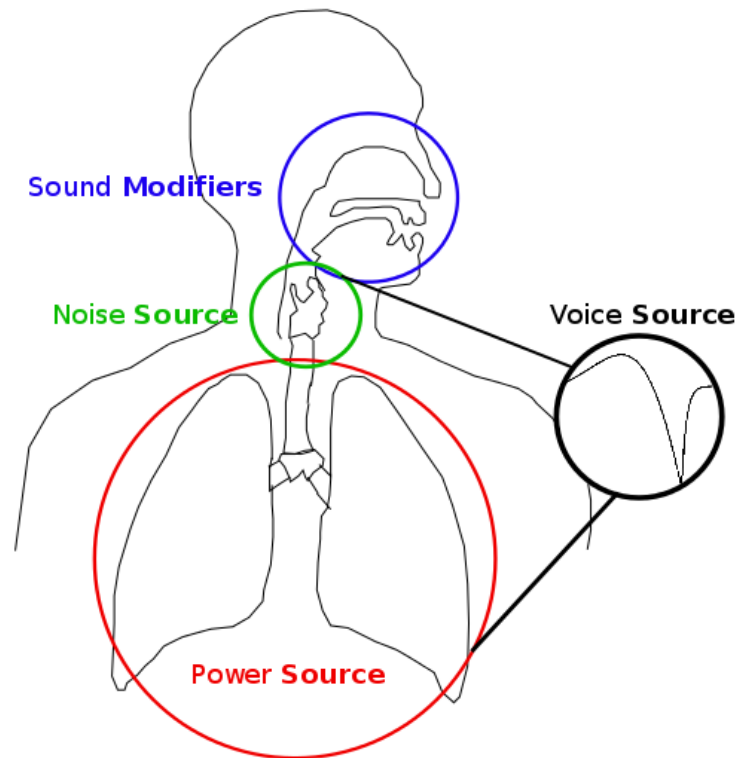


Figure 2.1 – The human vocal system with waveform of voice source (equivalent to power source + noise source)

It should be noted that, in reality, the vocal folds and vocal tract do not act fully independently of each other [7], and a truly accurate model of the speech system would take into account the cross-coupled relationship between the vocal tract and vocal folds [8]. Most existing voice source models remain fairly rudimentary, staying faithful to the discretised model presented above [7] [9]. There are advantages and disadvantages to both approaches - complex, cross-coupled, physical models are able to replicate the behaviour of the vocal folds under certain conditions, at the expense of computational ease. Rudimentary mathematical models of the glottal flow waveform can be more computationally efficient, at the expense of realistic behaviour under certain conditions. However,

the flexibility given by these models allows for increased functionality in terms of acoustic responses to given conditions.

2.2 Physiology of the Vocal Folds

Fig. 2.2 below shows a cross-section of the voice production system in humans. Voice production begins at the diaphragm below and the intercostal muscles surrounding the lungs. At rest, the diaphragm is bowed upwards, and flattens out when constricted. When the diaphragm is constricted and the intercostal muscles expand the ribs, air enters the lungs. Breathing out requires the lungs to be compressed in some manner, through contraction of the intercostal or abdominal muscles [6]. Airflow from the lungs then passes towards the glottis. The glottis is the area between the vocal folds. The vocal folds are described as 'the vibrating elements in the larynx' [6] and are the two mucosal membranes that traverse either side of the glottis, and meet in the middle to close the larynx completely. The prevailing theory for the kinematic process of vocal fold vibration is attributed to the Bernoulli effect [10]. This is the same process that is used to describe lift in aeroplanes, helicopters and aerofoils, and occurs when an airstream passes over a curved surface, creating an area of low pressure due to the faster airstream closer to the curve. When air passes through the glottis, the vocal folds are forced open. The curvature of the open folds creates an area of low pressure in between and below them, drawing the folds back together. This process repeats, creating a constant oscillation. It should be noted that more recent research has discredited the use of the Bernoulli effect to explain phenomena such as aerofoil lift and vocal fold vibration. In [11], Babinsky

explains the fallacy of invoking the Bernoulli equation, but an in-depth discussion of this is outside the scope of this thesis. To put it briefly, the Bernoulli equation can only legitimately be used when all airstreams originate from the same source. In the case of vocal folds, where the airstreams above and below the glottis have different origins (from the lungs below and the area above glottis), Bernoulli's equation cannot be used to describe the behaviour of both airstreams simultaneously.

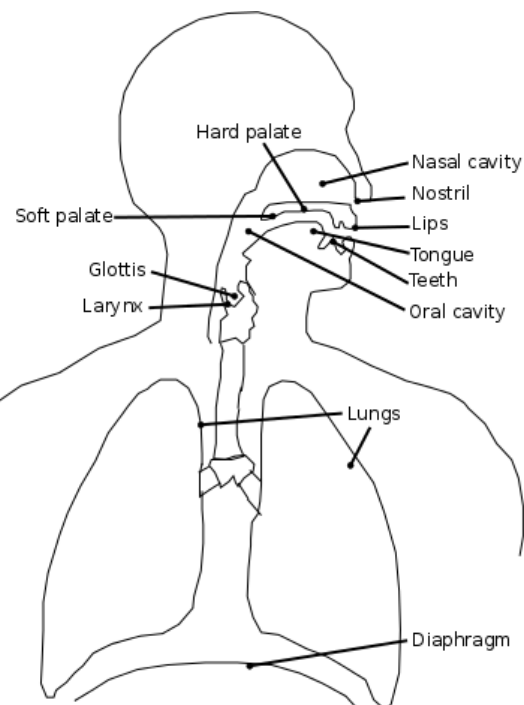


Figure 2.2 – Cross-section of the human speech system

The muscles surrounding the glottis alter the tension of the vocal folds. Like a stringed instrument, a change in tension causes slower or faster oscillations - in other words, a change in pitch or frequency. The frequency at which the vocal folds oscillate is the fundamental frequency of any voicing produced. The terms

glottal flow and *glottal flow derivative* are used throughout the literature to describe the observed glottal pulse waveform obtained via inverse-filtering and its numerical derivative. The glottal flow derivative waveform takes into account the effects of lip radiation, which can be modeled as a first-derivative filter [9].

Figure 2.3 displays the glottal flow waveform compared with its derivative:

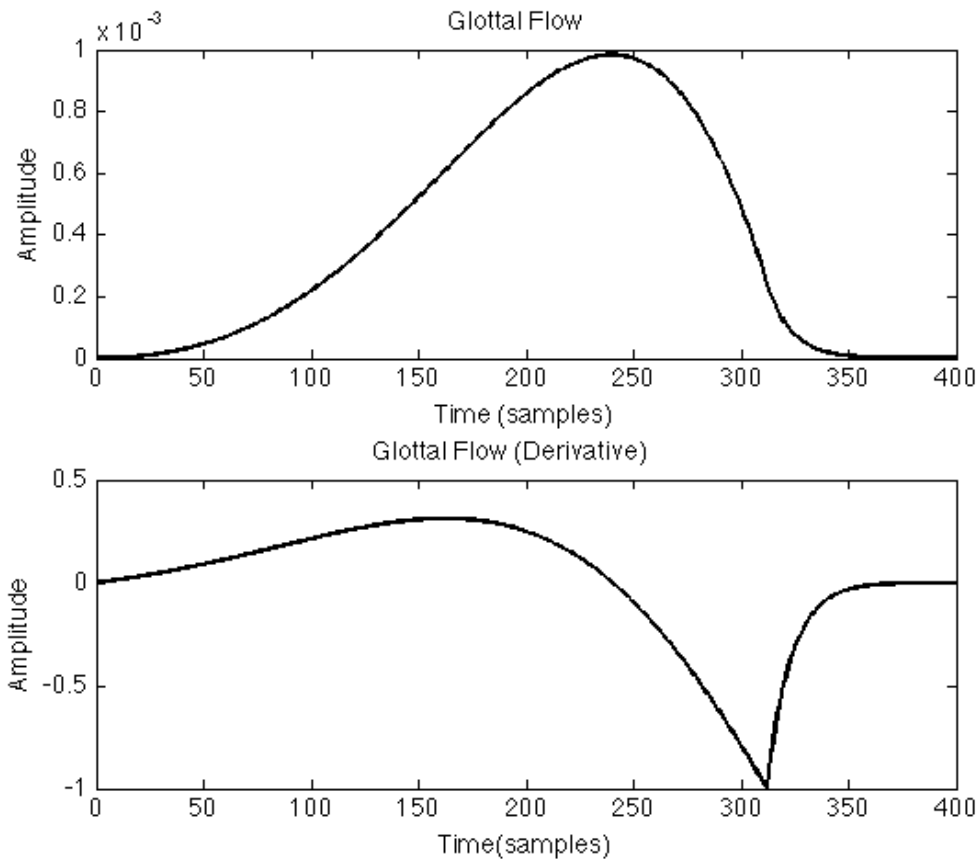


Figure 2.3 – One full pitch period of the glottal flow waveform and its numerical derivative

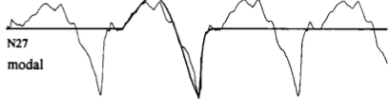
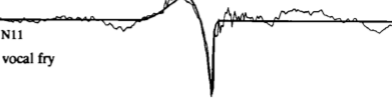
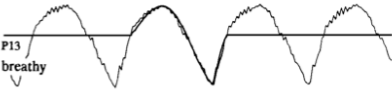
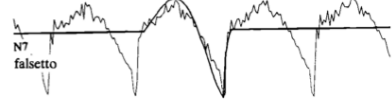
The waveform displayed above is an approximation of the true acoustic waveform, using the Liljencrants-Fant glottal flow model [12]. This is a mathematical model of the voice source waveform, which will be discussed later in this chapter. Fant's earlier work on the acoustic and physical properties of the

voice source [7] [13] highlighted the complex, interactive nature of the role of the vocal folds within the vocal system. He showed that the voice source is not merely a function of a pitched vocal fold vibration, but was also dependent on the speaker's physiology, impedance load from sub- and supra-glottal air pressure, and even the current vowel being spoken [7].

2.3 Voice Types

The *voice type* is a factor of voiced speech that is defined by the voice source. The speaker's age, gender, physiology, mood and setting all contribute to the overall acoustic properties of the glottal flow waveform, and thus the overall speech output. Childers and Lee [14] cite six distinct voice types: modal voice, vocal fry, falsetto, breathy voice, harshness and whisper. In their study, harshness and whisper were excluded due to the lack of periodicity in both voice types. The four voice types are presented in table 2.1, along with inverse-filtered voice source waveforms and their approximated LF-model fits.

Table 2.1 - Four voice types and their corresponding waveforms (LF-model fits to inverse filtered glottal source recordings from [14])

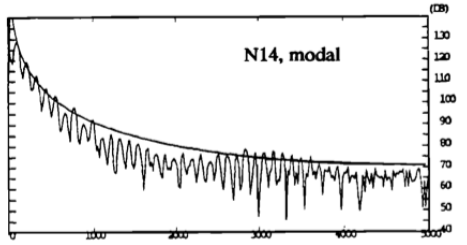
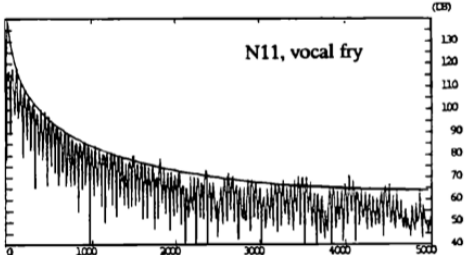
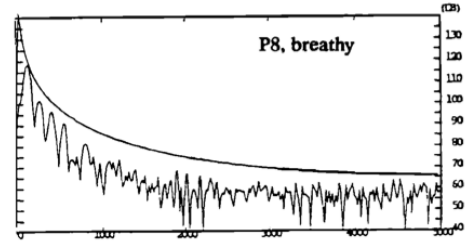
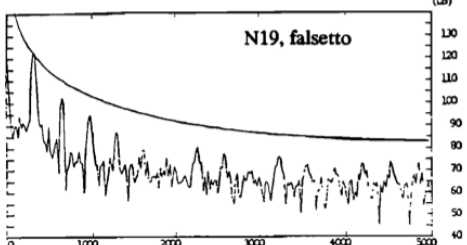
Voice Type	Description	LF-model waveform
Modal	The most commonly used voice type for speech and singing. Also referred to as the 'typical' voice type, most cultures and languages make use of the modal voice for everyday phonation. Little to no turbulent airflow present, meaning no high frequency noise component in the waveform [14]	
Vocal Fry	Commonly employed to achieve lower frequencies than is possible using a modal voice (although can extend into the modal pitch range as well). Characterised by very short glottal bursts followed by a large <i>closed quotient</i> [14]	
Breathy	During breathy voice phonation, the vocal folds do not fully seal the glottis, allowing an amount of turbulent airflow. This can be perceived as a high-frequency noise component during the <i>closing</i> and <i>opening stages</i> of the glottal flow cycle. [14]	
Falsetto	Created by only vibrating a small portion of the vocal cords, this allows the speaker/singer to achieve a much higher frequency range than modal voice. A noise component is also present due to a lack of complete closure at the glottis. [14]	

Childers and Lee found that the voice type could be characterised by four main factors, namely glottal pulse width, glottal pulse skewness, abruptness of glottal closure, and turbulent noise [14]. 'Glottal pulse width' refers to the portion of the

waveform where the glottis is open, also known as the *open quotient*. In terms of the glottal flow derivative, the open quotient 'is estimated by the time duration between a positive peak and the next adjacent negative peak' [14]. Glottal pulse skewness (or the *speed quotient*) refers to the relationship between the lengths of the opening phase and the closing phase. Abruptness of glottal closure and turbulent noise refer to the steepness of the return phase of the waveform and the high frequency noise created by airflow through the glottis respectively.

Table 2.2 shows the approximate spectrum, fundamental pitch range and turbulent noise properties for the four voice source types.

Table 2.2 - Four voice types with spectral content, pitch range, and noise component information

Voice Type	Spectrum (Diagrams taken from [14])	Range (approx. male voice)	Noise Component
Modal		~52-207 Hz	None
Vocal Fry		~24-94 Hz	None
Breathy		~52-207 Hz	Noise present at around 5% of total signal
Falsetto		~207-440 Hz	Noise present at around 5% of total signal

The voice source type (also referred to as ‘voice quality’) has been shown to play a major role in the perception of emotion and stress in speech [15]. Whilst there have been many empirical studies analysing the nature of these voice qualities, Gobl states that ‘very few ... have focussed on the voice source correlates of affective speech’. In Gobl’s study, a recording of an utterance spoken in Swedish was inverse-filtered to obtain an approximation of the voice source waveform. A

voice source model was then fitted to this approximation, which allowed for parameterisation of the voice source to fit seven voice qualities. The voice source model was then used to drive a formant synthesiser, and the original recorded phrase was resynthesised for each voice quality. The resynthesised utterances were played to a number of non-Swedish speaking subjects (so that the emotional context of the words would not influence the subject's perception of emotion). It was found that the perceived 'tenseness' of the voice source influenced the listener's perception of emotional content in the voice, although this was shown to be far more effective for some emotions (relaxed/stressed, bored, intimate, content) than others (happy, friendly, sad, afraid).

Chen discusses the glottal gap phenomenon in [16]. This is a feature of the voice source that occurs when the glottis does not fully close, such as in breathy or falsetto phonations. It was found that the size of the glottal gap relative to the pitch cycle affected the overall speech output to a significant degree, in terms of the perceived voice quality. Most affected were the spectral tilt and the turbulent noise component, both of which increased proportionally with the size of the glottal gap.

Though not a distinct voice type in and of itself, vocal vibrato is a common vocal feature that originates at the voice source, primarily used in singing. Sung phrases are typically of the modal or falsetto voice types (although vocal fry is somewhat prevalent in pop singing). In [17], the perceptual benefits of vocal vibrato are discussed. One such benefit is the effective 'gluing' of partials, or harmonics together. For example, while vocal sounds are generally perceived as

a homogenous blend of harmonics, it has been shown that, at a fixed pitch, it is possible to discern between separate partials present in the speech signal [17]. When the f_0 is constantly varied, as in vocal vibrato, these separate partials are 'glued' together again. Another hypothesised perceptual effect of vocal vibrato is the increased intelligibility of vowels when vibrato is present. As Sundberg states, it is reasonable to assume that as the harmonics above the fundamental frequency undulate in time with the f_0 , those harmonics present around vowel formant frequencies will reinforce the perception of the formant. This is due to the amplitude modulation of these harmonics as they align with the formant frequency. Counter-intuitively, further studies failed to prove this effect conclusively [18] although during the current research it was also found that subjective responses to a synthesised voice with varying pitch were much more favourable than a constant f_0 .

2.4 Modelling the Voice Source

Any source/modifier approach to synthesising the human voice will employ some form of voice source model. These can be fairly rudimentary, such as a simple saw-wave or pulse-train [19] [20], to resynthesised human voice source waveforms obtained via inverse-filtering [21]. As Chen et al. point out, 'few studies have attempted to systematically validate glottal models perceptually, and model development has focused more on replicating observed pulse shapes than on perceptual sufficiency' [22]. Fitting existing models to observed pulse shapes is so far the most reliable method for achieving accurate recreations, due to the impracticality of capturing an isolated voice source waveform using

conventional recording methods [15] - this has been attempted, but the highly invasive procedure involved miniature transducer microphones inserted between the vocal folds, which necessitated the use of local anaesthetic [23]. Inverse filtered glottal pulse signals and LX-waveforms obtained via laryngoscope [21] [24] are the most common references used for voice source modelling. This sub-section summarises attempts made to recreate this signal using mathematical modelling and other techniques.

2.4.1 Existing Methods for Voice Source Synthesis

In order to produce the formants that occur in natural speech, a complex source waveform with sufficient harmonics must be used. It has been recognized since at least the 1970s [25] that a source waveform approximating that found in natural speech would provide the most accurate speech output. While it is possible for very simple formant synthesisers to achieve speech-like results using saw-waves, square waves, or even white noise as an input, the spectral content of the glottal source signal is of significant importance to the overall naturalness of the synthesised speech content. Rosenberg was one of the first to compare differing methods of speech synthesis excitation using time-domain representations of the source waveform. He showed that out of six wave shapes of varying complexity, a complex trigonometric waveform, based on observations of glottal pulse movement and speech recordings was the most preferred in a listening test, when compared with a natural speech recording.

In [9], the distinction is made between:

- 1.) *non-interactive parametric glottal models* - mathematical models that assume a linear separability between the voice source and vocal tract,
- 2.) *interactive mechanical and parametric glottal models*, which are based on the interaction between the vocal source and the rest of the vocal system, either via a mechanical model or numerical simulation, and
- 3.) *physiological glottal models*, in which an attempt is made to accurately simulate the physical properties of the vocal folds in three dimensions.

Non-interactive parametric glottal models are intuitively the simplest to achieve, requiring only knowledge of the voice source waveform and its spectrum. Early studies such as Rosenberg's [25] confirmed that as the glottal pulse shape approached similarity with that observed through inverse-filtering techniques, the perceived quality of voice synthesis improved. In these early studies, the glottal flow waveform was modeled, rather than the glottal flow derivative. Liljencrants and Fant [12] were one of the first to apply the first-derivative filter to the glottal pulse model in order to simulate the effects of lip radiation. They developed a parameterised model of the glottal flow derivative, known as the Liljencrants-Fant or *LF Model* which is now the most commonly used among the non-interactive parametric models [9]. Due to the model's flexibility and ease of adaptation to existing speech source waveforms, it has been widely accepted as the standard voice source model for speech processing and analysis [14]. The LF model has provided the basis for this research, and so will be further analysed later in the chapter. Other parameterised models of the glottal flow derivative have been developed, such as Fujisaki and Ljungqvist's model [26] which was

shown to be equally successful in minimising the linear predictive error when directly compared with natural speech as the LF model, however due to the computational complexity in calculating this model, the LF model is generally favoured [9] [14].

Cummings et al. state that

'although simple non-interactive glottal models produce intelligible synthetic speech and are adequate for many coding and analysis tasks, very high-quality speech synthesis and complex speech analysis necessitate the ability to model glottal excitation more accurately' [9]

Cummings summarises these methods, which are achieved numerically or via equivalent-circuit design. Two common effects of source-tract interaction that are included in these models are the effects of low first-formant frequencies on the vocal tract's impedance load and the glottal pulse ripple.

The most complex form of glottal source model is the *physiological glottal model*. Titze and Talkin [27] [28] developed a four-parameter mathematical model of the glottis based on earlier theoretical work by Titze [8] [29]. This is essentially a mass-and-spring mathematical model of the physiology of the vocal folds, which takes into account the following:

- *'abduction quotient*, a measure indicating the amount of adduction or abduction of the vocal folds,

- *shape quotient*, a measure of the shape of the pre-phonatory glottis (converging, diverging, or partly converging and partly diverging)
- *bulging quotient*, a measure representing the amount of medial surface bulging of the vocal folds, and
- *phase quotient*, a measure of the phase delay between the upper and lower edges of the vocal folds.' [9]

Physiological glottal models such as these are capable of creating a highly sophisticated representation of the glottal flow. However, a precise knowledge of glottal physiology is required in order to use models such as these, as the glottal volume velocity waveform is an indirect result of the model, as opposed to the simpler model types which attempt to recreate the volume velocity waveform directly.

2.4.2 The Liljencrants-Fant Glottal Flow Model

As discussed, the Liljencrants-Fant (or LF-) model is one of the most widely used glottal flow models in voice synthesis and speech processing applications. This is largely due to its relative computational ease and parameterisation. Earlier work by Gunnar Fant [7] established a foundation for this model by observing predicted glottal flow volume velocity waveforms from inverse-filtered recordings of connected speech. Findings from this study allowed Fant to develop an early two-parameter glottal flow model (called the F-model). This early model comprised a rising and descending branch around the boundary between the opening and closing phases. The F-model contained a discontinuity

at the flow peak (Fig. 2.4), so a more sophisticated model was sought. The three-parameter *L-Model* developed by Liljencrants was used as a starting point.

The advantage of the L-model over the F-model is its continuity, which means that no secondary weak excitations are present in the acoustic waveform. The L-model also displayed less spectral ripple than the F-model. Neither models incorporated a term for the gradient of the return phase of the glottal pulse, which was found to be crucial for modelling certain voice types and phonations [12]. For example, during a voiced 'H' sound, the glottis remains open for most of the pitch cycle, allowing turbulent airflow to create the high-frequency noise component (also observed in breathy and falsetto voice types). In order to model voice source effects such as these, an exponential return phase whose gradient was a fourth parameter, based on observations by Liljencrants, Fant and Ananthapadmanabha [12] [7] [13] was added.

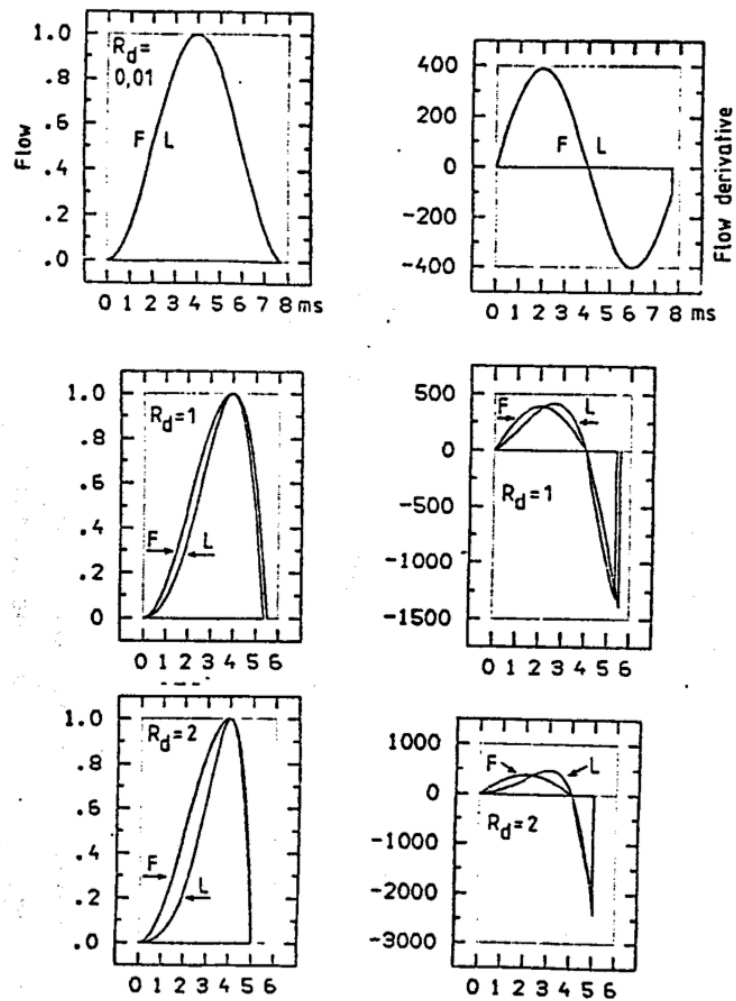


Figure 2.4 - Comparisons between the F- and L- glottal model waveforms (left) and their derivatives (right) with varying values of R_d - a 'shape parameter' based on the amplitude and position of the positive peak - taken from [12]

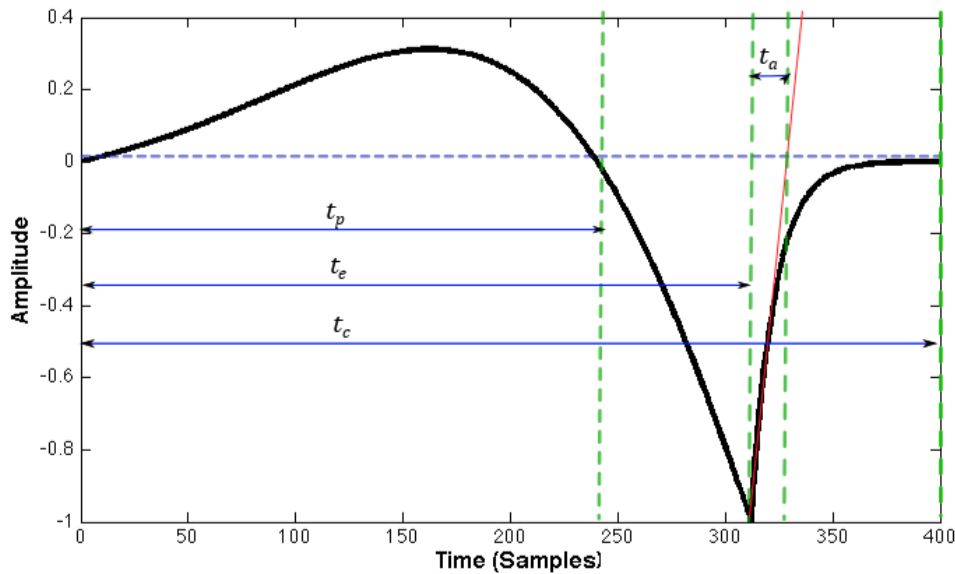


Figure 2.5 - LF-model glottal flow derivative waveform with timing parameter annotations. The value for t_a is the distance between point t_e and the zero crossing of the derivative of the return curve (red line on graph).

Figures 2.4 and 2.5 show one pitch period of each of the aforementioned glottal models. The timing parameters t_p , t_e , t_a , and t_c are shown on the LF-model diagram. These four timing parameters can be modified in order to fit existing glottal flow measurements for speech analysis, or to synthesise new waveforms in order to simulate different voice types in speech synthesis. These timing parameters are defined as a percentage of the overall pitch cycle length T_0 . Parameter t_p describes the length of the opening phase of the cycle, i.e when the vocal folds are moving upwards and the glottis is opening (the moment of maximum flow). T_e gives the timing of the negative peak in the waveform, which occurs at the beginning of the return phase. T_a gives the effective duration of the return phase, calculated by the length of time between t_e , and the zero-crossing of the derivative of the return slope at t_e . T_c describes the length of the open

phase, or the portion of the pitch cycle during which the vocal folds are in motion. If t_c is less than T_0 , the remainder of the waveform between t_c and T_0 is known as the closed phase. One requirement of the LF-model is that the overall net gain of flow during a pitch period must equal zero:

$$\int_0^{T_0} LF(t) dt = 0 \quad [2.1]$$

The waveform is calculated in two stages. The first stage involves an exponentially growing sinusoid between the moment of glottal opening ($t = 0$) and the negative peak at $t = t_e$. An exponential component describes the second stage - the return phase between t_e and t_c . The two equations for the LF-model waveform can be written as

$$LF(t) = E_0 e^{\alpha t} \sin(\omega_g t), 0 \leq t \leq t_e \quad [2.2]$$

$$LF(t) = -\frac{E_e}{\varepsilon t_a} [e^{-\varepsilon(t-t_e)} - e^{\varepsilon(t_c-t_e)}], t_e \leq t \leq t_c \leq T_0 \quad [2.3]$$

Where E_0 describes the maximum positive flow, E_e the maximum negative flow, α and ω_g are respectively the exponential growth factor and the angular frequency of the sinusoidal component, and ε is the exponential time constant of the return phase. In order to maintain the area balance condition described in equation 2.1, ε then E_0 and α are solved iteratively so that the following conditions hold:

$$\varepsilon = \frac{1 - e^{-\varepsilon(t_c-t_e)}}{t_a} \quad [2.4]$$

$$E_0 = -\frac{E_e}{e^{\alpha t_e} \sin(\omega_g t_e)} \quad [2.5]$$

(analysis of LF-Model equation based on Jack Mullen's summary [30])

By manipulating the values of the timing parameters (t_c , t_e , t_p , t_a), the LF-model can be modified to describe certain voice types, or matched to pre-recording voice source data. 'Voice quality factors: Analysis, synthesis, and perception' [14] is an example of one of the many studies into voice synthesis and analysis that have used the LF-model in an attempt to synthesise different voice types, as well as establish the role of various LF-parameters in terms of the perception of the synthesised voice. Beginning with inverse-filtered speech waveforms and data from electroglottographic recordings, Childers & Lee [14] analysed the spectral content and waveform characteristics of four voice types - modal, breathy, vocal fry and falsetto. From earlier studies [31], it was found that the LF-model provided a convenient and efficient basis from which to recreate the timing and spectral characteristics of the four voice types. By adjusting LF-parameters to fit the initial recordings, then optimising the LF-model estimate using a least-mean-squared error criteria, average values of the LF-parameters for different voice types were found:

Table 2.3 – Four male voice types and corresponding timing parameter values

(taken from [14])

	T_e (%)	T_p (%)	T_a (%)	T_c (%)
Modal	55.4	41.3	0.4	58.2
Breathy	57.5	45.7	0.9	100
Vocal Fry	59.6	48.1	0.27	72
Falsetto	89	62	4.3	n/a

By modifying each timing parameter in turn followed by the overall pulse width (open quotient or OQ) and pulse skewing (speed quotient or SQ), keeping all other parameters fixed, and synthesising short vowels using a Klatt formant synthesiser [32], it was possible to evaluate the perceptual effects of each parameter, and to establish which were most useful for synthesising different voice types. Criteria for simulating hypo-/hyperfunction (lax/tense vocal quality) were established in the time and frequency domains, with a high SQ creating more high frequency energy, contributing to a perceptually more tense voice quality. This study also incorporated a noise generator, in order to simulate breathiness. It was found that white noise, high-pass filtered at 2 kHz added to the LF-model signal contributed to the perception of breathiness. Modulating the noise signal's amplitude so that it was present during 50% of the pitch cycle (roughly lining up with the closed part of the vocal fold oscillation), with a noise-signal ratio of 0.25%, provided the best results for simulating breathiness. This study confirmed the importance of a variation in voice source quality in natural speech synthesis, and concludes that 'various intonation and stress patterns may be correlated to source parameters other than fundamental frequency and timing' [14]. This idea is the primary concept behind the current research, which is aimed at developing a more natural, dynamic and user-configurable voice source for voice synthesis applications.

Whilst the least-mean-squared-error technique described in [14] and [33] provides a close fit of the LF-model waveform to a glottal source recording, further research has been undertaken to optimise the timing parameter values in order to more accurately recreate voice source qualities [34] [21] [24]. One such

method is described in [21], known as Extended Kalman Filtering (EKF). EKF is an iterative error correction method that makes use of a priori estimates to converge on an optimum estimate. By incorporating the EKF equations in those describing the LF-model, it is possible to calculate α and ε values to achieve an optimum model fit. Further research into EKF techniques for model fitting also generated a time-domain fitting algorithm using EKF that was shown to be far more accurate than a previously used standard algorithm [35]. The timing parameters described in [21] were originally obtained from [24], which describes the use of a pitch-synchronous model-based glottal source estimation method to obtain an accurate set of mean values for LF-parameters from an inverse-filtered glottal source waveform.

In [36] the many parameters used to describe the glottal source model are investigated and their importance in terms of vocal quality perception is explored. It is acknowledged that 'the closing phase constitutes the main excitation of the vocal tract'. The closing phase, or normalised amplitude quotient (NAQ), describes the phase of the pitch period from the negative peak to the point of glottal closure. The authors recommend varying the NAQ for the largest and most effective perceptible variation in voice type. These findings are corroborated in [37] [38] and [39].

2.5 Vocal Tract Modelling

As well as the voice source, the physical properties of the vocal tract can be mathematically modeled in order to recreate its acoustic effects. This method of voice synthesis falls into the category of ‘articulatory speech synthesis’. [40] gives the following definition for articulatory synthesis:

‘Articulatory speech synthesis models the natural speech production process as accurately as possible. This is accomplished by creating a synthetic model of human physiology and making it speak.’ [40]

Palo acknowledges that articulatory speech synthesis methods are less effective at creating intelligible speech when compared with concatenative synthesis, but vastly more flexible in terms of the range of speech-like vocalisations that are available. The first example of synthesised speech created by a vocal tract model was developed by Kelly and Lochbaum in the 1960s [41]. This was a fully digitised acoustic model of the human vocal tract, achieved by discretising the vocal tract into a series of concatenated tubes (fig. 2.6). The travelling wave solution for each tube was obtained, and then digitised using Nyquist’s sampling theorem. Vocal tract area data was obtained via x-ray for several vowel sounds, and the cross-sectional area of each tube section of the model was proportional with the corresponding vocal tract area. This was one of the first and most enduring examples of *physical modelling synthesis*.

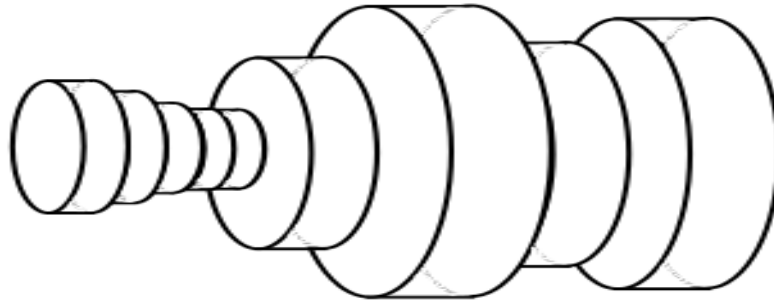


Figure 2.6 - Representation of the vocal tract idealized as a series of concatenated acoustic tubes, with glottis end at the left and lips at the right. Note that the 'bend' in the vocal tract that occurs above the glottis is not included in this representation.

The advances made in computing by the 1980s meant that new methods of physical modelling synthesis were being experimented with. One such method that had implications for vocal tract modelling was digital waveguide synthesis. Julius Orion Smith III describes the early conception of the one-dimensional digital waveguide in [42]. As d'Alembert first pointed out, the vibration of an ideal string can be described as the sum of two travelling waves going in opposite directions [43]. The conception of the digital waveguide is based on this principle. A digital waveguide is essentially a bi-directional digital delay line, with the sample propagation travelling in opposite directions (fig. 2.7). This approach allows for an efficient discrete-time simulation of the traveling wave solution, which can be used to model 'any one-dimensional linear acoustic system such as a violin string, clarinet bore, flute pipe, trumpet-valve pipe, or the like' [44]. Terminations and changes in impedance along the acoustic system can be modelled using boundary conditions and scattering junctions. A termination

(for example a bridge on a guitar) can be modelled simply by inverting the phase of the incoming signal, which acts as a total reflection of the displaced wave. Changes in impedance are modelled using the Kelly-Lochbaum scattering junction. Conservation of mass and energy dictates that for a change in impedance (such as from a narrow to a wide section of tube), the pressure and volume velocity variables of the travelling wave must be continuous [44]. This means that some of the acoustic energy will be transmitted across the impedance discontinuity, and the remainder will be reflected back. This is achieved digitally via the scattering junction.

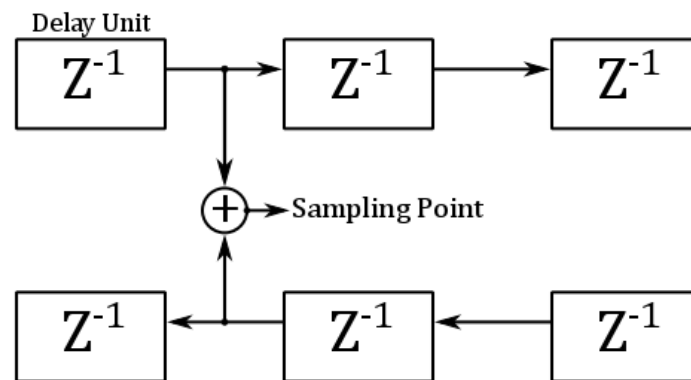


Figure 2.7 - 1D Digital Waveguide Structure. Sample delay units (marked z^{-1}) propagate an input signal in left and right directions, with changes in impedance modelled by attenuating the signal between delay units. Sampling points extract the current sample and a particular space along the DWG – similar to a pickup along a guitar string.

The 1D digital waveguide models changes in cross-sectional area in the vocal tract as a series of impedance changes in a 1D linear acoustic system. A 2D extension of this method, known as the *2D Digital Waveguide Mesh (DWM)*

models the same cross-sectional area function as a 2D plane, with width-wise delay lines of varying length, as seen in figures 2.8 and 2.9.

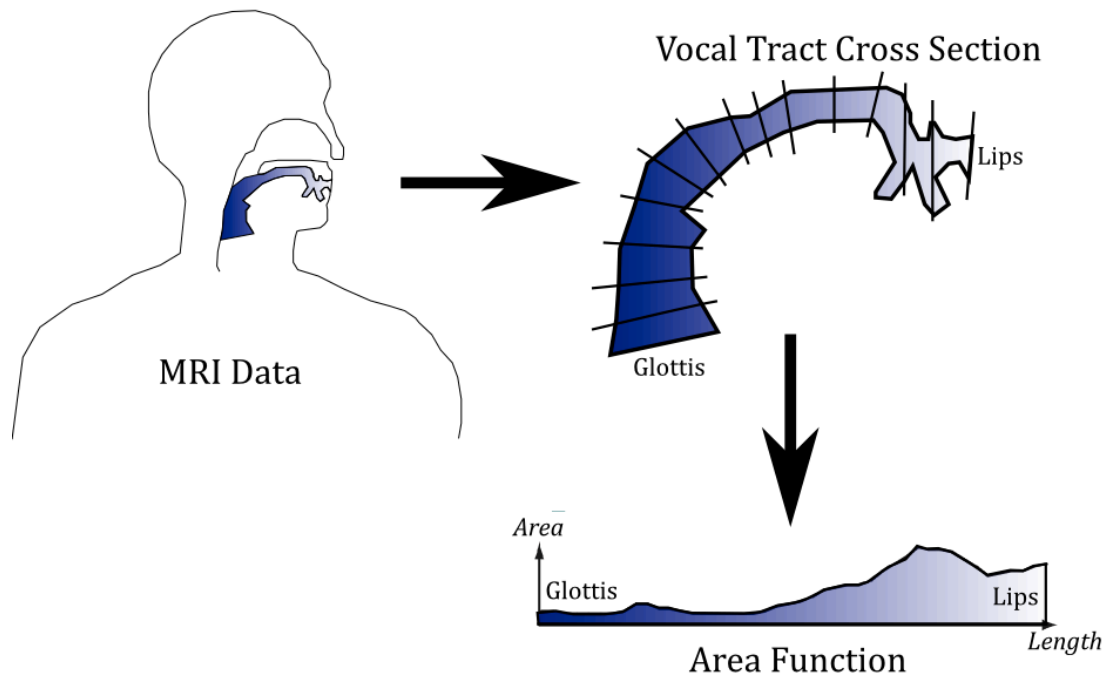


Figure 2.8 - Achieving a cross-sectional area function from MRI data. Note the lack of nasal cavity in the vocal tract cross-section and the 'straightening' of the track when converted to an area function.

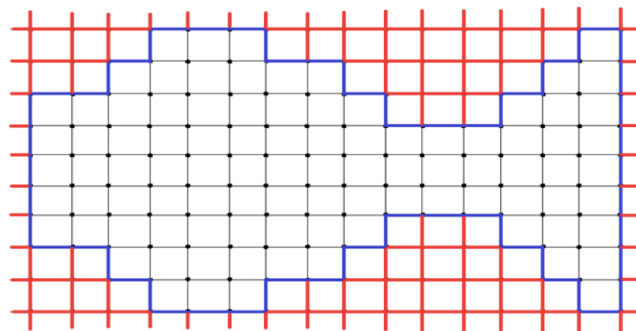


Figure 2.9 – 2D Digital Waveguide Mesh structure with impedance mapping, with glottis end at the left and lips at the right (red indicates a high impedance, creating effective 'boundaries' [highlighted in blue])

The 2D DWM was developed by Van Duyne and Smith in the early 1990s [45] with further development at the Audio Lab at the University of York [46] [47] [48] [49] [50] [51]. The DWM structure is ideally suited to modelling the propagation of acoustic waves across a membrane or plate, although the extra dimensionality is also an advantage over the 1D waveguide for modelling other acoustic systems. In the example of vocal tract modelling, the cross-sectional tract area can be directly modeled as a widthwise number of waveguide points, as opposed to the 1D solution, which requires a conversion from area to impedance. Inputs and outputs to the system can also be included at spatially meaningful points on the mesh, due to the analogous topography of the mesh to the modeled surface [46]. As Mullen points out,

‘it should be noted that the magnitude of vibrations is the physical variable under simulation that would be observed in the real-world system. The bi-directional travelling wave components are a hypothetical consideration to facilitate propagation’. [30]

Waveguide mesh topographies are not limited to a grid layout as illustrated above, and other arrangements of delay lines and scattering junctions have been experimented with [50].

An extensive study into vocal tract modelling using the 2D DWM is described in [30]. This thesis describes the theory behind digital waveguide mesh modelling, and its application to vocal tract modelling. It also describes the development of a novel method of modelling dynamic area function changes in real time, known

as *dynamic impedance mapping*. Conventional waveguide mesh structures follow the layout of the acoustic area they are modelling. Vocal tract modelling requires a more flexible method, as the layout is constantly changing depending on the current articulation. Dynamic impedance mapping allows the mesh size and shape to remain constant, while manipulating the impedances at each node to effectively alter the shape of the area through which acoustic energy can propagate. This is much less computationally expensive than altering the layout of the mesh at each sample step, and allows for real-time, dynamic articulations.

The process for vocal tract modelling using the digital waveguide mesh is as follows:

1. Obtain cross-sectional area function data of the vocal tract for a set of specific vowels. This is achieved using a magnetic resonance imaging (MRI) machine (see Figure 2.8 above).
2. Convert area function data to a series of discrete area values at regular intervals along the tract.
3. Calculate size of a single waveguide. This is related to the theoretical distance an acoustic wave would propagate during one sample length. It is calculated using the following formula:

$$\sqrt{2} \times c / f_s$$

[2.6]

Where c is the speed of sound and f_s is the sampling frequency.

4. Calculate the size of the waveguide mesh in terms of the number of individual waveguides in the x and y direction. The average dimensions for a male vocal tract are 17.5 cm long and 5 cm wide.
5. Interpolate area function data from original number of values to number of waveguides in x direction. Invert each value to obtain the related impedance value for each cross-section.
6. The impedances of the width-wise waveguides (y direction) are calculated using a raised-cosine area function (fig. 2.10). This was found to be the ideal solution for maintaining an open 'channel' in the middle of the mesh (i.e. at minimum impedance) with maximum impedance at the outer edges of the impedance map. This means at each point in the x direction, for a DWM n waveguides wide, a raised cosine function of n samples is created. Each point in the y direction is assigned an impedance value based on the corresponding raised cosine value.

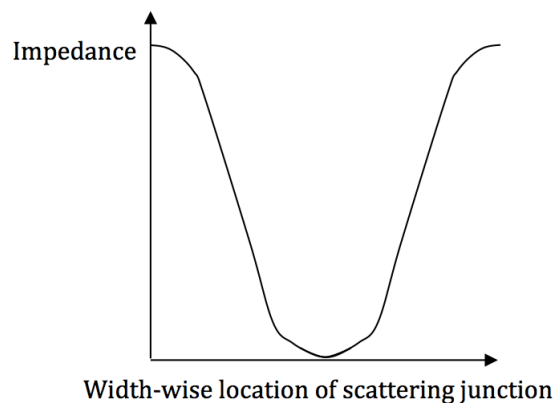


Figure 2.10 - Approximation of raised cosine function, with minimum impedance (Z_{min}) at the centre and maximum (Z_{max}) at the edges

7. The averages between adjacent points in the mesh are taken, and the impedance map is updated based on these averages. The pressure at the current junction (average of all surrounding points) is taken, and the outgoing pressures are calculated.
8. At every timestep, the incoming pressures to each junction are calculated based on the previous pressure values at surrounding points. Boundaries are modeled in the same way as a termination in a 1-D waveguide, for each outer point in the mesh. At the glottis end, the incoming pressure for each junction is excited with the current sample of the input waveform (i.e the voice source).
9. Finally, the output pressure is taken as the sum of all rightmost junctions multiplied by the lip radiation.

The impedance-mapped 2D DWM was excited with Gaussian noise to obtain a frequency response for several vowel area functions. The results showed that formant frequencies obtained from the 2D DWM varied in accuracy when compared with average formant frequency values for male speakers. For some vowels, the 2D DWM formant frequencies were less in line with average values than the 1D waveguide counterpart. It is acknowledged that these average formant values are not definitive, and the strongest case for vowel accuracy would be a perceivable similarity to the simulated vowel, based on subjective listening results.

The increased dimensionality introduced by the 2D DWM allows for more accurate plane-wave propagation simulations than the 1D counterpart. However,

the impedance mapping of the 2D DWM is based on the same 1D area function data. The effects of the curve in the vocal tract, the addition of the nasal tract, and 3D asymmetrical cross-sections are not considered in the 1D or 2D models.

Further research has taken place which has attempted to expand on the underlying theory of 1D and 2D waveguide modelling in order to develop a 3D acoustic simulation of the vocal tract [52]. The 3D digital waveguide is a theoretically straightforward but computationally expensive extension of the 2D waveguide method, but which an acoustic resonator's 3D properties can be modelled by simply expanding the dimensionality of the DWM. As long as the topology of the mesh remains constant throughout, any 3D acoustic resonator can be modelled. Various mesh topologies exist for 2D and 3D acoustic modelling, the most common being rectilinear (2D and 3D), triangular (2D) and tetrahedral (3D) [47] [46](fig. 2.11).

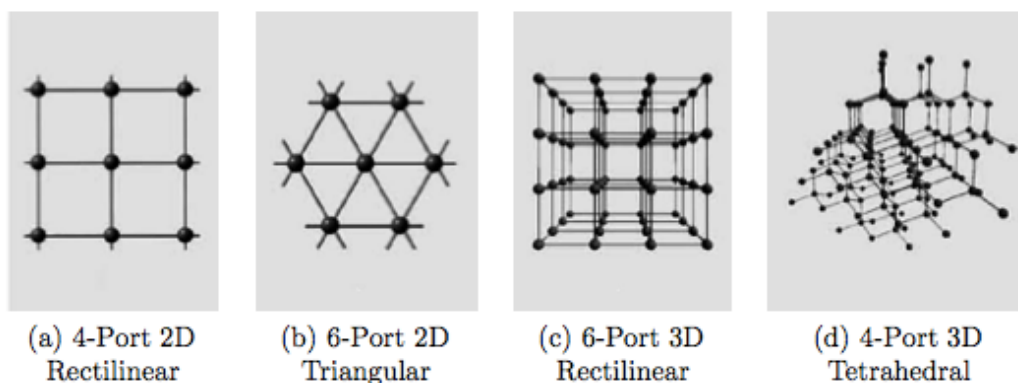


Figure 2.11 – 2D and 3D DWM topologies [from [47]]

Speed's method of 3D vocal tract acoustic modelling followed a similar principle to Mullen's 2D equivalent. This time, 3D area function data had to be obtained from MRI scans using a biomedical structure segmentation algorithm [53]. These

3D structures were converted to 3D sampling grids following similar principles to the 2D DWM construction. In this case, static sampling grids were used for each vowel shape, as opposed to the dynamic impedance-mapping techniques used with 2D implementations. This is due to the exponentially increased number of calculations per sample involved with the increase in dimensionality. A dynamic implementation of the 3D method would therefore be unfeasible given the sheer amount of computational power required.

Speed also touches on the role of the voice source within articulatory voice synthesis. In one study, both vocal tract area data and inverse-filtered voice source recordings were taken for four different subjects. The voice source recordings were then convolved with the vocal tract data of a different subject, effectively splicing the vocal folds of one subject onto the vocal tract of another. It was found that the perceived naturalness of the resulting voice synthesis was not affected when a vocal tract model was excited by a non-matching voice source – the physical correlation between the two being of less importance than ‘natural’ features of phonation such as irregular pitch, prosody, amplitude and rhythm [52].

2.6 ‘Naturalness’ in Speech Synthesis

While this research is primarily concerned with the development of a parameterised voice source model, the wider aim is to contribute towards the improved ‘naturalness’ of speech synthesis. This research follows on from two

previous projects that were concerned with speech synthesis naturalness, which was explored via real-time dynamic control of Mullen's VocalModel synthesiser [5] [30]. One of the conclusions from the earlier research was that while real-time responsive control of synthesised speech provided mostly positive subjective results, the synthesis method itself was not fully adequate. This led to the current research project, as the fixed voice source wavetable model used with the previous voice synthesiser was highlighted as an area to be improved upon. In order to set this research in context, the two previous projects will be summarised, followed by a brief review of the literature on speech synthesis 'naturalness'.

The project began as part of a short research project with the York Centre for Complex Systems Analysis (YCCSA). The primary aim of the project was to recreate a 21st-century version of the von Kempelen 'talking machine' (fig. 2.11). The von Kempelen machine was designed by Wilhelm von Kempelen in the late 18th century and described in his book 'Mechanism of Human Speech and Language' [54] [55]. This was a system composed of a set of bellows, a reed, a compressible leather tube and several valves, that were analogous to the lungs, vocal cords, vocal tract, nasal cavities and articulators in the speech system. It was supposedly possible to create speech-like sounds using this machine, and while attempts to do so using replicas of the machine have not given convincing results, its importance as a research and demonstration device has been defended [56].

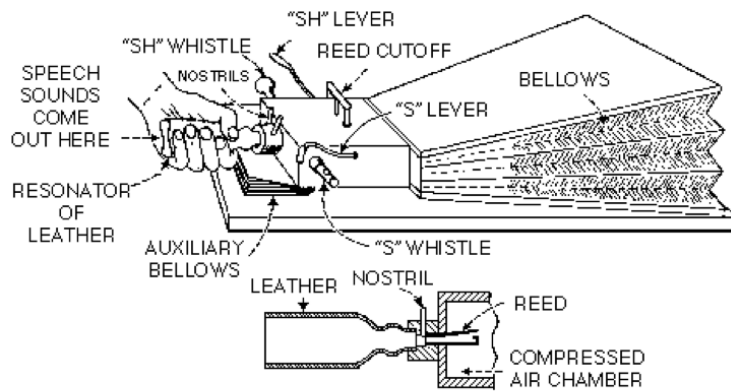


Figure 2.12 - Wolfgang von Kempelen's 'Speaking Machine' [image taken from [57]]

The approach to creating a '21st-century' von Kempelen machine was to develop a controller for the MIDI-compatible VocalModel synthesiser, based on the design of the original machine. As the voice source was already provided in the software, the bellows and reed of the original machine were accounted for, so only the leather tube and articulators were considered in the design. A MIDI controller was developed, made up of a compressible foam tube with eight force-sensing resistors (FSRs) at regular intervals along the tube. Using a series of potential divider circuits, eight separate channels of varying voltage were converted to MIDI messages based on the amount of compression. Each MIDI channel controlled one of the eight on-screen sliders within the VocalModel software, so that the compression of the foam tube was directly analogous to the 'shape' of the modeled vocal tract, and thus the vowel produced. The MIDI conversion software used was Apollo Ensemble [58], and the FSR material used was QTC, provided by David Lussey of Peratech [59].

Subjective responses to this system were that the real-time control of articulations, pitch and amplitude were somewhat speech-like, but the method of control was physically very difficult to use due to the latency introduced by the intermediate Apollo software, and the lack of sensitivity of the controller device itself. This prompted the second research project, which looked to improve the comfort and usability of the controller device as part of a deeper study in improving speech synthesis naturalness. One of the improvements made to the system was the introduction of an Arduino board, which handled the voltage input, scaling and MIDI conversion at the same time, which improved latency issues by avoiding any intermediate software. Through consultations with David Lussey, a controller device was developed with accessible FSR layers, and different QTC types were tested, so that the sensitivity and layout of the device could be changed. A more rigorous user test was involved in this study in order to gain both subjective and objective results. Users were introduced to the system over a number of tests and allowed to explore and experiment with the controller device. They were then asked to recreate vowel sounds using diagrams of the vocal tract shape, and also to identify pre-recorded vowels using the same system. Their subjective responses to the system in terms of its control method and the overall acoustic output were also collected. It was found that while it was very difficult to create recognisable vowel sounds at first, either using the visual aids or from memory, when a vowel-like sound was achieved, the inherent spontaneity of the pitch, amplitude and rhythm of the output was deemed far more 'natural' than the predictable 'robotic' voices produced by text-to-speech systems, for example. It was also noted that using an articulatory speech synthesiser allowed the user to create articulations such as plosives and

glottal stops using the control method, which many users discovered with no prior instruction. While the control method was incredibly complex, after a number of sessions experimenting with the system users were able to learn short consonant-vowel utterances such as 'ma' and 'ba'. The link between this method of learning and early speech acquisition in children was noted.

A study that was formative in the conception of these earlier projects was Newell's 'Place, Authenticity and Time: A Framework for Liveness in Synthetic Speech' [60]. This study used objective data collection as well as subjective responses to various modes of synthetic speech performance in order to determine their perceived 'liveness'. By taking an interdisciplinary approach to this research, with perspectives from the world of drama and music performance, a generic framework for what constitutes a 'natural' performance was proposed. One conclusion was that spontaneity of pitch, amplitude and pauses in synthetic speech performances gave a degree of unpredictability that was naturally associated with unprepared, conversational speech.

A commonly cited effect when discussing synthetic speech naturalness is the 'uncanny valley' phenomenon. This was first described in Mori's 'The Uncanny Valley' [61]. This phenomenon is a way of describing the change in attitude towards imitations of human features as they approach total realism. For example, a response to a simple stick drawing of a human isn't likely to illicit sympathetic responses. A well-drawn cartoon or video game character however could convey enough human-like expression to provoke sympathy, and the audience would forgive the obvious non-human characteristics (a willing

suspension of disbelief). However, the same audience might in fact find a hyper-realistic 3D rendering of a human face, or a wax figure in Madame Tussaud's, somewhat disturbing. The reason for this, Mori posits, is that the near-total realism causes the audience to give up their willing suspension of disbelief, which is replaced by a more critical viewpoint, in which minor imperfections or non-human qualities are more pronounced. Figure 2.12 displays Mori's graph of human likeness vs. affinity for the object.

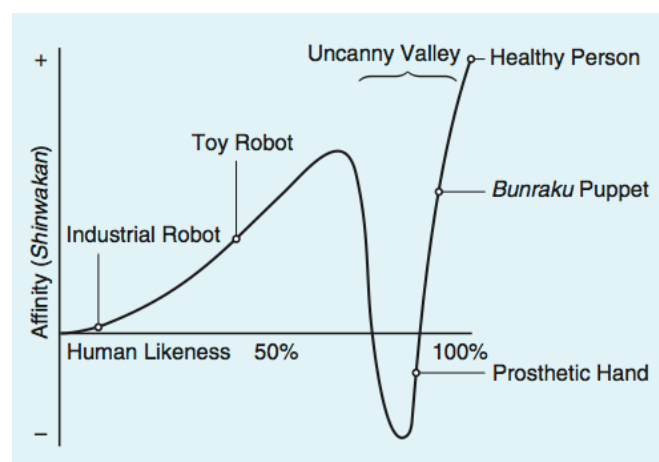


Figure 2.13 - Graph displaying the effect of human likeness on affinity or 'Shinwakan' with an object. The 'Uncanny Valley' refers to the large drop as the subject approaches human likeness [image taken from [62]]

Newell and others have drawn the link between this effect and perception of synthetic voices [60], which was also discussed in the previous research [5]. It was hypothesised that there was such a thing as a synthetic voice that was 'too perfect', with little to no spontaneity present, and that the unpredictability introduced by real-time human interaction would contribute towards more sympathetic responses.

While Mori's uncanny valley theory is concerned with audience reactions to humanoid figures, avatars, or other visual cues, it is straightforward to draw the link to speech synthesis and perception when considering Moore's Bayesian model of the uncanny valley effect [63], in which an attempt is made to quantify and model the level of 'affinity' one might have with any simulation of human qualities. His explanation leads to the conclusion that the uneasiness felt by the audience due to the uncanny valley effect is a result of the 'mis-match in sensory cues' that help the viewer/listener to identify a subject as human-like. In the context of speech synthesis, this might occur when the listener perceives a certain 'cue' for human speech present in a synthesised word phrase, alongside a non-human 'cue'. This could mean that an accurate representation of a speech signal in terms of formants and spectral content that lacks other cues such as prosody, pitch and timing would evoke a similar feeling of uneasiness to Mori's example of a life-like animatronic human figure that lacks typical human characteristics such as breathing or blinking.

3. A Parametric, Real-Time Voice Source Model

This research project began as a continuation of the author's previous studies into real-time implementations of vocal tract modelling described in the previous chapter. The motivation for these earlier projects was from a general interest in the theme of 'naturalness' in voice synthesis, as well as an interest in assistive technology and human computer interaction (HCI). It was concluded during the final stages of the previous work that while the HCI aspect of naturalness-oriented voice synthesis research was an important one, the most pressing issue is the synthesis engine itself. Developing a synthesiser that is capable of producing in real time an equivalent range of vocalisations as the human voice would ideally aid any future research into more HCI-focused areas. Whilst the general motivation for the current research remains the same as earlier projects, the scope of the project has been revised in order to focus on a key component of the voice synthesis software used in the earlier projects: the voice source.

This chapter describes the development of a parametric, real-time voice source modelling application for iOS. The motivation for the application design is given first, followed by the application specifications. The design and implementation stages are documented, and the results from the system testing stage are given. As with many software development projects, the progression from the initial concept to design, implementation and testing is not a linear one. The following

sub-sections are therefore not necessarily presented in chronological order (for example, results from system testing informed certain changes in the design of the app).

3.1 Motivation for Design

As stated in Chapter 1, the motivation for this project is twofold. Firstly, the application is intended to be used as a research tool in voice synthesis projects. This means the final application should be capable of producing a *versatile* and *expressive* voice source model that can be incorporated into other voice synthesis systems. In terms of versatility, the application should provide sufficient parameters to the user so that an appropriately wide range of voice source waveforms can be modeled. The term ‘expressive’ here means an application that is capable of a wide range of pitch, amplitude, voice types and so on that can be modified in real time, in a similar manner to ‘expression’ in musical performance.

The secondary motivation for the design of this application is the implications for its use within assistive technology software. This informed the decision to focus on technology for portable devices. Technologies such as Voice Output Communication Aids (VOCAs) [64] have been used by the disabled community for many years as a means of communication. Since the advent of touchscreen technology, users of limited mobility have made use of the large range of options

3. A Parametric, Real-Time Voice Source Model

made available by not being limited to hard-coded buttons [3]. Touchscreens allow any part of the screen to be used as a controller, from onscreen switches and sliders to more complex gestures such as pinching and multitouch swiping [65]. Devices such as Apple's iPad, often equipped with quad-core processors, have surpassed the performance of the average desktop computer from a decade ago, and so make the ideal platform for a touchscreen-based communication aid. Modern VOCAs that make use of concatenative synthesis (a speech synthesis method where appropriate diphones are selected from a bank of pre-recorded samples and 'stitched' together to create entire phrases) are already capable of producing highly realistic speech output. This is, however, at the cost of limited vocabularies and lack of expressive options in terms of pitch, amplitude, speed of talking, and voice source quality. It is proposed that a more flexible speech synthesis system would incorporate some form of voice source modelling, and so the application has been designed with this in mind. It should be noted that the application is by no means intended as a completed speech synthesis tool, but is intended to be incorporated into such projects.

3.2 Specifications

The specifications for the design of the application are given below:

1. In order to allow expressive control of voice source features and voice types, the application will provide appropriate parameters for the user to control.
2. The application will run as close to real time as possible. This means any changes made to fundamental frequency, amplitude or any LF-model parameters will be processed and heard instantly. This is essential for the application both as a contribution to HCI and voice synthesis research, as well as any potential use as an assistive technology enhancement.
3. Whilst it is not essential that the final application run on portable devices at this stage, it is highly desirable that the application is at least designed to be compatible with such devices. This is so that any future iterations of the software can be easily incorporated into portable devices such as tablets and phones, which are widely used as assistive technology devices, as well as research and demonstration tools [66].
4. The application will be compatible with existing voice synthesis methods. This means specifically that the voice source model will be designed to run at the same sample rate (the rate at which audio samples are calculated and played back) as existing vocal tract modelling applications,

and the two models can be integrated without compromising the quality of output.

3.3 Design

This section provides a detailed account of the design stages of the application, with focus given to some of the more crucial design choices.

3.3.1 Choice of Parameters

The choice of voice source features to parameterise was a crucial decision in the design of the software. Through many iterations of the design, various parameters were made available to the user, with each parameter's effectiveness and suitability noted. This version of the LF-model makes use of the four timing parameters detailed in section 2.4.2. To reiterate, these are notated as t_p , t_e , t_a , and t_c . Figure 3.1 illustrates how these timing parameters relate to the LF-model waveform:

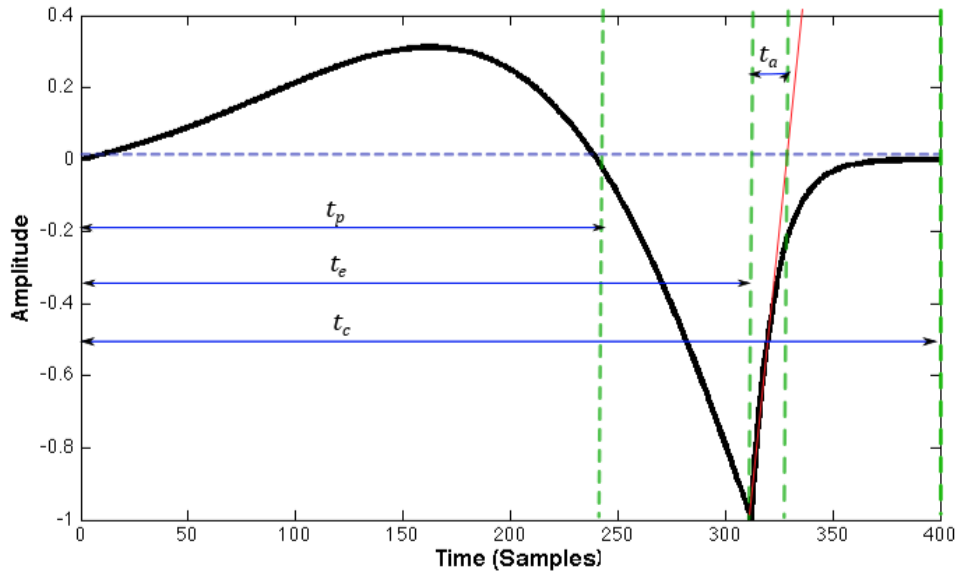


Figure 3.1 - Annotated 'Typical' LF-waveform

T_p is the moment of zero crossing during the differentiated glottal flow waveform. This relates to the moment of maximum flow during the original waveform. T_e refers to the negative peak. The section of the waveform from $t = 0$ (where t is the current sample) to t_p is the exponentially growing sinusoidal component. T_a defines the gradient of the exponential return phase. It is notated as the distance between the negative peak and the moment of zero crossing when the return slope is differentiated. T_c is the total length of the LF-waveform portion of the pitch period (note that t_c is usually equal to the length of one pitch cycle). The values for these parameters are given as percentages of the total pitch period.

MATLAB software was used throughout the prototype stages of the design in order to test various algorithms and methods without the added complication of working with Core Audio [67] in iOS. A basic fixed-parameter LF-model was

3. A Parametric, Real-Time Voice Source Model

implemented in MATLAB based on that found in the VOICEBOX toolkit [68]. Each of the four LF-parameters was then altered within an appropriate range in order to assess their acoustic function. It was found that altering each parameter independently provided notable and predictable results corresponding to their function described in the literature.

Parameter t_e , or the moment of the negative peak, can be varied over a range of around 20% of the pitch cycle before losing the typical LF-model waveform shape. In figure 3.2, it can be seen that at the extremes ($t_e = 0.9$ and $t_e = 0.7$), discrepancies in the waveform occur where the value of one parameter approaches the value of another. This provides a rough estimate of the appropriate ranges to be made available to the user on the iOS interface.

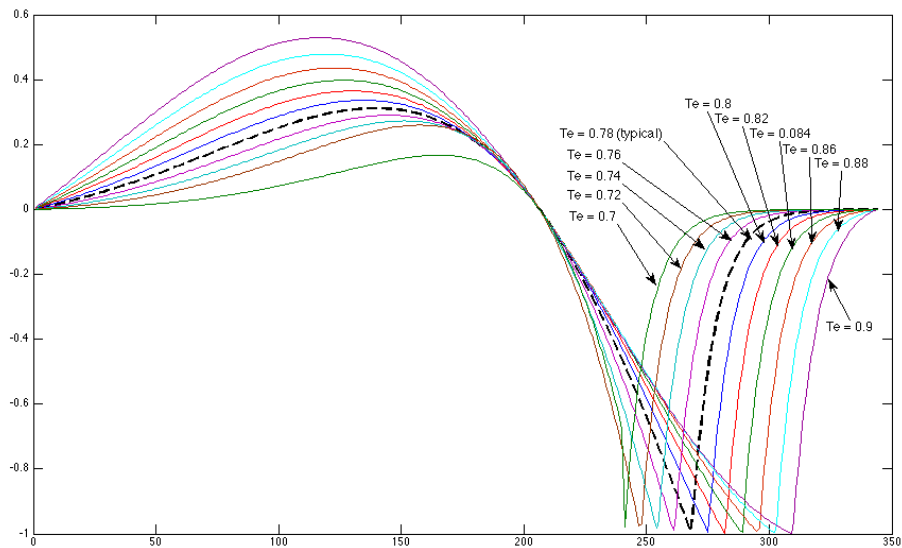


Figure 3.2 – ‘Typical’ LF-waveform with varying t_e value from 0.7 to 0.9 (x-axis represents time in samples, with relative amplitude on the y-axis)

3. A Parametric, Real-Time Voice Source Model

The moment of zero crossing occurs at point t_p . As with t_e , varying this parameter has an extreme effect on the opening phase as well as the closing phase, in terms of amplitude and steepness (fig. 3.3). Discontinuities occur at the upper ranges (around $t_p = 0.66$), while the sinusoid component of the open phase is more pronounced for lower values of t_p . The return phase remains constant for all values.

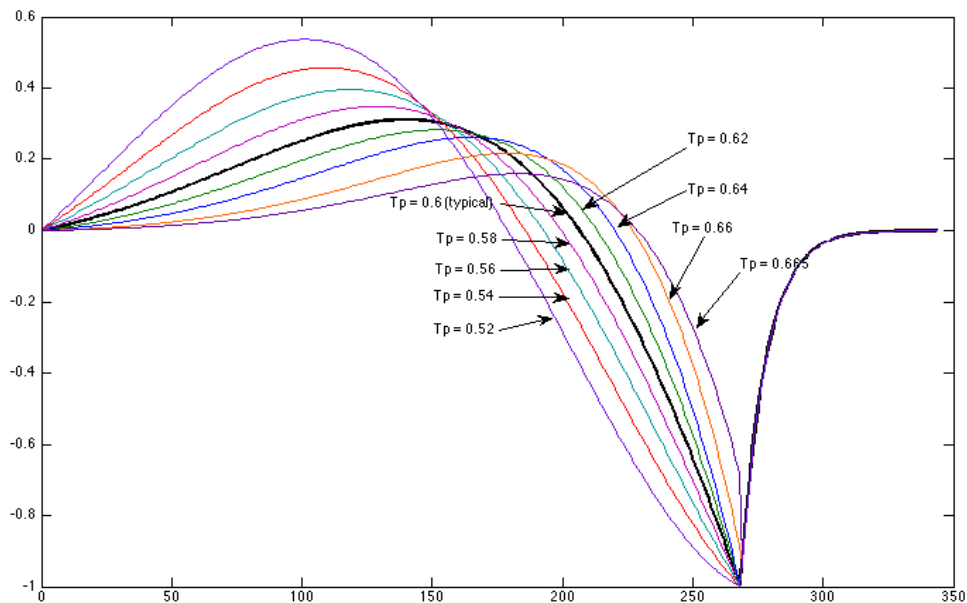


Figure 3.3 – ‘Typical’ LF-waveform with varying t_p value from 0.52 to 0.665 (x-axis represents time in samples, with relative amplitude on the y-axis)

T_a , which roughly describes the length and gradient of the return phase, has the most pronounced effect on the spectrum of the waveform. This is due to the negative peak being the main excitation of the voice source [7]. A steep gradient in the return phase will produce more high frequency harmonics than a gradual return. It was also found that t_a could be set to a large range of frequencies without causing any discrepancies in the typical LF-waveform (fig. 3.4).

3. A Parametric, Real-Time Voice Source Model

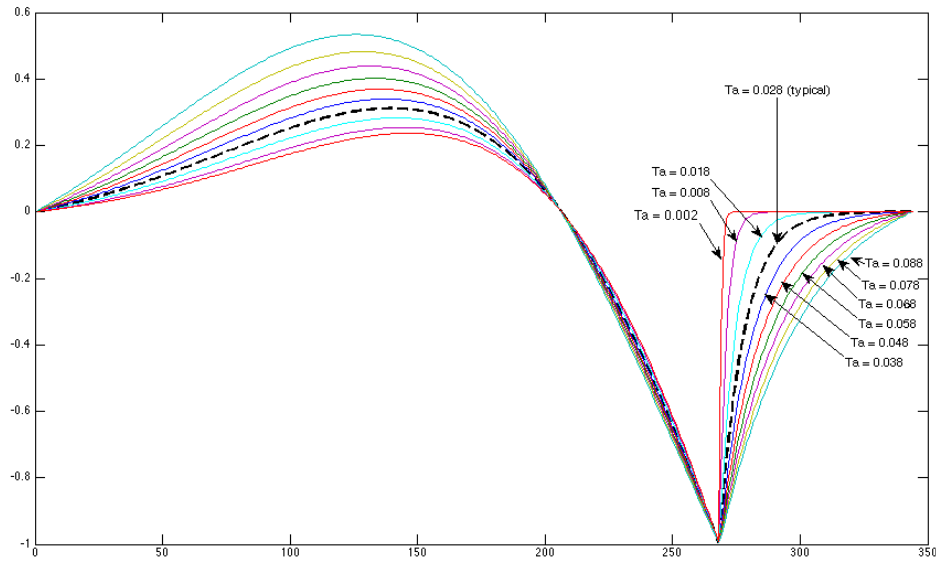


Figure 3.4 – ‘Typical’ LF-waveform with varying t_a value from 0.002 to 0.038 (x-axis represents time in samples, with relative amplitude on the y-axis)

As t_e , t_p and t_a are all calculated independently of t_c , there was no audible effect from modulating this parameter. If t_c is set to a value approaching t_e , the equation does not hold true, which in the worst case scenario results in digital clipping in the audio output of the software. For voice types where the open phase is less than the length of the pitch cycle (i.e where $t_c < 1$), all four parameters reflect this. It is possible to derive values for t_e , t_p and t_a from t_c rather than as a percentage of the total pitch cycle. This would, however produce timing parameter values that are not related to any recorded function of the voice source from the literature. An example of this would be a model of the vocal fry voice type, in which t_c is often set to less than half of the overall pitch cycle. The waveform in this case does not simply represent a contracted version of a typical

LF-model cycle, as the gradient of the return phase is much steeper than for a modal or breathy voice.

In [14], Childers discusses the effects of voice source parameters in terms of vocal 'tension', which relates to the perceived effort of the speaker. It was found that varying t_e , t_p and t_a individually could produce shifts in perceived vocal tension. However, a closer correlation to perception of vocal effort was found in the speed quotient (SQ). The speed quotient is defined by a cross-coupled relationship between the four LF-parameters, and therefore reflects a more natural mode of modification of the vocal source, as opposed to altering each parameter independently of the others.

3.3.2 Wavetable Synthesis

Initially, playback of the LF-waveform as a periodic signal was achieved via wavetable synthesis. This is a method of digital synthesis in which a precomputed waveform is stored as an array of discrete values. A lookup address is calculated at each sample step depending on the pitch of playback and sampling frequency. Often, this address will be a non-integer, and so some form of interpolation is required to define a value for the current output sample.

Wavetable synthesis provided a stable playback method for pitch and amplitude changes in the LF-model signal. However, when modifications such as timing parameter changes were introduced, digital distortion occurred. This was due to

3. A Parametric, Real-Time Voice Source Model

the fact that to effect a change in the LF-model waveform, it had to be recalculated from the first sample, thereby creating discontinuities. A less computationally efficient alternative to wavetable synthesis was tested. This involved performing the LF-model equations at every sample step. It was found that this performed well in MATLAB and iOS, even when rapid changes were made to timing parameters and pitch, with no detectable discontinuities. Further additions to the software may affect the computational load, and so a more optimised wavetable synthesis method such as those described in [69] could be employed.

The advantage of using a fixed-size wavetable to represent the LF-model waveform was that it was relatively straightforward to provide visual feedback for the user in iOS. Using the Core Plot library [70] a simple on-screen graph was produced that represented the values of each sample in the wavetable (fig. 3.5).

3. A Parametric, Real-Time Voice Source Model

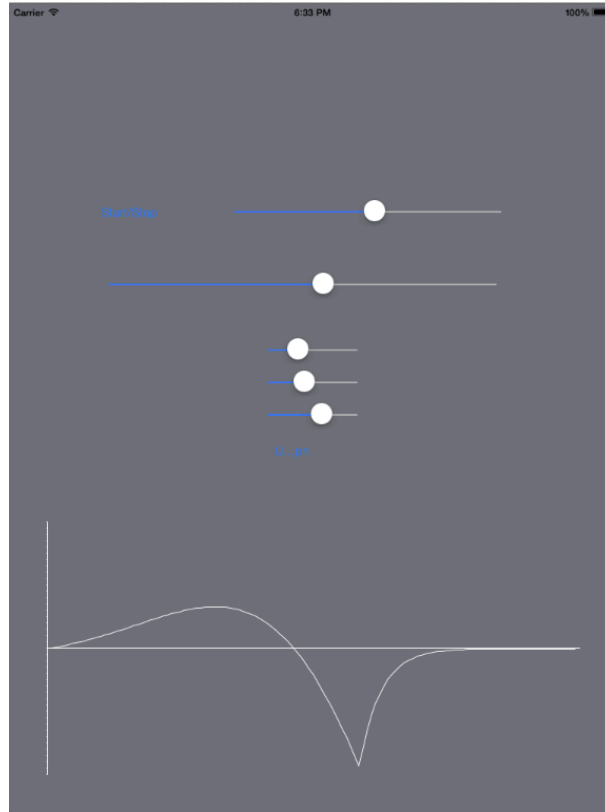


Figure 3.5 – Screenshot from first implementation of the ‘LFGGen’ application with CorePlot waveform display

3.3.3 Voice Types

An important feature of a voice source modelling synthesiser is the ability to synthesise different voice types. The four voice types that can be modelled with an LF waveform are modal, breathy, vocal fry and falsetto [14]. Other voice types exist, such as whisper, but these are aperiodic signals that consist largely of white noise. LF-parameter values for these four voice types are found in [24] as well as the source code for Mullen’s VocalModel synthesiser [71]. The VocalModel source code also provides a ‘typical’ voice type. It is assumed that this represents the archetypal waveform commonly associated with the LF-model.

3. A Parametric, Real-Time Voice Source Model

These four voice types (as well as ‘typical’ voice) were made available to the user. In MATLAB, the LF-parameters found in both [24] and [30] were provided so that a preferential set of voice types could be used for the final application. During informal subjective listening, the LF-parameter values from [24] provided the most noticeable difference between each voice type, with the modal and vocal fry voices eliciting more positive subjective responses than their counterparts from Mullen’s VocalModel code.

Table 3.1 – Five voice types used for ‘LFGGen’ app and their corresponding timing parameter values

	T_e (%)	T_p (%)	T_a (%)	T_c (%)
Modal	57.5	45.7	0.9	100
Breathy	75.6	52.9	8	100
Vocal Fry	25.1	19	0.8	100
Falsetto	77	57	13	100
‘Typical’	78	60	2.8	100

Appropriate pitch ranges were considered for each voice type. In general, breathy and modal voices are both used for typical speech and singing pitch ranges, while vocal fry is associated with low speech and singing pitches, and falsetto for high singing pitches. An f_0 -dependant voice type switching method would mimic this feature of the human voice, and prevent irregularities such as a vocal fry waveform appearing at high pitches.

Breathy and falsetto voice types also contain a high frequency noise component [14]. A random number generator was implemented in the main processing loop

3. A Parametric, Real-Time Voice Source Model

to provide a white noise signal with values ranging between -1 and +1. The turbulent noise component of a breathy or falsetto voice features a low-frequency cutoff at around 2 kHz. Due to low-frequency masking effects when added to the LF-model signal, it is not necessary to filter the noise component first, and so non-filtered white noise was sufficient. The relative amplitude of the white noise signal is quoted as around 5% of the source signal [14]. This value was tested in MATLAB and provided a sufficient amount of 'breathiness' to the voice source. Also significant to the turbulent noise component is the duration and start time of the noise signal. Breathiness is caused by an incomplete closure of the glottis during the 'closed phase' of the signal. This produces a turbulent airflow from the narrow constriction, causing noise. This usually occurs with a start time at around 75% of the pitch period and for a duration of 50% [14], meaning that the 'breathy' phase of the cycle actually overlaps two adjoining pitch periods. Three cycles of the breathy voice waveform with added noise is displayed in figure 3.6 below.

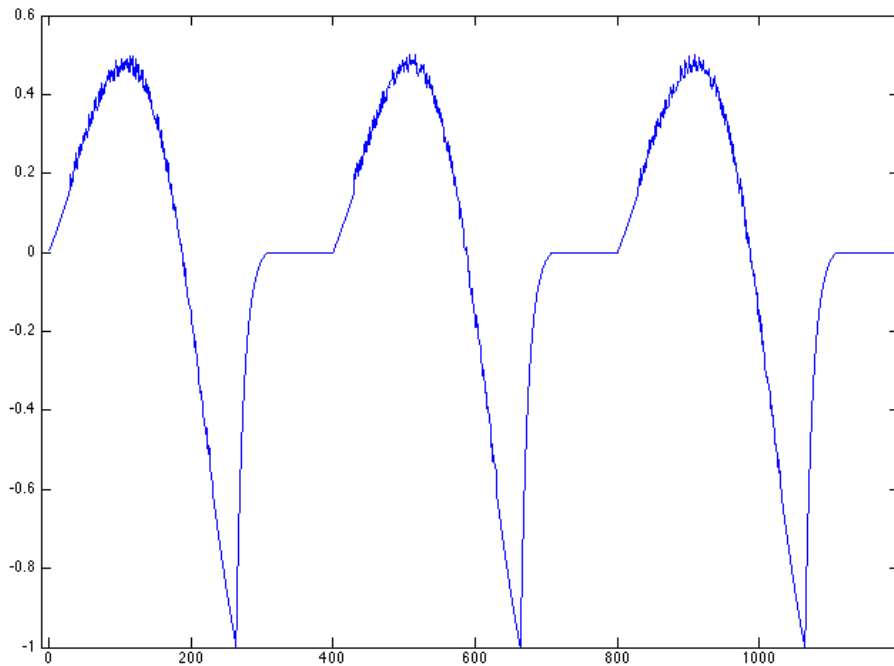


Figure 3.6 – Three cycles of breathy voice type waveform with turbulent noise quotient visible

3.3.4 iOS Interface

In keeping with specification 3 from section 3.2, an iOS implementation of the voice source model was developed alongside the MATLAB prototypes. This meant developing a suitable interface alongside the voice source model. Several approaches were considered, including a multitouch, gesture-based interface. Early in the development stages, it was concluded that such an interface, whilst providing an added layer of cross-coupling between the many variables at play

during speech production, would constitute a lengthy development process that would detract from the primary task.

3.3.5 Final Design

The final design for the application incorporates an LF-waveform generator, a noise generator, pitch and amplitude modulation, voice type selection and interpolation and a method of providing automatic f_0 input. A simple iOS interface provides the user with on-screen buttons and sliders for discrete and continuous parameters. In MATLAB, the user is provided with a larger range of options such as custom voice types, noise source amount and length, and audio file playback and saving.

Figure 3.7 presents the functionality of the application in a 'black box' format.

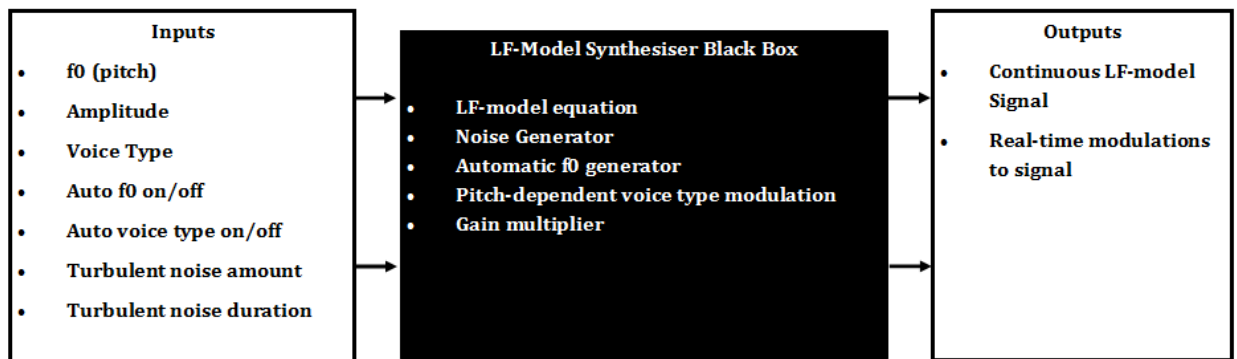


Figure 3.7 – Black box diagram for 'LFGGen' application

The app design can be summarised in pseudocode as follows:

```
Set initial values:  
  Set voice type to TYPICAL  
  Set f0 to 110 Hz  
  Set amplitude to 50%  
  Set auto f0 to OFF
```

3. A Parametric, Real-Time Voice Source Model

```
    Set auto voice type to OFF
End

User input handling:
    Set f0, amp, voice type to user selected values
End

Voice types:
    Set VOCAL FRY max and min f0 values
    Set FALSETTO max and min f0 values

    SWITCH voice types
        Case 0
            Set to TYPICAL voice
        Case 1
            Set to MODAL voice
        Case 2
            Set to VOCAL FRY voice
        Case 3
            Set to BREATHY voice
        Case 4
            Set to FALSETTO voice
    END

    IF auto voice type is set to ON:
        IF f0 is less than min VOCAL FRY f0
            Set voice type to VOCAL FRY
        END

    IF f0 is more than min VOCAL FRY f0 and less than max    VOCAL FRY f0
        INTERPOLATE between VOCAL FRY and MODAL/BREATHY/TYPICAL voice types
    END

    IF f0 is more than max VOCAL FRY f0 and less than min FALSETTO f0
        Set voice type to currently selected (MODAL/BREATHY/TYPICAL)
    END

    IF f0 is more than min FALSETTO f0 and less than max FALSETTO f0
        INTERPOLATE between MODAL/BREATHY/TYPICAL and FALSETTO voice type
    END

        IF f0 is more than max FALSETTO f0
            Set voice type to FALSETTO
        END
    END

LF-waveform calculation:
    CALCULATE LF-equation coefficients

    CALCULATE length of one full pitch period in number of samples
End

Main Processing Loop:
    IF auto f0 is set to OFF
        Perform LF-equation calculations
        Assign value of current sample to output variable

    IF BREATHY or FALSETTO voice types are selected (i.e if white noise
    is needed)
        CALCULATE white noise sample from random number function
```

3. A Parametric, Real-Time Voice Source Model

```
CALCULATE portion of waveform to add noise to. If start time +
duration is larger than one pitch period, CALCULATE the remainder to
add to the beginning of next pitch period

MULTIPLY noise sample by noise amplitude

ADD noise sample to current output variable
  END

  IF auto f0 is set to ON
    IF current sample is at beginning of new pitch period
      SET length of pitch period in samples
    END

    Perform LF-equation calculations
    Assign value of current sample to output variable

  IF BREATHY or FALSETTO voice types are selected (i.e if white noise
  is needed)
  CALCULATE white noise sample from random number function

  CALCULATE portion of waveform to add noise to. If start time +
  duration is larger than one pitch period, CALCULATE the remainder to
  add to the beginning of next pitch period

  MULTIPLY noise sample by noise amplitude

  ADD noise sample to current output variable
    END
  END

  SET output sample to output variable value
End
```

3.4 Implementation

The implementation stage comprised of two parallel tasks. First, each feature of the application was implemented and refined in MATLAB. Once tested and approved, this feature was incorporated into the iOS application. This allowed for the functionality of each parameter to be studied closely in isolation from other factors before being implemented in the final app. The MATLAB simulations are not in real time, and take the form of a step-by-step process in which a waveform of a defined length is calculated and converted to a PCM audio file. The real-time processing in iOS is achieved via Core Audio. This allows for a

continuous waveform that can be modified on the fly. Despite the MATLAB and iOS development taking place in parallel, they are presented here separately for the sake of clarity.

3.4.1 Implementation in MATLAB

The MATLAB implementation provides the user with several options in the form of on-screen text prompts. The first section of code is a switch-case statement for defining the timing parameters for each voice type:

```
% Voice type selection:

voicetype = input('Enter a voice type (number between 1-8): ');

switch voicetype

% for TYPICAL typical voice type
    case 1
        f0 = input('Enter a frequency between 94-287: ');
        period = 1/f0;
        tc = 1.000*period;
        te = 0.780*period;
        tp = 0.600*period;
        ta = 0.028*period;
% for MODAL voice type
    case 2
        f0 = input('Enter a frequency between 94-287: ');
        period = 1/f0;
        tc = 0.582*period;
        te = 0.554*period;
        tp = 0.413*period;
        ta = 0.004*period;

...

% for CUSTOM voice type
    case 9
        f0 = input('Enter a suitable frequency: ');
```

3. A Parametric, Real-Time Voice Source Model

```
period = 1/f0;
TC = input('Enter a value for Tc (0-1): ');
TE = input('Enter a value for Te (0-1): ');
TP = input('Enter a value for Tp (0-1): ');
TA = input('Enter a value for Ta (0-1): ');
tc = TC*period;
te = TE*tc;
tp = TP*tp;
ta = TA*ta;
end
```

The timing parameters are given in terms of percentage of the overall pitch period. The user is given nine possible options. These are typical, modal, vocal fry and breathy voice, taken from the VocalModel source code, modal, vocal fry, breathy and falsetto voice types obtained from [24], and a custom voice type for user defined timing values. The user is also presented with suggested values for fundamental frequency based on the selected voice type. These suggested frequencies are: 94–287 Hz for typical, modal and breathy, 287–440 Hz for falsetto, and 24–52 Hz for vocal fry. These pitch range values were obtained from various sources, including [14], [72] and [17].

Next, the user is prompted for a ‘vocal tension’ value. This is a rough calculation that raises or lowers the SQ by a factor of +/- 30%. This is achieved by adding/subtracting a percentage from the previously defined timing parameters. This allows the user to select, for example, a breathy voice type, but to modify the SQ to illicit a more or less ‘tense’ voice than the standard breathy voice values allow:

```
vocalTension = input('Enter a vocal tension value (+/- 0.3): ');
te = te + te*vocalTension;
tp = tp + tp*vocalTension;
ta = ta - ta*vocalTension;

SQ = tp/(te+ta-tp); % Speed Quotient
```

3. A Parametric, Real-Time Voice Source Model

The bulk of the LF-waveform calculation is performed next. First, oversampling is performed in order to avoid waveform discontinuities produced by a lower sample rate, followed by the definition of the remaining coefficients. Next, values for ε , α and E_0 are calculated iteratively. In order to satisfy the area balance condition,

$$\int_0^{T_0} LF(t) = 0$$

[3.1]

ε , the exponential time constant of the return phase; α , the exponential growth factor of the sinusoid portion (opening phase) and E_0 , which is the maximum positive flow, are calculated and recalculated until the positive and negative areas of the waveform are equal:

```
% Oversampling
period = period/overSample;
tc = tc/overSample;
te = te/overSample;
tp = tp/overSample;
ta = ta/overSample;

% LF coefficient calculation
tn = te - tp;
tb = tc - te;

wg = pi/tp;
Eo = Ee;

areaSum=1.0;
peakChange=0.001;
optimumArea=1e-14;
epsilonDiff=10000.0;
epsilonOptimumDiff=0.1;

% solve iteratively for epsilon

epsilonTemp = 1/ta;

while abs(epsilonDiff)>epsilonOptimumDiff
```

3. A Parametric, Real-Time Voice Source Model

```
epsilon = (1/ta)*(1-exp(-epsilonTemp*tb));
epsilonDiff = epsilon - epsilonTemp;
epsilonTemp = epsilon;

if epsilonDiff<0
    epsilonTemp = epsilonTemp + (abs(epsilonDiff)/100);
end
if epsilonDiff>0
    epsilonTemp = epsilonTemp - (abs(epsilonDiff)/100);
end

end

% iterate through area balance to get Eo and alpha to give A1 + A2 =
0

while (areaSum > optimumArea)
    alpha = real(( log(-Ee/(Eo*sin(wg*te))))/te);

    Area1 = ( Eo*exp(alpha*te)/(sqrt(alpha*alpha+wg*wg)))...
        * (sin(wg*te-atan(wg/alpha)))...
        + (Eo*wg/(alpha*alpha+wg*wg));

    Area2 = ( -(Ee)/(epsilon*epsilon*(ta)))...
        * (1 - exp(-epsilon*tb*(1+epsilon*tb)));

    areaSum = Area1 + Area2;

    if areaSum>0.0
        Eo = Eo - 1e5*areaSum;
    elseif areaSum<0.0
        Eo = Eo + 1e5*areaSum;
    end

end

end
```

The length of one full pitch period in samples is calculated, followed by the turbulent noise coefficients. `noiseDuration` is the length of the turbulent noise portion, determined as a percentage of the pitch period. `noiseStart` is the point at which the noise begins:

```
% Noise parameters
noiseAmt = input('Enter a turbulent noise amount (0-1): ');
if noiseAmt > 0.0;
    noiseDuration = input('Enter a turbulent noise duration: ');
    noiseStart = input('Enter a turbulent noise start position: ');
    noiseDuration = noiseDuration*dataLength;
    noiseStart = noiseStart*dataLength;
```

3. A Parametric, Real-Time Voice Source Model

The user is then prompted to define a length in seconds for audio output. This is converted to a value in samples depending on sampling frequency, and an output waveform array is created:

```
% playback section
durationSeconds = input('Enter a duration (s): ');
durationSamples = durationSeconds*Fs;
waveform = zeros(1,durationSamples);
```

The main processing loop incorporates a sample-by-sample calculation of the LF-waveform based on the LF-model equation

$$LF(t) = E_0 e^{\alpha t} \sin(\omega_g t), 0 \leq t \leq t_e \quad [3.2]$$

$$LF(t) = -\frac{E_e}{\epsilon t_a} [e^{-\epsilon(t-t_e)} - e^{\epsilon(t_c-t_e)}], t_e \leq t \leq t_c \leq T_0 \quad [3.3]$$

```
for i = 1:durationSamples
    t = j*period/dataLength;

    % This is where the LF waveform is calculated
    if t<te
        LFSample = Eo*(exp(alpha*t)) * sin(wg*t);
    end
    if t>=te
        LFSample = -((Ee)/(epsilon*ta))*(exp(-epsilon*(t-te))...
            - exp(-epsilon*(tc-te)));
    end
    if t>tc
        LFSample = 0.0;
    end
end
```

This is followed by the turbulent noise samples being created using a random number generator. These are added to the current sample depending on the `noiseAmt` variable. If no noise is selected then the waveform stays the same. If a noise amount has been defined, then a sample of random amplitude (scaled by the `noiseAmt` variable) is added to the current sample. Only the portions of the waveform defined by `noiseDuration`, `noiseStart` and `remainder` are assigned

3. A Parametric, Real-Time Voice Source Model

a noise sample:

```
if noiseAmt == 0.0;
    waveform(i) = LFSample;
end

% Add noise
if noiseAmt > 0.0;
    noiseSample = rand(1,1)*noiseAmt;

% This adds white noise to LF-waveform at noiseStart for
noiseDuration.
% need to cycle around to the start of pitch period if noiseStart +
% noiseDuration is bigger than length of period (dataLength):

    if noiseStart + noiseDuration < dataLength;
        if j >= noiseStart && j <=noiseStart+noiseDuration
            waveform(i) = LFSample + noiseSample;
        end
        if j < noiseStart || j > noiseStart+noiseDuration
            waveform(i) = LFSample;
        end
    end
    if noiseStart + noiseDuration > dataLength;
        remainder = (noiseStart+noiseDuration)-dataLength;
        if j <= remainder || j >= noiseStart
            waveform(i) = LFSample + noiseSample;
        end
        if j > remainder && j < noiseStart
            waveform(i) = LFSample;
        end
    end
end
end
```

Finally, a counter j is incremented and reverted to 0 if its value is larger than the pitch period size in samples. This allows multiple pitch periods to be calculated in the same output waveform. The user is then provided with options for playback and saving of the waveform, as well as a plot for visual reference:

```
j=j+1;

if j > dataLength
    j = j - dataLength;
end

end

plot(waveform);

p = input('press 1 to play sound: ');
```

3. A Parametric, Real-Time Voice Source Model

```
if p == 1
    sound(waveform,Fs)
end

s = input('press 1 to save sound: ');

if s == 1
    filename = input('Enter .wav filename: ');
    audiowrite(filename, waveform, Fs);
end

q = input('press 1 to start or 0 to quit: ');
end
```

A separate MATLAB script was created to prototype an automatic f_0 trajectory obtained from a recorded phrase. Praat software was used to obtain the f_0 data from a short utterance with a rising and falling f_0 , lasting one second. This data was then loaded into MATLAB. The sampling frequency of the f_0 acquisition was 100 Hz, so the data had to be interpolated to the MATLAB sampling frequency of 44.1 kHz. An output array equivalent to the size in samples of the interpolated f_0 data was created. The rest of the code is similar to the previous, with the exception that the processing loop encompasses the entire process, allowing the waveform to be recalculated at each pitch cycle depending on the current f_0 value:

```
f0_file = load('naturalf02.mat');
f0 = f0_file.F0_Hz;
lengthf0 = size(f0,1);
f0sampleRate = 100; % sample rate for pitch data in PRAAT
Fs = 44100;
overSample = 1000;
f0changeRate = Fs/f0sampleRate;
newf0 = interp(f0,f0changeRate);
lengthNewf0 = size(newf0, 1);
output = zeros(1,lengthNewf0);

q = 1;
k = 1;

while q == 1

    voicetype = input('Enter a number between 1-5: ');
```

```
for i = 1:lengthNewf0
    period = 1/newf0(i);
```

3.4.2 Implementation in iOS

Implementation of the application in iOS involved adding to a pre-written Core Audio app template by Dimitrios Zantalis. This template provides the necessary Core Audio configurations for a real-time audio app. The RemoteIO audio unit is configured for simple playback with no input sound source. The Audio Stream Basic Description (ASBD) defines the format and size of the audio data, including number of channels and bytes per channel. More information on core audio for iOS is available in [67].

The Model-View-Controller (MVC) design paradigm [73] dictates that core processes (the ‘models’) are calculated in separate functions from user interface handling (the ‘views’), with communication between the two handled by ‘controller’ functions. This implementation follows the MVC paradigm, with user input handling methods (the ‘View’) provided in the `ViewController` function, whilst the main audio processing (the ‘Model’) occurs in the `AudioEngine` function. In order to pass variables and coefficients between these methods and functions, a structure of type `EffectState` (the ‘Controller’) is declared in the `AudioEngine` header file (figure 3.8). This structure allows basic operations such as GUI handling and coefficient calculation to happen in separate threads to the

high-priority DSP thread, passing variables via EffectState only when necessary.

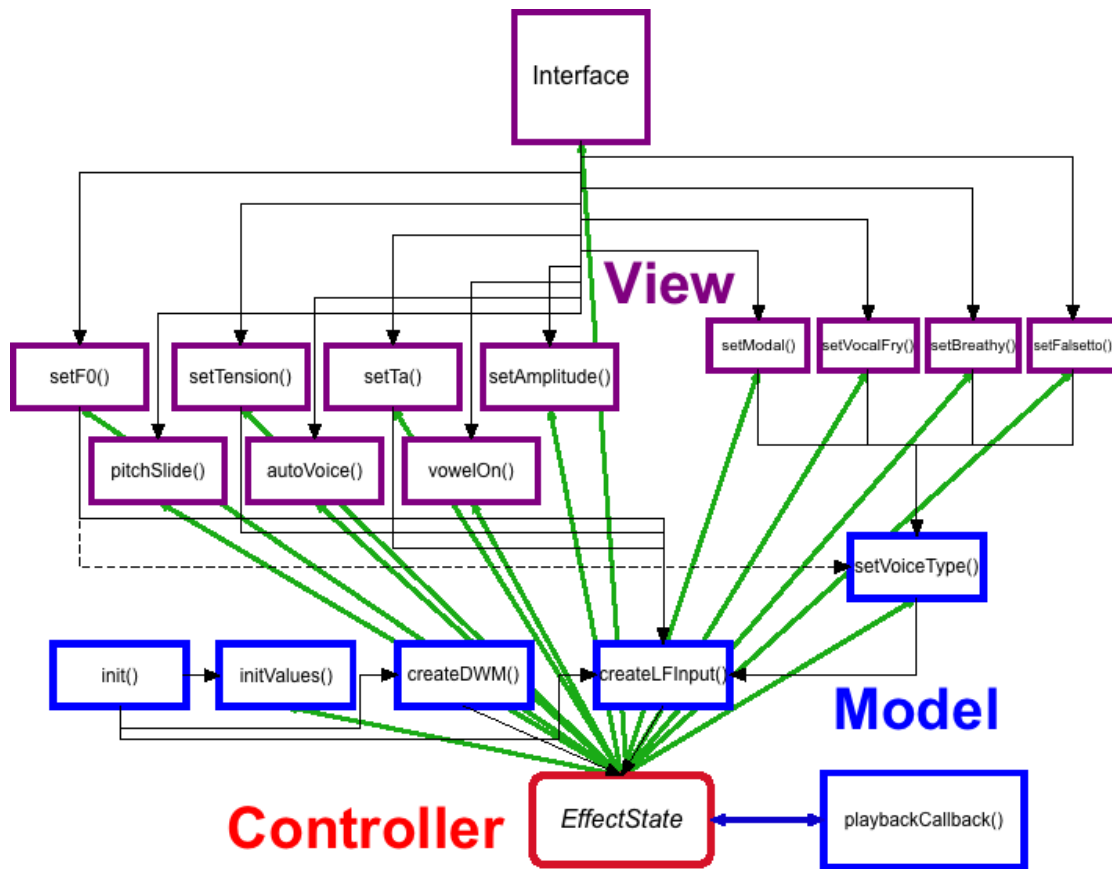


Figure 3.8 – Software diagram for LFGGen iOS application with Model-View-Controller layout highlighted.

Aside from the Core Audio implementation, the algorithm for the iOS version follows a very similar structure to the MATLAB version. First, the parameters available to the user are set based on user input. The main deviation from the MATLAB implementation is the inclusion of a set of initial values. This allows instant playback as soon as the application is opened, rather than waiting for the user to set every parameter value before any sound is processed:

3. A Parametric, Real-Time Voice Source Model

```
-(void)initValues{
    effectState._f0 = 110.0;
    effectState._k = 0;

    //TYPICAL voice type
    effectState._tcVal = 1.0;
    effectState._teVal = 0.575;
    effectState._tpVal = 0.457;
    effectState._taVal = 0.009;

    effectState._amp = 0.5;
    effectState._pitchSlide = FALSE;
    effectState._autoVoice = TRUE;
}
```

This sets the initial voice type to ‘typical’, with an automatic f0 input and amplitude set to 50%.

The following methods handle the user input from the interface class. If a variable is used in another method, this method is called when the variable is changed. For example if a timing parameter is altered, the main LF-waveform method is called so that a new waveform can be calculated, for example:

```
-(void)setTa:(Float32)taValue{
    effectState._taVal=taValue;
    [self createLFInput];
}
```

The setVoiceType method defines the timing parameters and turbulent noise quotients for the current voice type. The values are passed to the createLFInput method, which calculates the LF-waveform. First, the pitch thresholds for vocal fry and falsetto voice types are defined. These define the pitch ranges over which the voice type will automatically modulate from low-f0 voice types (vocal fry) to mid-f0 voice types (breathy, modal and ‘typical’) to high-f0 voice types (falsetto). These values are defined in terms of the amount of samples in one full pitch period:

3. A Parametric, Real-Time Voice Source Model

```
int vocalFryMax, vocalFryMin, falsettoMax, falsettoMin;
vocalFryMax = 848; // 52 Hz
vocalFryMin = 469; // 94 Hz
falsettoMax = 213; // 207 Hz
falsettoMin = 153; // 288 Hz
```

The timing parameter and turbulent noise quotient values are given the same values as in the MATLAB script, with an extra statement to call the `createLFInput` method in order to update the waveform:

```
switch (effectState._voiceType) {
    case 0: // typical voice type
        effectState._tcVal = 1.0;
        effectState._teVal = 0.780;
        effectState._tpVal = 0.600;
        effectState._taVal = 0.028;
        effectState._noiseOn = FALSE;
        effectState._vocalTension = 0.0;
        [self createLFInput];
        break;
```

The rest of this method handles the interpolation between voice types. An 'if' statement checks if the 'auto voice type' option has been selected, and if true, performs the interpolation:

```
if (effectState._autoVoice == TRUE){
    // Set to Vocal Fry if dataLength is larger than vocalFryMax
    (i.e. if  $f_0 < 52$  Hz)
    if (effectState._dataLength > vocalFryMax) {
        effectState._tcVal = 1.0;
        effectState._teVal = 0.251;
        effectState._tpVal = 0.19;
        effectState._taVal = 0.008;
        effectState._noiseOn = FALSE;
        effectState._vocalTension = 0.0;
        [self createLFInput];
    }

    // Interpolate from Vocal Fry to Breathy/Modal if dataLength
    is between vocalFryMax & vocalFryMin (i.e if  $52 < f_0 < 94$ )
    if ((effectState._dataLength < vocalFryMax) &&
        (effectState._dataLength > vocalFryMin)){
        interpFraction = ((double)vocalFryMax -
```

3. A Parametric, Real-Time Voice Source Model

```
(double)effectState._dataLength)/((double)vocalFryMax -
(double)vocalFryMin);

    if (effectState._voiceType == 3) { // if BREATHY voice
is selected, interpolate from vocal fry params to breathy params
        effectState._noiseOn = TRUE;
        effectState._noiseDuration = 0.5;
        effectState._noiseStart = 0.75;

        effectState._noiseAmount = 0.025*interpFraction;
        effectState._teVal = (0.251*(1-
interpFraction))+0.756*interpFraction);
        effectState._tpVal = (0.19*(1-
interpFraction))+0.529*interpFraction);
        effectState._taVal = (0.008*(1-
interpFraction))+0.082*interpFraction);

    }

    if (effectState._voiceType == 0) { // if TYPICAL voice
is selected
        effectState._noiseOn = FALSE;
        effectState._teVal = (0.251*(1-
interpFraction))+0.780*interpFraction);
        effectState._tpVal = (0.19*(1-
interpFraction))+0.600*interpFraction);
        effectState._taVal = (0.008*(1-
interpFraction))+0.028*interpFraction);

    }

    if (effectState._voiceType == 1) { // if MODAL voice is
selected
        effectState._noiseOn = FALSE;
        effectState._teVal = (0.251*(1-
interpFraction))+0.575*interpFraction);
        effectState._tpVal = (0.19*(1-
interpFraction))+0.457*interpFraction);
        effectState._taVal = (0.008*(1-
interpFraction))+0.028*interpFraction);

    }
    [self createLFInput];

}

// Set to Falsetto if dataLength is less than falsettoMin
(i.e if f0 > 288 Hz)
if (effectState._dataLength < falsettoMin) {
    effectState._tcVal = 1.0;
    effectState._teVal = 0.770;
    effectState._tpVal = 0.570;
    effectState._taVal = 0.133;
    effectState._noiseOn = TRUE;
    effectState._noiseAmount = 0.015;
    effectState._noiseDuration = 0.5;
    effectState._noiseStart = 0.75;
    effectState._vocalTension = 0.0;
}
```

3. A Parametric, Real-Time Voice Source Model

```
[self createLFInput];
}

// Interpolate from Breathy/Modal to Falsetto if dataLength
is between falsettoMin and falsettoMax (i.e if 207 Hz < f0 < 288 Hz)
if ((effectState._dataLength >= falsettoMin) &&
(effectState._dataLength < falsettoMax)) {
    interpFraction = (((double)vocalFryMax -
(double)effectState._dataLength)/((double)vocalFryMax -
(double)vocalFryMin);

    if (effectState._voiceType == 3) { // if BREATHY voice
is selected, interpolate from breathy params to falsetto
        effectState._noiseOn = TRUE;
        effectState._noiseDuration = 0.5;
        effectState._noiseStart = 0.75;

        effectState._noiseAmount = (0.025*(1-
interpFraction))+0.015*interpFraction;
        effectState._teVal = (0.756*(1-
interpFraction))+0.770*interpFraction;
        effectState._tpVal = (0.529*(1-
interpFraction))+0.570*interpFraction;
        effectState._taVal = (0.082*(1-
interpFraction))+0.133*interpFraction;
    }

    if (effectState._voiceType == 0) { // if TYPICAL voice
is selected
        effectState._noiseOn = TRUE;
        effectState._noiseDuration = 0.5;
        effectState._noiseStart = 0.75;

        effectState._noiseAmount = 0.015*interpFraction;
        effectState._teVal = (0.780*(1-
interpFraction))+0.770*interpFraction;
        effectState._tpVal = (0.600*(1-
interpFraction))+0.570*interpFraction;
        effectState._taVal = (0.028*(1-
interpFraction))+0.133*interpFraction;
    }

    if (effectState._voiceType == 1) { // if MODAL voice is
selected
        effectState._noiseOn = TRUE;
        effectState._noiseDuration = 0.5;
        effectState._noiseStart = 0.75;

        effectState._noiseAmount = 0.015*interpFraction;
        effectState._teVal = (0.575*(1-
interpFraction))+0.770*interpFraction;
        effectState._tpVal = (0.457*(1-
interpFraction))+0.570*interpFraction;
        effectState._taVal = (0.028*(1-
interpFraction))+0.133*interpFraction;
    }
}
```

3. A Parametric, Real-Time Voice Source Model

```
    }

    [self createLFInput];
}

    // if dataLength is between vocalFryMin and falsettoMax,
    keep t values the same
    if ((effectState._dataLength >= 213) &&
        (effectState._dataLength <= 469)) {
        if (effectState._voiceType == 3) { // if BREATHY voice
            is selected, interpolate from breathy params to falsetto
            effectState._noise0n = TRUE;
            effectState._noiseDuration = 0.5;
            effectState._noiseStart = 0.75;

            effectState._noiseAmount = 0.025;
            effectState._teVal = 0.756;
            effectState._tpVal = 0.529;
            effectState._taVal = 0.082;
        }

        if (effectState._voiceType == 0) { // if TYPICAL voice
            is selected
            effectState._noise0n = FALSE;
            effectState._teVal = 0.780;
            effectState._tpVal = 0.600;
            effectState._taVal = 0.028;
        }

        if (effectState._voiceType == 1) { // if MODAL voice is
            selected
            effectState._noise0n = FALSE;

            effectState._teVal = 0.575;
            effectState._tpVal = 0.457;
            effectState._taVal = 0.028;
        }

        [self createLFInput];
    }
}
```

The `createLFInput` method is largely identical to its MATLAB counterpart, with some minor differences. Due to the real-time nature of the waveform calculation, any changes made to the timing parameters that result in discontinuities or abnormal waveforms can produce unwanted digital distortion and clipping. In

order to prevent timing parameter values from falling outside their appropriate ranges, the following lines of code were added:

```

if (te <= tp) {
    te = tp + tp*0.01;
}

if (te >= (tc-ta)){
    te = tc-ta - (tc-ta)*0.01;
}

```

This simply ensures that t_e never falls before the point t_p or after the return phase (at $t_c - t_a$).

Once the LF-waveform parameters have been calculated, they are passed to the effectState structure to be accessed in the main processing loop. The LF-model equations 3.2 & 3.3 are calculated here, with the turbulent noise component created and added to the waveform:

```

if (noiseOn == TRUE) {
// Creates white noise between -1 and +1
noiseSample = rand() % 200;
noiseSample = noiseSample - 100;
noiseSample = noiseSample/100;
// Attenuate noise signal by noise amount selected
noiseAdd = noiseSample*noiseAmount;
// Turbulent noise portion of waveform begins at the closing phase
and ends during the opening phase (i.e. crosses over two pitch
cycles)
// Need to 'wrap' noise around so that the opening phase portion of
noise begins at the start of the pitch cycle
// Calculate remainder (portion of noise that extends beyond pitch
period):
noiseRemainder = (noiseStart + noiseDuration) - dataLength;

// if start time + duration is less than length of period (i.e no
remainder)
if (noiseStart + noiseDuration < dataLength) {
    if (k >= noiseStart + noiseDuration){
        LFcurrentSample1 = LFcurrentSample;
    }
    if (k < noiseStart) {
        LFcurrentSample1 = LFcurrentSample;
    }
}
}

```

3. A Parametric, Real-Time Voice Source Model

```
    }
    if (k > noiseStart & k <= noiseStart + noiseDuration) {
        LFcurrentSample1 = LFcurrentSample+noiseAdd;
    }
}

// if start time + duration is greater than length of period (i.e
noise wraps round)
if (noiseStart + noiseDuration >= dataLength){
    if (k > noiseRemainder && k < noiseStart) {
        LFcurrentSample1 = LFcurrentSample;
    }

    if (k <= noiseRemainder || k >= noiseStart) {
        LFcurrentSample1 = LFcurrentSample+noiseAdd;
    }
}
}
```

The 'auto f0' option allows the user to select a pre-recorded pitch sequence taken from a natural speech recording using Praat [74]. The f0 data from this recording is stored in an array f0Data[]. This data is recorded at a sampling frequency of 100 Hz, while the iOS system sampling frequency is set 44.1 kHz. This means in order to play back the recorded pitch sequence, the f0 must be updated every 441 samples. A variable sampleCount is incremented every loop. When this variable reaches 441, a new f0 value is taken from f0Data[]. In order to prevent waveform discontinuities, the pitch period length is only updated at the beginning of a new cycle (i.e. when the variable kk = 0):

```
if (kk>dataLength) {
    kk = 0;
}

if (sampleCount == f0Update) {
    sampleCount = 0;
}

// Update new value for dataLength based on natural f0 data
if (sampleCount == 0) {
    period = (1/f0Data[f0Count])/overSample;

    f0Count += 1;

    if (f0Count == 100) {
```


3. A Parametric, Real-Time Voice Source Model

```
        f0Count = 0;
    }
}

if (kk == 0) {
    dataLength = floor(Fs*period*overSample);
}

kk += 1;
sampleCount += 1;
```

The interface for the app is a simple design with a single view (fig. 3.8). Only on-screen buttons and sliders are used to provide control of toggle events and continuous variables. The user is given the five voice type options, buttons for automatic pitch and voice type modes, pitch control, ‘vocal tension’ (or speed quotient), vocal fold return rate (the value for t_a) and amplitude. Another button allows the user to start and stop audio processing, with a final button for switching between only voice source synthesis and voice source with vocal tract modelling synthesis (discussed further in the next chapter):

3. A Parametric, Real-Time Voice Source Model

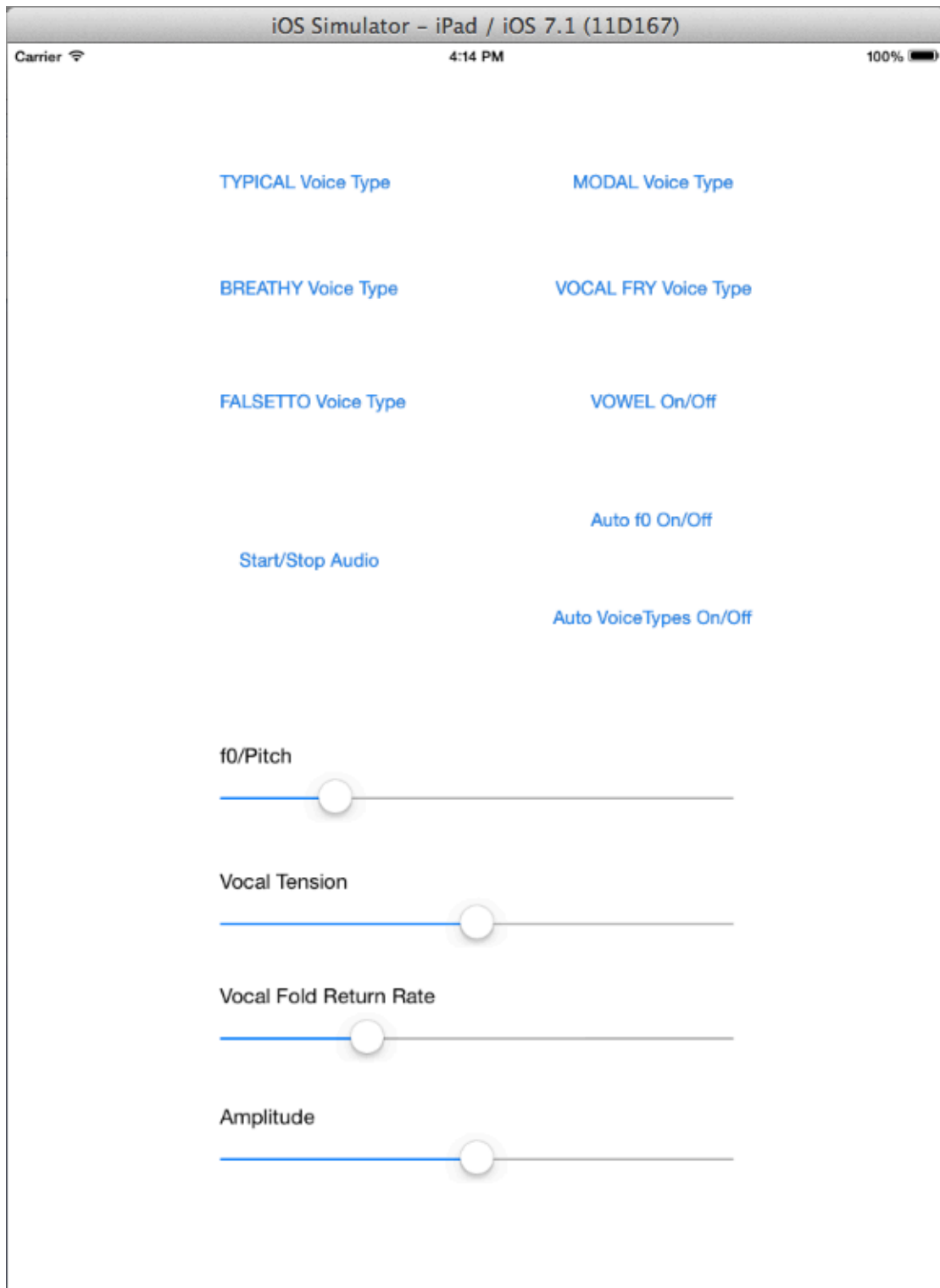


Figure 3.9 – Screenshot of final implementation of 'LFGGen' application interface (a screen-captured demonstration video is available with the accompanying files under 'LFGGenDemoVideo.mp4')

3.5 System Testing

In order to verify the functionality of this application, a set of criteria must be set out that should be fulfilled by a completed system. Based on the specifications and design given earlier in the chapter, the system testing criteria for this application are:

1. Reproduction of the glottal source waveform based on the LF-equation given earlier. The output waveform should resemble that discussed in [12] [14] [21] etc., with matching spectra (i.e. -12 dB/octave slope for falsetto and breathy voice).
2. Fundamental frequency matches that specified by the user.
3. Appropriate manipulation of breathiness, vocal tension, and vocal fold return rate available to the user. These should alter the output spectrum whilst maintaining the overall properties of the LF-waveform.
4. Automatic pitch mode gives an accurate reproduction of the f_0 slide of an existing voice recording.
5. Automatic voice type mode modulations across voice types for specific frequency ranges.

A point of interest from the testing stages is the method by which the voice source model was subjectively verified. Even a highly accurate reproduction of the voice source waveform would sound unrecognisable as a human voice, due to the effects on the signal of the vocal tract. In order to evaluate the voice source model during testing stages, a 3D printed vocal tract model was used (fig. 3.9). These were developed at the University of York, produced from 3D MRI data. The

3. A Parametric, Real-Time Voice Source Model

3D printing process converts this data into a physical object with an acoustic cavity of the exact dimensions as a human vocal tract during a single sung vowel. When attached to a loudspeaker at the glottis end, any acoustic signal can be used to excite the printed tract's acoustic chamber. Some accuracy is lost due to the rigidity of the material used to produce the model when compared with the soft tissue within the vocal tract, as well as the static nature of the vowel produced. However, it was considered that this provided a good enough way of quickly assessing the perceived effects of voice source modulation.



Figure 3.10 – A 3D printed vocal tract model for a sung /A/ vowel. The loudspeaker is attached via a seal at the glottis end.

The process for evaluating the voice source model via the 3D printed vocal tract consisted of noting subjective responses from the author and colleagues to the various modifications made to the voice source model. This was an informal process, however care was taken not to skew external listener's responses with

3. A Parametric, Real-Time Voice Source Model

leading questions such as 'does this version sound more breathy?', for example. Static pitch tests were played at 110Hz except in the case of vocal fry and falsetto, which were also tested at 40 and 300Hz respectively. Unfortunately no acoustic recordings were made of the resulting waveforms when used in conjunction with 3D vocal tract, however, the voice source waveforms can be found in the accompanying data CD under 'Audio'. Varying pitch was applied using a linearly rising and falling f_0 between 20 and 400 Hz, followed by the pre-recorded f_0 trajectory described earlier in this section. These tests were largely brief comparisons between different versions of the same voice type being modelled in order to ascertain which elicited the most positive response.

A more quantitative study was considered for ascertaining the accuracy of each model as well as the 3D printed vocal tract. This would have consisted of taking anechoic recordings of a.) the 'default' LF waveform (from the VocalModel synth implementation) exciting the 3D vocal tract, b.) the extended LF-model for each voice type exciting the 3D vocal tract, c.) both versions of the LF-model used with the 2D DWM digital vocal tract model, and c.) a human voice singing the same vowel and pitch as the synthesised versions. The human voice recording would be recorded first so that the f_0 and amplitude data could also be extracted and applied to the synthesised versions. By comparing the frequency response of each recording, it would be possible to determine the effectiveness of each combination of models in simulating a spoken or sung vowel. As well as time constraints, one of the reasons for not implementing this testing methodology in this study is that fact that in order to ensure a fair comparison across each model

3. A Parametric, Real-Time Voice Source Model

type, a new set of MRI data would have to be captured so that the 3D and digital models were of the same human vocal tract. Currently, the MRI data used for the 2D DWM is taken from a separate study to the 3D vocal tract.

3.5.1 Waveform Reproduction

In order to verify the accuracy of the LF-model equation and its corresponding audio output, plots were made of a full pitch period of each voice type, along with an amplitude spectrum of the signal. The following waveform plots were captured by plotting the data from the PCM-encoded .wav files created in MATLAB (the corresponding audio files can be found under the 'Audio Examples' folder with the accompanying media):

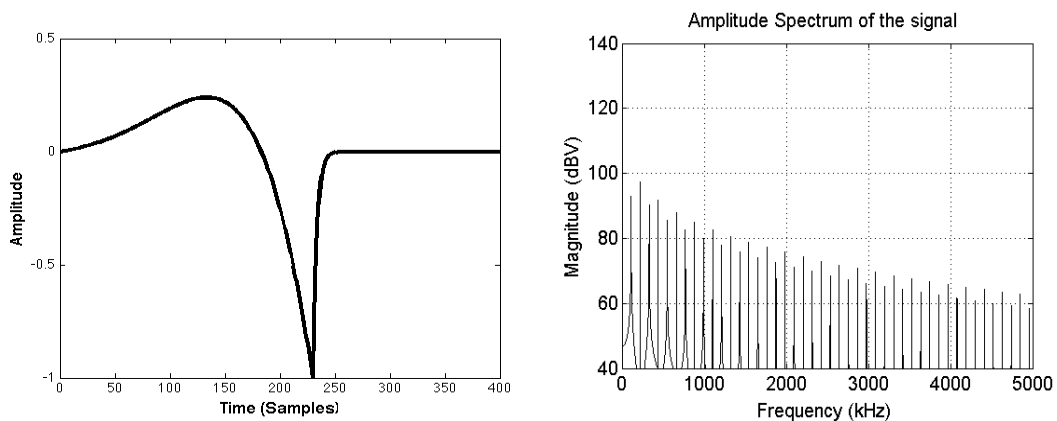


Figure 3.11 – Waveform and spectrum for modal voice type

3. A Parametric, Real-Time Voice Source Model

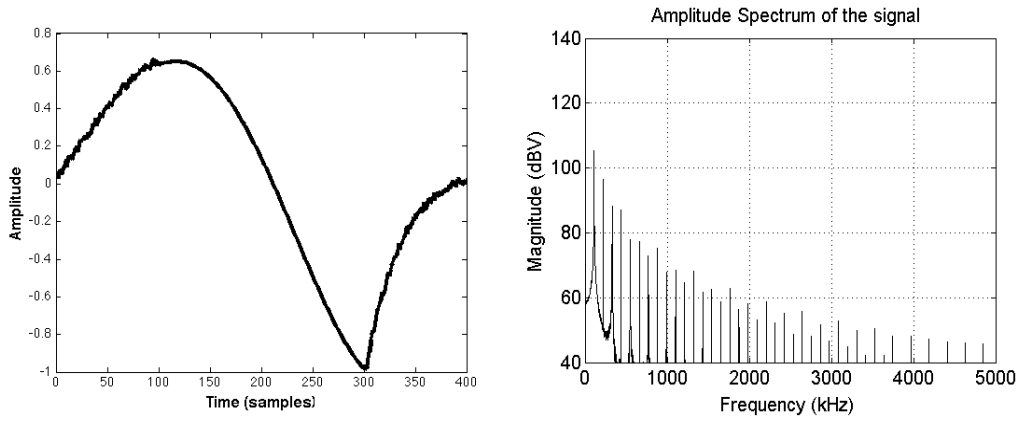


Figure 3.12 – Waveform and spectrum for breathy voice type

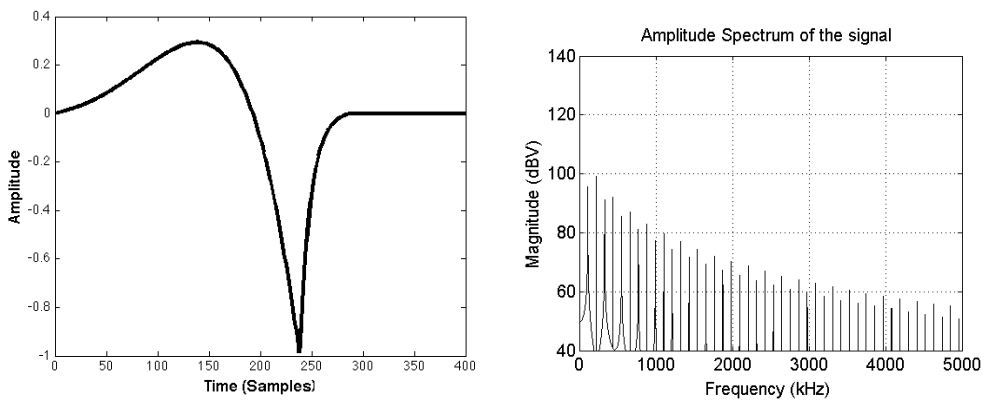


Figure 3.13 – Waveform and Spectrum for vocal fry voice type

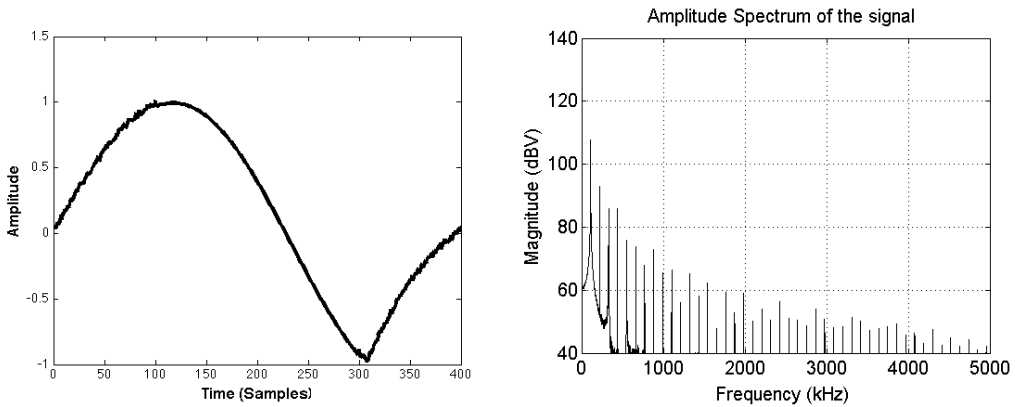


Figure 3.14 – Waveform and spectrum for falsetto voice type

3. A Parametric, Real-Time Voice Source Model

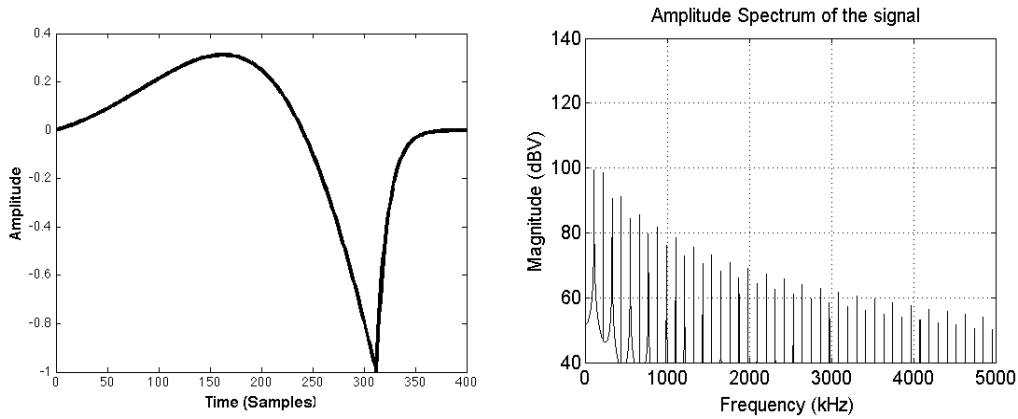


Figure 3.15 – Waveform and spectrum for ‘typical’ voice type

These plots show that the algorithm produces a very similar waveform to the LF-model. It can immediately be seen that the ‘flute-like’ falsetto voice (fig 3.12) more closely resembles a sine wave, whereas the tense vocal fry waveform (fig 3.11) possesses a far steeper return curve with a shorter open quotient.

Figures 3.10 and 3.12 both clearly display a small amount of amplitude modulated white noise at the beginning and end of the waveform. It can be seen from the amplitude spectrum plot that this makes up for the interharmonic white noise portion of the spectrum, consistent with the breathy voice simulations explained in [14]. As Childers et al. found, the effect of the strong low-frequency harmonics masking the noise component can be seen clearly in the spectrum. This confirms that a high-pass filter is not necessary to eliminate perceptible noise below 2 kHz.

3.5.2 Fundamental Frequency

The fundamental frequency for each voice type in MATLAB AND iOS was analysed when set to 24 Hz, 110 Hz and 440 Hz. This was achieved Using the Pitch Detection Nyquist plug-in for audacity [75] routing the system audio from MATLAB and iOS to the DAW using the SoundFlower internal soundcard plug-in [76]. This was considered the optimum method as no digital-to-analogue conversion (and vice versa) was required, leaving the PCM signal largely untouched. All voice types produced the desired pitch when set to 24 and 110 Hz, with small discrepancies (+/- 1 Hz) when set to 440 Hz. This could be purely due to the pitch detection algorithms used within the plug-in (indeed, using the spectrum plotting plug-in yields differing results from the dedicated pitch detection plug-in, and is variable with window types and sizes) but may also require further work to improve the fundamental frequency accuracy.

3.5.3 'Vocal Tension' Parameters

The two parameters identified as capable of producing a noticeable change in vocal tension (i.e relaxed or stressed voices) were the speed quotient (SQ) and the vocal fold return rate (t_a). Tense voices are defined as possessing a 'broad peak in the spectrum at high frequencies' while relaxed voices possess a 'steeply declining spectral slope' [14]. To confirm that altering these parameters produces the desired spectral effects, the 'typical' voice source was synthesised with no vocal tension and a return rate of 0.028. The vocal tension parameter

3. A Parametric, Real-Time Voice Source Model

was then set to -0.3 and +0.2 (found to be the outer limits that this value can be set to before the waveform becomes distorted and clipping occurs). Vocal fold return rate variation is achieved by adding or subtracting a percentage of the default t_a value. This was set to -0.9 then 0.1. The resulting waveform plots and spectra are presented in figures 3.15 to 3.22.

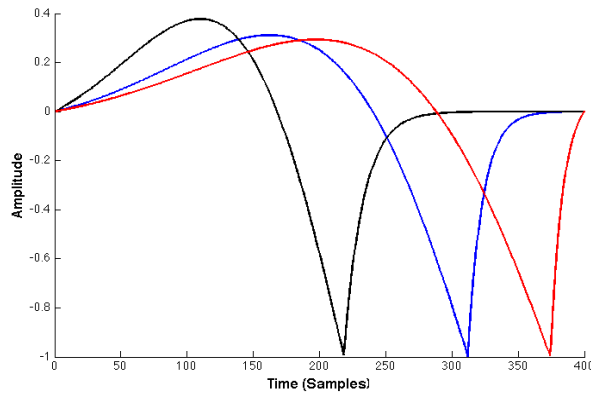


Figure 3.16 – ‘Typical’ voice type waveform with minimum (black) and maximum (red) ‘vocal tension’ values superimposed

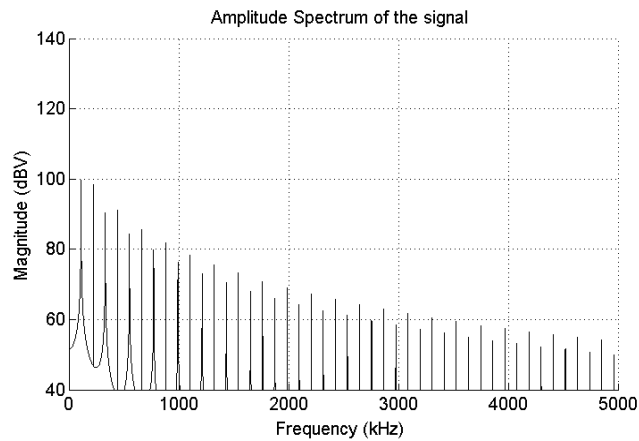


Figure 3.17 – ‘Typical’ voice type spectrum with no adjustments to ‘vocal tension’ (VT)

3. A Parametric, Real-Time Voice Source Model

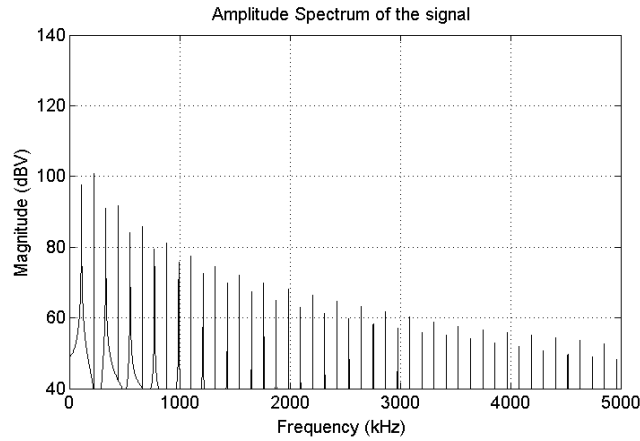


Figure 3.18 – ‘Typical’ voice type with minimum VT

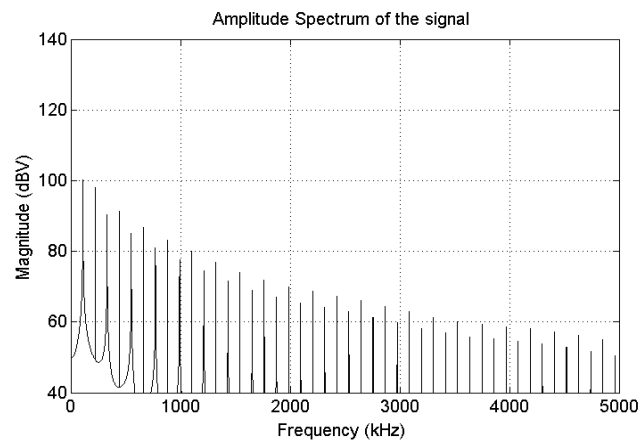


Figure 3.19 – ‘Typical’ voice type spectrum with maximum VT

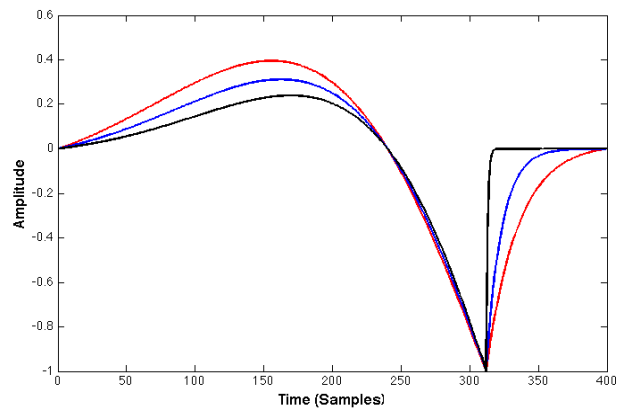


Figure 3.20 – ‘Typical’ voice type waveforms with minimum t_a value (black), default (blue) and maximum (red) superimposed.

3. A Parametric, Real-Time Voice Source Model

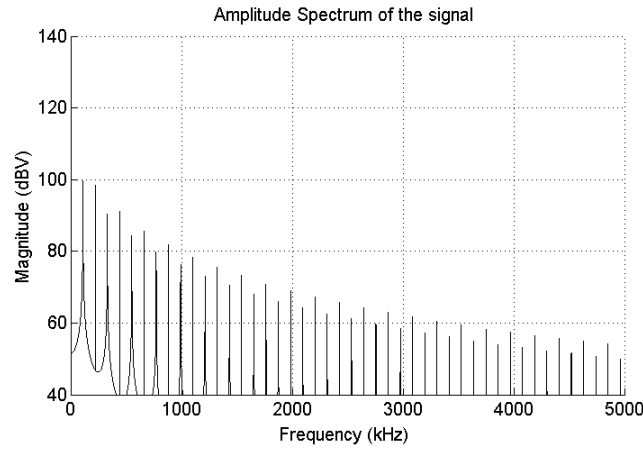


Figure 3.21 – ‘Typical’ voice type spectrum with no adjustments to t_a value

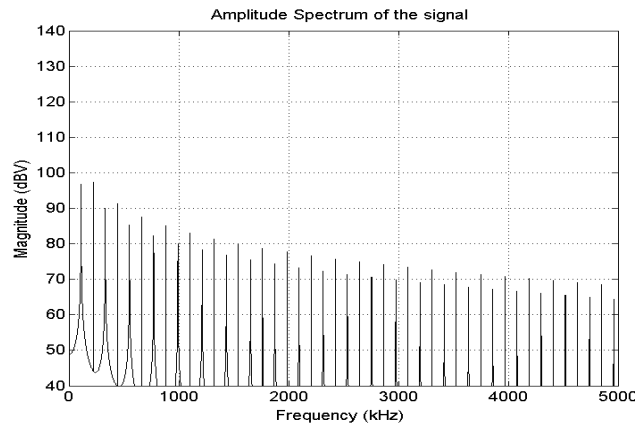


Figure 3.22 – ‘Typical’ voice type spectrum with minimum t_a

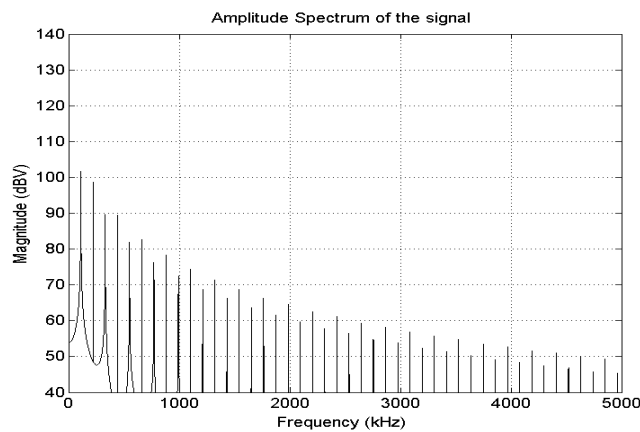


Figure 3.23 – ‘Typical’ voice type spectrum with maximum t_a value

3. A Parametric, Real-Time Voice Source Model

The spectra show that an increase in the VT parameter provides a small boost in high frequencies above 3 kHz. A steeper high-frequency roll-off occurs with a low SQ. From the waveform plots, it is clear that the most marked effect of altering the SQ is the extension of the opening phase and decrease in the positive peak amplitude due to the area balancing condition. While the effect on the waveform is quite extreme, the audible spectral effects are far more subtle. This is due to the main excitation of the glottal source being the negative peak and the return slope gradient, which is only marginally increased with the SQ value, and has no variation in amplitude.

Variations in t_a value provide more distinct spectral effects, in line with Childer's definition of hypo-/hyper-tension. This also provides a more stable alternative, as explained in section 3.3; t_a can be altered over a wide range of values without producing discontinuities, while maintaining the overall LF-model 'shape'.

3.5.4 Automatic Pitch-Dependent Voice Types

A key feature of this application designed to mimic the way the human voice source responds to pitch is the automatic modulation between voice types, with vocal fry voice for low frequencies, modal, breathy or 'typical' for mid-range, and falsetto for high frequencies. In natural speech the voice type does not instantly change as it passes certain pitch thresholds, but modulates between voicings.

3. A Parametric, Real-Time Voice Source Model

A recording of the output from iOS was made using Audacity, in order to closely analyse the waveforms produced by the 'auto-voice' function. An f_0 sweep over the entire accessible range of the app (24–440 Hz) was made and the output waveform image produced (fig. 3.23):

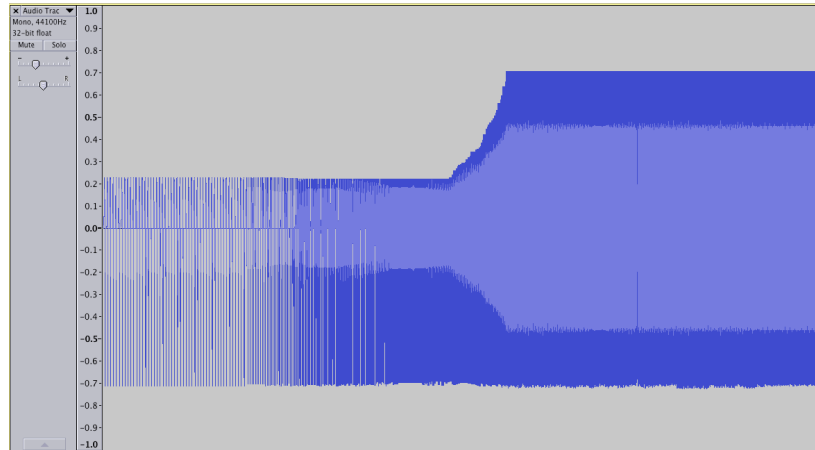


Figure 3.24 – Waveform of audio output produced with 'auto-voice' function enabled, with an f_0 sweep from 24-440 Hz

From observing the full waveform, it is immediately obvious that the interpolation between modal and falsetto induces a much larger positive peak. This is due to the falsetto waveform's close similarity to a sine wave, with positive and negative peaks at almost equal magnitudes. This is an artefact from the area balance condition in the LF equation (equation 2.1). Because the negative peak is set to -1, as the waveform becomes more sinusoidal, the positive peak approaches +1. This produces a perceived level increase that future implementations of the falsetto voice type would ideally take into account.

3. A Parametric, Real-Time Voice Source Model

Zooming in, it is possible to observe the vocal fry waveform from 24-52 Hz (fig. 3.24):

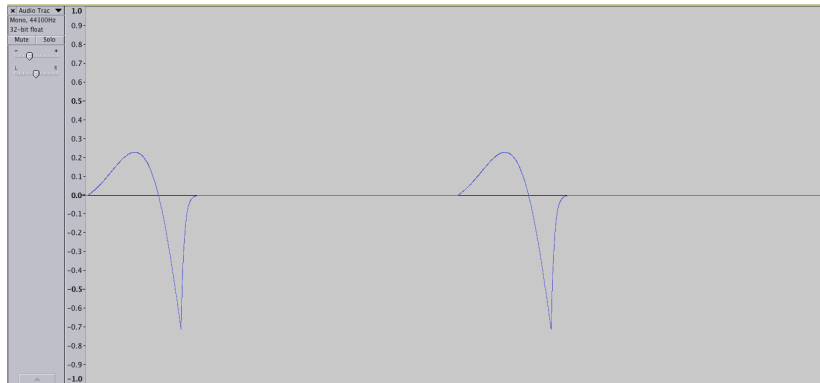


Figure 3.25 – Two cycles of waveform between 24-52 Hz with ‘auto voice-type’ enabled

Between 52-94 Hz, the voice type modulates from vocal fry to modal. This is evidenced by the shorter open phase and shallower return slope (fig. 3.25):

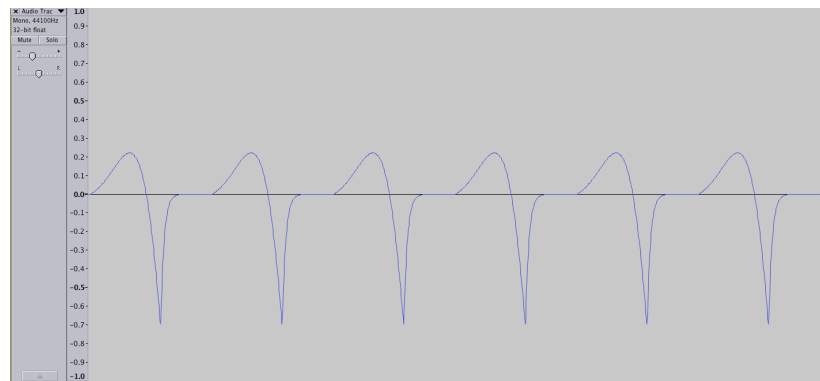


Figure 3.26 – Waveform between 52-94 Hz with ‘auto voice-type’ enabled

3. A Parametric, Real-Time Voice Source Model

From 94-207 Hz, the voice type remains an unaltered modal voice (fig. 3.26):

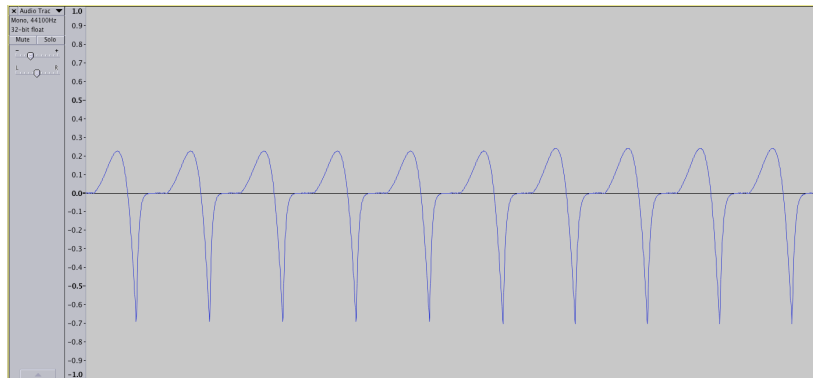


Figure 3.27 – Waveform between 94-207 Hz with ‘auto voice-type’ enabled

From 207-288 Hz, the modal voice type modulates to a falsetto waveform, with the noise quotient increasing in amplitude over this range (fig. 3.27):

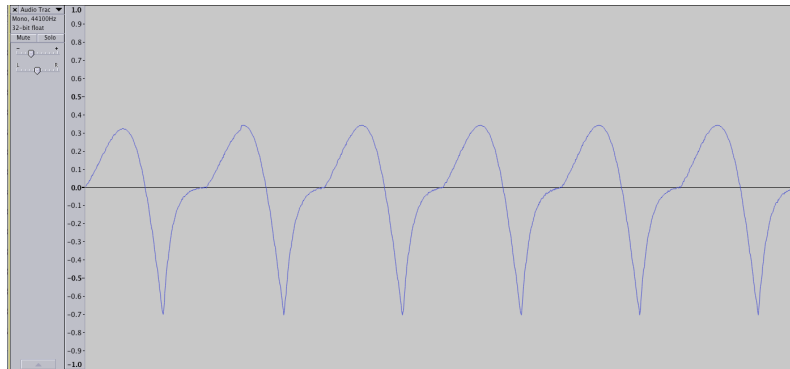


Figure 3.28 – Waveform between 207-288 Hz with ‘auto voice-type’ enabled

3. A Parametric, Real-Time Voice Source Model

Finally, at 288 Hz and above, a pure falsetto waveform is produced, with noise quotient at around 5% of the total amplitude (fig. 3.28):

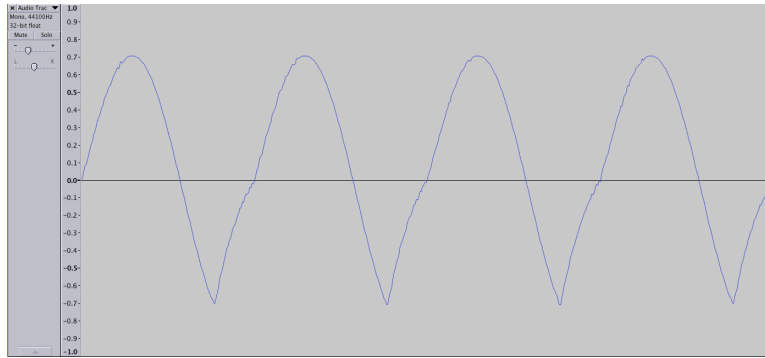


Figure 3.29 – Waveform above 288 Hz with ‘auto voice-type’ enabled

An important note on the ‘auto-voice’ function: the ‘default’ voice (i.e the voice type for mid-range frequencies) will always revert back to the most recently selected voice type. This means that if the user selects Vocal Fry on-screen, then uses the auto-voice feature, the vocal fry waveform will be also used for mid-range frequencies.

3.5.5 Automatic f_0 Trajectory

Figure 3.29 below displays the waveform of the original voice recording and its corresponding spectrogram. The phrase lasts just under one second and modulates over a range from 75 Hz to 121 Hz:

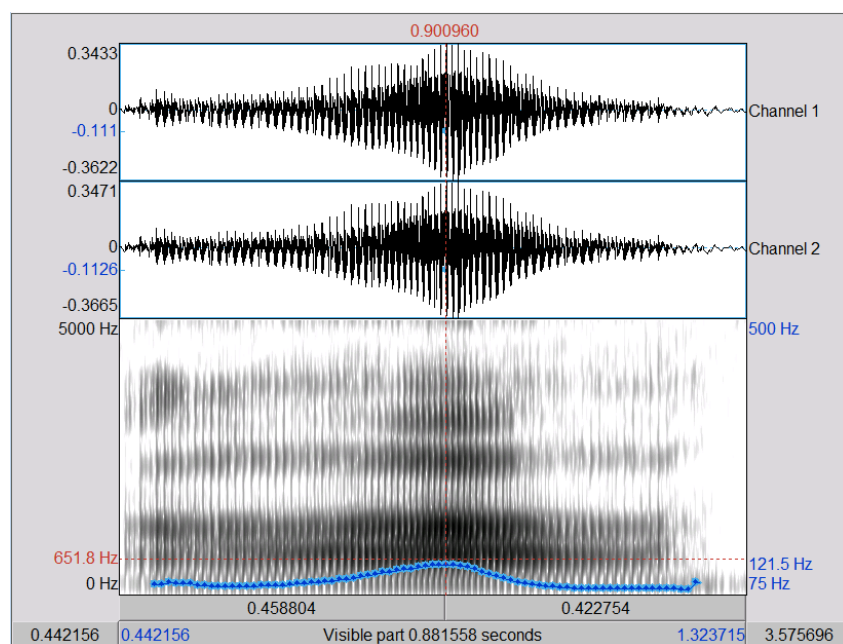


Figure 3.30 – Screenshot of Praat software spectrogram produced from an audio recording of a human voice producing an /A/ vowel with a modulating f_0

After synthesis using the same f_0 data taken from the Praat software, the waveform was stored as a .wav sound file and loaded back into praat. The fundamental frequency was then analysed following the same process (fig. 3.30):

3. A Parametric, Real-Time Voice Source Model

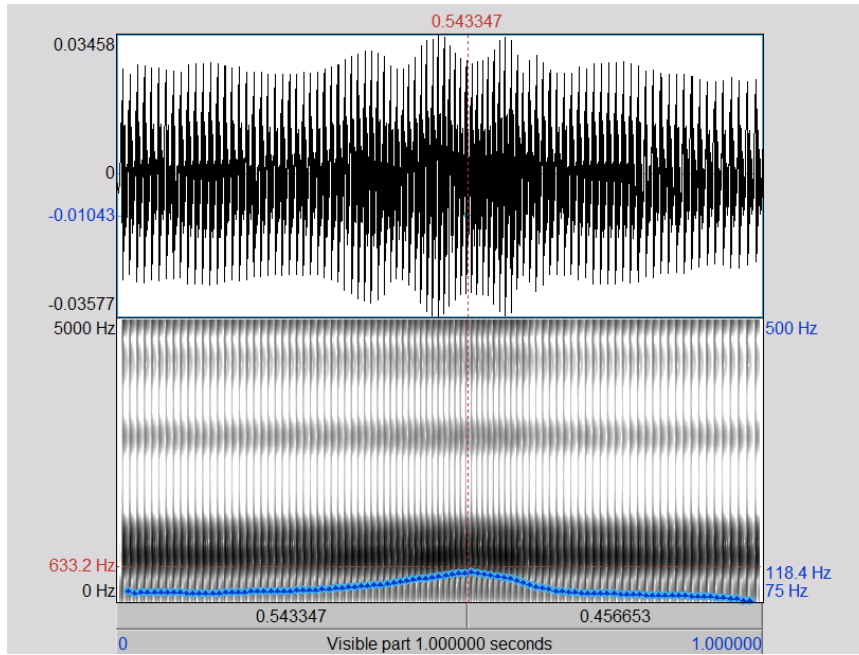


Figure 3.31 - Screenshot of Praat software spectrogram produced from an audio recording of a synthesised voice producing an /A/ vowel with a modulating f_0

It can clearly be seen that the resynthesised audio possesses a remarkably similar, though not identical, f_0 trajectory to the original voice recording. The resynthesised audio lasts for just over 0.1 second longer, with a peak frequency of 3 Hz less than the original recording. This indicates that there is some minor discrepancy between the sampling frequency of the original audio and the playback in MATLAB. It is also worth noting that the synthesised audio was produced with the 2D DWM vocal tract model (discussed in detail in the next chapter) enabled. The spectrogram shows similar formant patterns to the original recording.

3.6 Conclusions

In this chapter, the key concepts to consider for the final app design have been explored. The effects of modulating each of the available LF-parameters were presented, and an appropriate set of parameters available to the user were chosen based on these findings. Responses to subjective listening from both the author and colleagues, corroborated by literature on the subject, showed that an f_0 trajectory taken from existing voice recordings provided a greater sense of realism. The acoustic attributes of each voice type were considered, including the turbulent noise portion of the breathy and falsetto voice types. The final design was set out, with key stages of the implementation process documented. System test results confirm that voice type waveforms and spectra are sufficiently similar to those described in the literature. Alterations to perceived 'vocal tension' can be achieved by altering the vocal fold return rate, with the spectra associated with 'hypo-/hyper-tension' similar to those described in [14]. The 'auto-voice' function modulates voice types across the pitch range at the intended values.

4. Vocal Tract Modelling

This chapter details the vocal tract modelling techniques that were implemented as part of the final application. Although this research focuses on the voice source, a large portion of the development and design stages were focused on vocal tract modelling. As the voice source model described in the previous chapter is designed to work with existing voice synthesis software, a suitable mode of voice synthesis was chosen. The 2D Digital Waveguide Mesh (DWM) vocal tract model described in [30] was chosen, as a real-time articulatory synth allows a similar level of flexibility and expression as is intended for the vocal source model.

The fundamentals of the 2D DWM are described in detail in section 2.5. For clarity, a brief restatement of the 2D DWM vocal tract model is given below. This is followed by a technical report of the implementation stages in MATLAB and iOS, and finally the results from the system testing are presented.

4.1 Vocal Tract Modelling with the 2D DWM

The digital waveguide is a means of representing wave propagation within a digital system. In acoustic terms, a one-dimensional waveguide is implemented as a bi-directional digital delay line. 1D digital waveguides (DWGs) are useful for modelling wave propagation in one dimension, such as vibrations on a string or

acoustic pressure waves in a tube [77]. Terminations, reflections and changes in acoustic impedances (such as the nut on a guitar string or the holes in a flute bore) can be modeled using 1D waveguides by altering the amplitude of the propagated signal between delay units.

Extensions to the 1D waveguide method include 2D and 3D waveguide meshes using a variety of topologies. The effectiveness of these synthesis methods can also be improved using accurate models of the input source signal (i.e turbulent airflow models for flute synthesisers, and excitation for guitar 'plucks').

The 2D digital waveguide mesh (DWM) extension has been shown to provide a computationally efficient and acoustically accurate model of the human vocal tract [30] [49] [50]. The 2D DWM allows a further dimension of reflection to the 1D DWG, with widthwise reflections allowing for more complex wave propagation to be modeled. The following summarises the 2D DWM process as described in detail in section 2.5:

1. Cross-sectional area function data of the vocal tract for a set of vowels is obtained via MRI imaging techniques.
2. Area function data is stored as a set of discrete area values at regular intervals, and the size of a single waveguide is calculated based on the length of the vocal tract and the wave propagation distance over the length of one sample step.

4. Vocal Tract Modelling

3. The overall size of the waveguide mesh is calculated based on the size of one waveguide. The area function data is interpolated across the number of waveguides in the x direction.
4. A raised-cosine function converts the area data to width-wise impedance values (waveguide impedance in the y direction)
5. An 'impedance map' is calculated based on the pressure values at each waveguide junction (this is an average of the incoming pressure from surrounding junctions). Incoming pressure values at the left-most end of the DWM (i.e the glottis end) are taken from the current sample produced by the voice source model.
6. Output pressure is taken as the sum of all rightmost junctions multiplied by a lip radiation value.

As discussed in section 2.5, voice synthesis using vocal tract modelling techniques allows a large variety of phonations and articulations to be directly modelled. A complex enough model could theoretically reproduce any articulation possible with the human vocal tract.

A 2D DWM implementation of the vocal tract model that provided the design for this project is the VocalModel software, described in [71]. This was developed by Jack Mullen at the University of York, and was the first vocal tract model capable of producing articulations in real-time, thanks to a technique known as dynamic impedance mapping. This technique allows for a static DWM structure, whilst manipulating the impedance values at the boundaries to emulate variations in the area function being modeled. This was found to be far more computationally

efficient than recalculating the mesh size and shape for each area function, and also allowed dynamic modulation between vowel area functions, allowing for diphthongs and short vowel-only phrases to be synthesised.

Spectral analysis of the resulting synthesis output showed that the impedance-mapped 2D DWM approach could be used as a somewhat accurate formant synthesis method. The increased dimensionality provides simulation of non-planar acoustic effects such as cross-axial modes which are not represented by the 1D DWG alternative. Concessions are made to the fact that the 2D DWM method utilises one-dimensional area function data obtained by considering the vocal tract as a straight cylindrical tube of varying diameter, as with the 1D counterpart. It is acknowledged that the 2D DWM presented in Mullen's thesis is more of a proof of concept that the heightened dimensionality provides an extra degree of accuracy, and that the extension to a 3D mapping of an asymmetric, non-cylindrical vocal tract model is viable. Despite these concessions, the results were comparable with formants produced by the tried-and-tested 1D DWG method, and diphthongs were reproduced in a stable and accurate manner with no discontinuities [30]. For these reasons (as well as the extensive documentation made available by Mullen), this 2D DWM implementation was chosen as the basis for the vocal tract model used with the current software.

4.2 Implementation of the 2D DWM in MATLAB

The full code listing discussed in this section can be found in the appendix under 'LFModelStaticVowel.m'

The original VocalModel software was written using C++ and compiled as a Microsoft Windows application. In order to port sections of this software to iOS, an interim MATLAB port was considered worthwhile. This would allow the development and testing of a stable vocal tract model within an environment suited for straightforward and in-depth debugging.

[NB: The 2D DWM MATLAB port described below is heavily based on Amelia Gully's work, which was produced at the University of York Audio Lab at the same time as this research]

The primary difference between the MATLAB and C++ implementations is the move from object-oriented programming to MATLAB's procedural processing. This meant that each individual function found in the VocalModel source code was consolidated into one process. The advantage of this is the removal of any processes to do with real-time articulations, meaning only the fundamentals of the 2D DWM model are required to recreate the vowel. The disadvantage is that real-time articulations are impossible using MATLAB.

The first lines of code define the size of the DWM based on the average length of the human vocal tract, and the size of one waveguide based on the sampling frequency and speed of sound:

```
vtLength = 0.175;
vtWidth = 0.05;

wgSize = sqrt(2) * 343 / Fs;

sizeXMax = floor(vtLength/wgSize);
sizeYMax = floor(vtWidth/wgSize);
```

The waveguide mesh 'grid' is made up of two sets of two-dimensional arrays: pressure and impedance. The pressure arrays are made up of incoming and outgoing pressure values for each junction, whilst the impedance arrays store impedance values between junctions:

```
% Pressure:
pNPlus = zeros(sizeYMax, sizeXMax);
pNMinus = zeros(sizeYMax, sizeXMax);
pEPlus = zeros(sizeYMax, sizeXMax);
pEMinus = zeros(sizeYMax, sizeXMax);
pSPlus = zeros(sizeYMax, sizeXMax);
pSMinus = zeros(sizeYMax, sizeXMax);
pWPlus = zeros(sizeYMax, sizeXMax);
pWMinus = zeros(sizeYMax, sizeXMax);

% Impedance
zNorth = ones(sizeYMax, sizeXMax);
zEast = ones(sizeYMax, sizeXMax);
zSouth = ones(sizeYMax, sizeXMax);
zWest = ones(sizeYMax, sizeXMax);
```

The vowel area function data is stored in an array of size `nslices`, so this data is interpolated across an array of size `sizeXMax`. This area data is converted to impedance data, so that a 1D array containing length-wise impedance values is stored. To convert this 1D information to two dimensions, a raised cosine

function is applied to the impedance value at each data point, providing a width-wise array of impedance values, with maximum impedance at the outer edges and minimum impedance at the centre.

The width-wise raised cosine function allows for a total reflection at the outer edges of the waveguide mesh, whilst providing a central channel for signal propagation. A raised cosine of sufficient amplitude allows for an obstruction in the mesh, allowing for plosives and glottal stops to be simulated [51].

Now that the 2D impedance map has been defined, the incoming and outgoing acoustic pressure at each junction is calculated. Nested 'for' loops cycle through every value in the 2D arrays and calculate the pressure value for the current junction based on the average of all incoming pressures and impedance values:

```

for x = 1:sizeXMax
    for y = 1:sizeYMax
        % Calculate pressure at current junction
        pJ = 2*((pNPlus(y,x)/zNorth(y,x) + ...
                pEPlus(y,x)/zEast(y,x) + ...
                pSPlus(y,x)/zSouth(y,x) + ...
                pWPlus(y,x)/zWest(y,x))) / ...
            ((1/zNorth(y,x)) + (1/zEast(y,x)) ...
             + (1/zSouth(y,x)) + (1/zWest(y,x)));

        % Calculate outgoing pressures from junction
        pNMinus(y,x) = pJ - pNPlus(y,x);
        pEMinus(y,x) = pJ - pEPlus(y,x);
        pSMinus(y,x) = pJ - pSPlus(y,x);
        pWMinus(y,x) = pJ - pWPlus(y,x);
    end;
end;

```

At each timestep, the incoming pressure values are updated, with reflection values applied to the mesh boundaries. At this point, the current input sample is taken from the LF-waveform output and applied to west-going pressure values at the glottis end (i.e when $x = 1$). Finally, the output pressure is taken as the sum of all right-most pressure values multiplied by the lip radiation amount, and divided by lip impedance (the right-most impedance value).

4.3 Implementation in iOS

The full code listing discussed in this section can be found in the appendix under 'AudioEngine.m' and 'AudioEngine.h'.

The iOS implementation of the 2D DWM is a fairly straightforward port from MATLAB to C (the app is written in Objective-C, however C language is supported and in fact recommended for audio applications [67]). The intention is to test the compatibility between the LF-model and the 2D DWM synthesis techniques and to ascertain the plausibility of these two methods as a full voice synthesis package for portable devices. As such, dynamic vowel articulations are considered unnecessary at this stage of development. This allows for a far more streamlined implementation of the 2D DWM than that found in the VocalModel source code, as only one vowel area function is accounted for, with the waveguide structure also remaining static. This allows for the size of the waveguide mesh to be defined as a constant in the header file, using values obtained from the MATLAB script:

```
#define SIZE_X_MAX 15
#define SIZE_Y_MAX 4
```

Using a constant vowel shape also allows the area function data array to be initialised directly, rather than loaded from a text file (found in the `createDWM()` method):

```
double areaData[] = { 0.5625,
    0.4620,
    0.2074,
    0.2139,
    ...
    4.2713,
    4.6729,
    5.0273};
```

The key difference between the MATLAB and iOS implementation is the use of 1D arrays to store pressure and impedance values. This is due to the fact that the arrays are defined and calculated outside of the main processing loop, and must be passed to the `effectState` structure in order to be referenced at each timestep. Passing a 2D array between methods requires the use of multiple pointers, whereas a 1D array can be passed straight to the structure and then to the processing loop. This allows for a much more straightforward implementation, but requires a small amount of array manipulation.

A 2D array such as `my2DArray[Y][X]` can be considered equal to a 1D array `my1DArray[Y*X]`. In order to initialise and address the 1D array, the value at `my2DArray[a][b]` is equal to `my1DArray[a*X + b]`. In this manner, the pressure and impedance arrays are initialised:

```
// initialise empty arrays for pressure and impedance
```

```

// Pressure:
Float32 pNPlus[sizeYMax*sizeXMax];
Float32 pNMinus[sizeYMax*sizeXMax];
Float32 pEPlus[sizeYMax*sizeXMax];
Float32 pEMinus[sizeYMax*sizeXMax];
Float32 pSPlus[sizeYMax*sizeXMax];
Float32 pSMinus[sizeYMax*sizeXMax];
Float32 pWPlus[sizeYMax*sizeXMax];
Float32 pWMinus[sizeYMax*sizeXMax];
// Initialise pressure arrays to 0
for (int y = 0; y < sizeYMax; y++) {
    for (int x = 0; x < sizeXMax; x++) {
        pNPlus[y*sizeXMax + x] = 0;
        pNMinus[y*sizeXMax + x] = 0;
        pEPlus[y*sizeXMax + x] = 0;
        pEMinus[y*sizeXMax + x] = 0;
        pSPlus[y*sizeXMax + x] = 0;
        pSMinus[y*sizeXMax + x] = 0;
        pWPlus[y*sizeXMax + x] = 0;
        pWMinus[y*sizeXMax + x] = 0;
    }
}

// Impedance
Float32 zNorth[sizeYMax*sizeXMax];
Float32 zEast[sizeYMax*sizeXMax];
Float32 zSouth[sizeYMax*sizeXMax];
Float32 zWest[sizeYMax*sizeXMax];
// Initialise impedance arrays to 1
for (int y = 0; y < sizeYMax; y++) {
    for (int x = 0; x < sizeXMax; x++) {
        zNorth[y*sizeXMax + x] = 1;
        zEast[y*sizeXMax + x] = 1;
        zSouth[y*sizeXMax + x] = 1;
        zWest[y*sizeXMax + x] = 1;
    }
}

```

As with the MATLAB script, the reflection coefficients are defined here (lines 1416-1418), and the impedance power is defined (line 1422). Lines 1442-1461 contain the linear interpolation code as well as the conversion from area data to impedance. The minimum impedance value is then found and raised to the area power (lines 1463-1470).

Lines 1476-1517 contain the raised cosine function and its application to the impedance map:

```

for (int i = 0; i<sizeYMax; i++) {
    raisedCosine[i] = (0.5 + ...
    0.5*cos(2*MY_PI*((double)i/(double)(sizeYMax-1))));
}

```

Lines 1527-1553 pass all arrays and variables required in the processing loop to the `effectState` structure.

The waveguide synthesis portion occurs in the main processing loop. First, an 'if' statement checks if the user has selected the vowel synthesis function (line 287). If false, the waveguide synthesis is skipped, and the output sample is set to `LFcurrentSample1`. Otherwise, the same process is followed as in the MATLAB implementation. The pressure at the current junction and its outgoing pressure values are calculated from lines 288-304.

The acoustic pressure values are then propagated to their adjacent sampling points. For example, the current outgoing pressure value for east-going signals (stored in `pEPlus[]`) is assigned the incoming pressure value from the corresponding west-going signal from the previous sample step (stored in `pWMinus[]`). In this way, the acoustic pressure value is propagated in all directions across the waveguide mesh, and averages of these values are taken at the sampling points. If a boundary is reached, reflection coefficients are applied to the signal (lines 306-343).

Lines 346 to 354 sum the right-most pressure values to obtain the total output pressure, which is then divided by the right-most impedance value. It was found

that in order to maintain a consistent amplitude with the non-vowel output, the signal had to be multiplied by 125.

4.4 System Testing

The system testing criteria for the 2D DWM functionality are:

1. Formant analysis corroborates with /A/ vowel formants from VocalModel software and other sources.
2. 2D DWM is capable of synthesising other vowel formants using various sets of area function data.
3. An overall system performance check should be conducted to identify any bugs or anomalies in the system. Unwanted signals and waveform discontinuities should be non-existent or sufficiently trivial so as not to affect the perceived output. CPU and memory usage should be minimal, ideally efficient enough to operate stably on an iOS device.

4.4.1 Formant Analysis

Each vowel sound has its own spectral peak or 'formant' that distinguishes it from other vowels. Articulations in the vocal tract attenuate or augment certain frequencies present in the voice source spectrum. These formant frequencies can vary across gender, age, accent and other factors, so it is difficult to quantify

whether synthesis of a specific vowel is successful. However, comparison to average formant values, such as those given in [30], allows for good enough evaluation of successful vowel synthesis.

NB: the notation of vowels in this subsection is given using the SAMPA alphabet [78], as well as an example use in a word.

Spectral peaks in an audio file can be automatically detected using the Praat software package. A waveform lasting two seconds at a static frequency of 110 Hz, with area function data for an /A/ (as in 'bart') vowel was produced using MATLAB. Figure 4.1 displays the waveform and its accompanying spectrogram, with formant values highlighted:

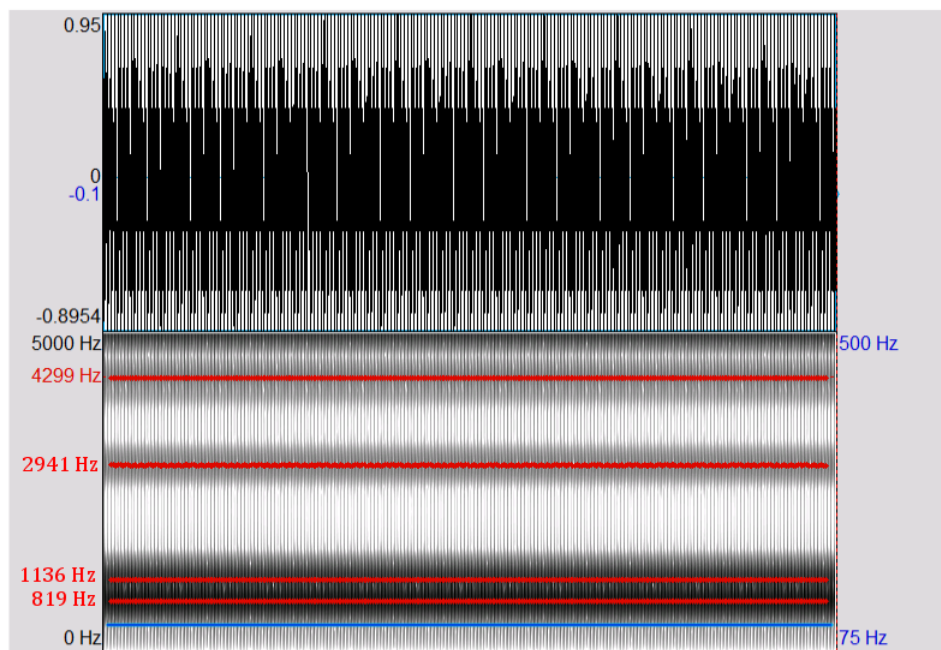


Figure 4.1 – Spectrogram produced from two second audio file of synthesised /A/ vowel using 2D DWM and voice source model set to 'typical' voice type

The average values for F1, F2, F3 and F4, given in [30] are 673 Hz, 1097 Hz, 2457 Hz and 3464 Hz respectively. Whilst the reproduced formants differ somewhat from the average, the high F1 and low F2 values concur with the placement of the /A/ vowel within the vowel chart described in [79], representing the high tongue position at the back of the mouth which produces this vowel.

4.4.2 Multiple Vowels

In order to confirm that this implementation of the 2D DWM vocal tract produces the expected formants for other vowel area functions, five more vowels were synthesised using area function data from the VocalModel source code. The resultant formants were examined using the same methods described above. Figure 4.2 is a reproduction of the vowel chart found in [79] for the six vowels that were simulated and their SAMPA symbols:

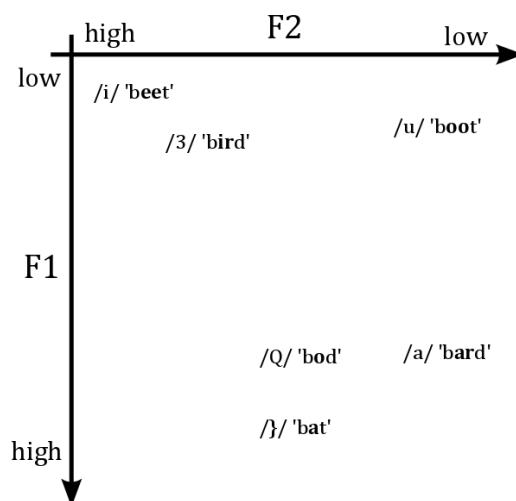


Figure 4.2 – Vowel chart for English vowels

4. Vocal Tract Modelling

The first four formants of each vowel are presented in table 4.1 alongside average formant values for English vowels taken from [30].

Table 4.1 – Formant frequencies produced from 2D DWM vocal tract model versus average recorded formant values in male English speech. The corresponding audio files were created using the ‘Typical’ voice type and can be found in the ‘Audio Files’ folder with the accompanying media.

Vowel	Synthesised Formants (Hz)				Average Formants (Hz)			
	F1	F2	F3	F4	F1	F2	F3	F4
/a/	819	1136	2941	4299	673	1097	2457	3464
/i/	168	2438	2910	4318	303	2172	2851	3572
/ɜ/	447	1731	1936	3257	477	1276	1707	3201
/u/	261	826	2903	4902	342	1067	2219	3342
/Q/	298	893	2792	4578	645	1622	2357	3464
/ʃ/	707	1954	2829	3908	N/A			

It is immediately apparent that whilst some formant values are remarkably similar to the average, most of the synthesised vowel formants deviate by up to several hundred Hz compared with the average (the most marked deviation being the first two formants for /u/ and /Q/). The synthesised results however are similar or identical to those produced using the VocalModel software, which suggests more refinement is needed in 2D DWM vocal tract modelling methods, such as the inclusion of more sophisticated boundary reflection methods, as well as incorporating the nasal tract within the mesh model.

4.4.3 System Performance

Whilst compatibility with mobile devices is ensured through the use of the Objective-C language and the Xcode IDE, physical performance of the app is of equal importance. At the time of writing, the application has been tested solely using the built-in iOS simulator program within Xcode. This is a useful tool for testing and debugging purposes, but does not provide an accurate platform for performance requirements of the variety of iOS compatible devices. A simple way to ensure that the app will run smoothly on a physical device is to use the performance check function within Xcode (fig. 4.3). The application was run with breathy voice selected, vocal tract modelling enabled, and the automatic f0 modulation. This was deemed to be the most memory-intensive combination of functions. The app performed well, with maximum CPU usage of 23% and 19.7 MB of memory used. This is well below the 1 GB+ of memory that current iPad models are equipped with, so assuming the application is used in isolation, performance should not be an issue. This is worth considering for future iterations of the application that may include more complex graphical user interfaces (GUIs) or more computationally expensive synthesis procedures such as dynamic impedance mapping within the 2D DWM.

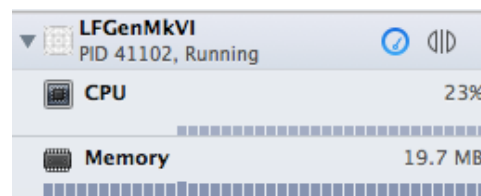


Figure 4.3 – Performance check for ‘LFGGen’ application within Xcode

4.5 Conclusions

In this chapter, methods for acoustic modelling of the vocal tract via digital waveguides were summarised. An in-depth analysis of the MATLAB and iOS ports of the 'VocalModel' 2D Digital Waveguide Mesh vocal tract model was provided, and test results presented. The test results confirm that identical or similar functionality to the results described in [30] [51] and [49] was achieved, indicating a successful port of the software. Significant deviations from the average vowel formant frequencies for English male speakers suggest further refinement of existing methods, however the key focus of this exercise was to establish similar functionality to existing software, and the viability of such methods with mobile devices. Areas for worthwhile future development have been highlighted, including improvements to the interface, accuracy in vowel reproduction, and dynamic vowel articulations.

5. Summary and Analysis

5.1 Summary

The primary focus of this research was to expand upon existing articulatory voice synthesis methods to include a more sophisticated voice source model. This was achieved via the implementation of the well-established LF-model within an iOS application. Secondary research showed that voice qualities such as type, breathiness, tension and so on are highly important in terms of voice perception [15], but are often overlooked where voice synthesis is concerned. It was proposed that a voice source model that is compatible with existing vocal tract modelling methods, but also capable of producing a wider range of vocal features than a simple LF-model or similar excitation method, would improve the perceived 'naturalness' of the synthesised voice. This was achieved using MATLAB and iOS implementations, with input parameters for voice type, breathiness and tension. Other features such as automatic f_0 tracking based on recorded utterances and f_0 -dependent voice type modulation were also implemented to varying degrees of success. This provided the bulk of the research and implementation stages, however a final task was to integrate the voice source model with the existing 2D Digital Waveguide Mesh model of the vocal tract, originally implemented in the VocalModel software [71] and described in [30]. This involved porting sections of the VocalModel source code first to MATLAB for testing and debugging, followed by an Objective-C implementation for iOS.

5.2 Analysis

5.2.1 Voice Source Synthesis using the LF-Model

Chapter 3 described the design and implementation process of an extended LF-model voice source synthesiser within MATLAB and iOS. Implementation in iOS allowed for real-time modification of various parameters. Amplitude and f_0 can be altered on the fly with no detectable distortions. Voice types can be selected during audio processing with instantaneous and easily perceptible results.

The voice types available are: 'typical', modal, breathy, vocal fry and falsetto. The resulting spectra of each of these voice type models are sufficiently similar to those described in [14] and were deemed recognisable as their intended type during informal, subjective listening sessions, although at the extremes in pitch it was harder to differentiate between certain voice types (e.g. vocal fry and modal at low frequencies).

5.2.2 Extensions to the LF-Model

Extensions to the basic LF-model include: real-time modifications to the waveform using the LF timing parameters, an additional white noise generator to simulate turbulent airflow in breathy voice, voice type selection, a 'vocal tension' parameter, automatic f_0 -tracking and f_0 -dependent voice type modulation. Spectrograms taken of the resulting waveforms proved that the extensions were achieved with some success.

Modulation to individual timing parameters (t_e , t_p , t_a and t_c) was originally made possible using the first iterations of the iOS app. Using wavetable synthesis, it was found that adjusting these parameters during playback caused digital distortion. A new method of implementation was tested which involved simply performing the LF-model equations at each sample step. This allowed for the waveform to be updated at any point during the pitch period without affecting the output waveform. It was found, however, that individual alterations either had no significant effect on the output waveform, or produced waveforms that did not fit the LF-waveform archetype. This is due to the cross-coupled nature of these timing parameters. The exception is the value of t_a , which adjusts the exponential return rate of the waveform, corresponding to the rate at which the vocal folds return to the closed position. This can be altered independently and was found to corroborate with Childers' assertion that it can affect the perceived tension present in the voice [14].

Another voice source parameter that has been shown to affect perceived 'vocal tension' is the speed quotient (SQ) [14] [15] [16]. An SQ variable called Vocal Tension was implemented in MATLAB and iOS, which altered the values of t_e , t_p , and t_a to modify the timing positions of the minimum and maximum peaks in a cross-coupled manner. This has a similar, yet less pronounced, effect on the spectrum as alterations of t_a alone. Simply modifying t_a proved also to be a more stable method than the SQ option due to the larger range of values that could be produced whilst maintaining a stable LF-waveform.

The addition of a random number generator to produce a relatively weak white noise signal allowed for turbulent airflow to be simulated for breathy and falsetto voice. This additional signal was pulse modulated in accordance with [14] and set to an amplitude of 5% of the peak amplitude. The resulting spectra displayed a noticeable amount of white noise above 2 kHz. Subjective listening also provided promising results, with an audible noise component that provided a distinctly noticeable 'breathy' component.

Fundamental frequency tracking using the Praat software showed that resynthesis of a pre-recorded f_0 trajectory was successful. It has already been suggested that constantly varying f_0 signals provide an enhanced realism to vowel synthesis [17], which would appear to be the case with this feature.

It was recognised early on in the research that simply allowing for various voice types to be synthesised is not sufficient for human voice source modelling. Voice quality is dependent on many factors such as pitch and amplitude. A fairly rudimentary attempt at modelling this behaviour was implemented within the iOS app. With 'Auto VoiceTypes' selected, as the fundamental frequency rises from 24 to 440 Hz, the voice type modulates from Vocal Fry to Falsetto, with an interpolation stage between Vocal Fry-Modal/Breathy and Modal/Breathy-Falsetto.

5.2.3 Use within 2D DWM Vocal Tract Model

Analysis of the output spectrum showed that both the voice source and vocal tract models operated as expected, with appropriate modifications to the signal when various voice source parameters were adjusted. Formant frequencies varied in terms of their correlation to average values in English male speakers, but this was also found in the original implementation of the 2D DWM vocal tract.

At present, vocal tract modelling within the software is only applicable with a static vowel. This provided sufficient information to determine feasibility of 2D DWM modelling within iOS, but does not allow for further extensions to the LF-model such as vowel-dependent modulations to the voice source as documented in [7].

5.2.4 Core Aims

The primary aims of the project (as stated in section 1.1 and 3.1) were to develop an application that could be applicable to voice science research as well as assistive technology development. It is considered that these core aims were met, at least partially, in a ‘proof-of-concept’ manner. In terms of a contribution to voice science and synthesis research, the application developed provides a versatile and expressive mode of voice source production that is capable of producing LF-model waveforms for five voice types. As far as existing research is concerned, the spectral qualities of the voice types produced are consistent with

previous studies. In addition to voice type selection, 'vocal tension' has been explored as a parameterised feature of the voice source model, and has to some extent been successfully implemented. Real-time modifications to the vocal tension parameters produce noticeable digital distortion, which does limit the expressiveness of this particular feature, however if a tension value is selected before a phrase is synthesised, no distortion is present. In the case of full speech synthesis, this would be sufficient for adding or removing tension to a single word or phrase. Other parameters such as 'breathiness', vibrato or glottal gap size could easily be made available to the user for further research using the app.

The contribution of this project towards assistive technology development is less conclusive. The implementation of a somewhat sophisticated voice synthesis engine within a portable device proves that vocal tract modeling synthesis is a viable method for assistive technology purposes. By extending the functionality of the voice source model, a larger, more expressive range of voice qualities can be modeled than current voice synthesis techniques commonly associated with mobile devices, such as Siri [1], which makes use of concatenative synthesis – a technique far more limited in terms of voice type, as well as pitch and rhythm. However, at this stage of development of both the voice source model and the 2D DWM vocal tract model, entire words and phrases cannot be recognisably created, and so effectiveness of the system's use within an assistive technology context remains speculative.

5.3 Future Research

As stated above, the primary focus of this research was to expand on existing methods of voice source modelling in order to explore more natural sounding voice synthesis techniques. There are many potential avenues towards achieving this goal, many of which fall outside of the scope of this project, or were considered only after the work described herein was completed. The following section summarises the main considerations for future work in this area.

5.3.1 Issues within LF-Model implementation

Some extensions to the LF-model that were included in this application were either intended as a proof of concept, or found to be far more complex than originally considered. Because of this, fairly rudimentary implementations of these extensions were used, leaving room for expansion on pre-existing methods and ideas. One such area for improvement is the f_0 -dependent voice type modulation technique. This method would benefit from further primary research into voice type modulation in natural speech. As no voice source recordings were made specifically for this research, voice type information such as pitch range and timing parameters was taken from secondary sources. It would be beneficial to probe the effects of pitch on voice source through inverse filtering or similar techniques, in order to ascertain the ranges in which certain voice types are present. This would also go towards defining the behaviour of the voice source as it modulates between voice types, so that a more complex technique than linear interpolation could be employed at the thresholds between voice types.

This could also be used to develop amplitude-dependent voice source modulation.

Two parameters for 'vocal tension' were made available to the user: SQ and t_a . Both of these were shown to have a noticeable effect on the spectral qualities for vocal tension as outlined in [14]. However, the use of vocal tension in natural speech is clearly a complex issue, as it relates to perceived emotional content of speech, as well as being affected by factors such pitch, amplitude and vocal pathology. Further research into vocal tension, perhaps through detailed analysis of inverse-filtered speech waveforms, would allow a more complex, rule-based vocal tension parameter to be incorporated within the model. At present, the inclusion or exclusion of vocal tension in the voice source model is defined only by the user, whereas a thoroughly verified rule-based system would prevent inappropriate use of a 'tense' or 'relaxed' voice source.

5.3.2 Further Extensions to the Voice Source Model

This implementation of the voice source model focused purely on voice types that could be modeled as a single pitch-period of an LF-model waveform variant. This excludes other voice types such as whisper and harsh voice [15] as well as accurate vocal fry representation, whose pitch period is in fact made up of several weaker LF pulses following the initial pulse (fig. 5.1).

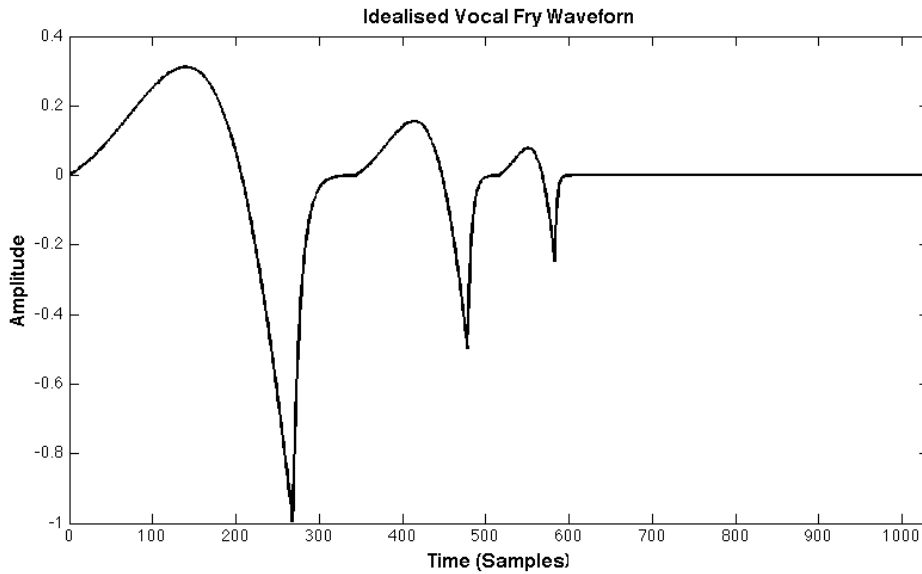


Figure 5.1 – One pitch period of an idealised vocal fry waveform displaying two secondary pulses

It is clear that the current implementation does not cover every possible voice type, so it is an intuitive extension to add to the existing software. This may involve the inclusion of a wavetable synthesis method resembling that described earlier. A stable wavetable synthesis approach would allow for any waveform to be stored and resynthesised, as opposed to only those which can be modeled using the LF-equation.

It has been acknowledged since the early days of voice source research that voice source qualities can be altered by supraglottal factors such as vowel choice and other articulations [7]. A worthwhile extension to the existing voice source model would be the inclusion of these dependent variations in voice quality. Again, this would benefit from primary research including analytical recordings of the voice source under a variety of conditions.

5.3.3 Multi-touch, Gestural User Interfaces

Whilst skeuomorphic interfaces made up of on-screen buttons and sliders, usually with a one-to-one relationship between control and parameters affected, are the most immediately intuitive to use, it has been suggested many times in HCI-related research that this approach does not directly lend itself to intuitive performance in real-time [80]. With a subject as complex as the vocal system, providing individual and independent user control of every possible parameter would not make sense in terms of modelling voice source behaviour in real time. An alternative approach would be to develop a gestural touch-screen interface that departs from the instantly recognisable skeuomorphic layout in favour of a highly cross-coupled, expressive interface such as that presented in figure 5.3. In formant synthesis, this approach has already proved effective with the HandSynth [81] software and hardware which allows a variety of vowels, pitch and amplitude to be controlled using a single touchscreen interface and a pressure-sensitive stylus (fig. 5.2). This method could conceivably be adapted to control an array of voice source features concurrently.



Figure 5.2 – HandSynth stylus operated touchscreen interface for articulatory synthesiser [image taken from [81]]

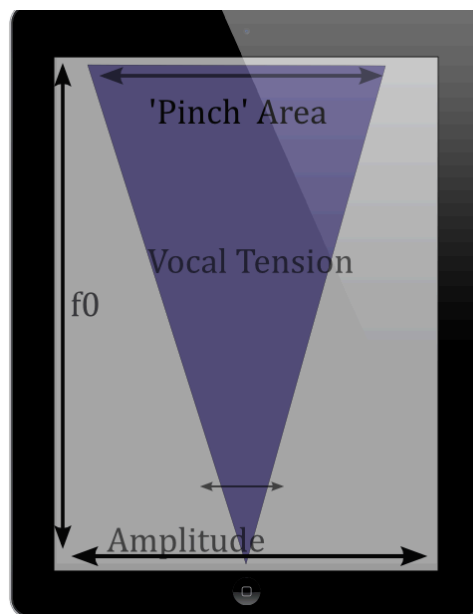


Figure 5.3 – Proposed design for multitouch interface. Position on x/y plane could control pitch and amplitude, with multitouch gestures such as ‘pinch’ (i.e. distance between two touch points) could be used to control parameters such as vocal tension

5.3.4 Implementation of Dynamic Impedance Mapping within the 2D DWM

In [30], a method for simulating vowel articulations with the 2D DWM vocal tract model is described. This allows for multiple vowels to be synthesised in software, with minimal digital distortion present. Other vocal artefacts such as diphthongs and some consonants can also be modeled using this technique. As a proof of concept, a single static vowel was modeled using the same method, however a full implementation of the dynamic impedance mapping feature would allow for multiple vowels and articulations to be used with the extend LF-model software. This would not only provide a more fully-fledged voice synthesis software application, but would also allow for further extensions to be implemented such as the vowel-dependent voice source factors mentioned in section 5.3.2.

5.4 Conclusion

Any research project conducted under a time constraint will be affected by limitations in size and scope, with the research described above being no exception. The primary goal of this project was to develop an application that could contribute to voice synthesis research, via a voice source synthesis engine that is both *expressive* and *versatile*. It is considered that the voice source model developed during this project is more expressive and versatile than other similar software. However, as discussed in section 5.3, there are many further improvements and extensions to consider.

5. Summary and Analysis

The research provides an impetus for further voice synthesis investigation, as well as a first step towards assistive technology development. The importance of the voice source in accurate synthesis of vocal effects such as breathiness and tension has been established, and a platform for further work into the perception of such effects has been developed.

Appendix A – 'LFModelFull.m' MATLAB Source Code

```

1 % Implementation of Liljencrants-Fant glottal flow model, based on
2 % Jack Mullens 'LFInput' code from 'VocalModel' synthesiser.
3 % Expanded with timing parameters taken from (Fu 2006),
4 % HPF gaussian noise for BREATHY and FALSETTO voices (Childers 1991)
5
6
7 q = 1; % QUIT on/off
8
9 while q == 1
10
11 Fs = 44100; % Sampling rate
12 overSample = 1000; % Oversampling used for more accurate waveform
    calculation
13 Ee = 1.0; % Positive peak amplitude
14
15
16 % Voice type selection:
17
18 voicetype = input('Enter a voice type (number between 1-9): ');
19
20 switch voicetype
21
22 % for TYPICAL typical voice type
23     case 1
24         f0 = input('Enter a frequency between 94-287: ');
25         period = 1/f0;
26         tc = 1.000*period;
27         te = 0.780*period;
28         tp = 0.600*period;
29         ta = 0.028*period;
30 % for MODAL voice type
31     case 2
32         f0 = input('Enter a frequency between 94-287: ');
33         period = 1/f0;
34         tc = 0.582*period;
35         te = 0.554*period;
36         tp = 0.413*period;
37         ta = 0.004*period;
38 % for VOCAL FRY(?) typical voice type
39     case 3
40         f0 = input('Enter a frequency between 24-52: ');
41         period = 1/f0;
42         tc = 0.720*period;
43         te = 0.596*period;
44         tp = 0.481*period;
45         ta = 0.027*period;
46 % for BREATHY voice type
47     case 4
48         f0 = input('Enter a frequency between 94-287: ');
49         period = 1/f0;
50         tc = 0.771*period;
51         te = 0.660*period;
52         tp = 0.462*period;
53         ta = 0.027*period;
54 % for MODAL voice type (from Fu 2006)
55     case 5
56         f0 = input('Enter a frequency between 94-287: ');
57         period = 1/f0;

```

```

58         tc = 1.0*period;
59         te = 0.575*period;
60         tp = 0.457*period;
61         ta = 0.009*period;
62 % for VOCAL FRY voice type (from Fu 2006)
63     case 6
64         f0 = input('Enter a frequency between 24-52: ');
65         period = 1/f0;
66         tc = 1.0*period;
67         te = 0.251*period;
68         tp = 0.19*period;
69         ta = 0.008*period;
70 % for BREATHY voice type (from Fu 2006)
71     case 7
72         f0 = input('Enter a frequency between 94-287: ');
73         period = 1/f0;
74         tc = 1.0*period;
75         te = 0.756*period;
76         tp = 0.529*period;
77         ta = 0.082*period;
78 % for FALSETTO voice type (from Fu 2006)
79     case 8
80         f0 = input('Enter a frequency between 287-440: ');
81         period = 1/f0;
82         tc = 1.0*period;
83         te = 0.770*period;
84         tp = 0.570*period;
85         ta = 0.133*period;
86 % for CUSTOM voice type
87     case 9
88         f0 = input('Enter a suitable frequency: ');
89         period = 1/f0;
90         TC = input('Enter a value for Tc (0-1): ');
91         TE = input('Enter a value for Te (0-1): ');
92         TP = input('Enter a value for Tp (0-1): ');
93         TA = input('Enter a value for Ta (0-1): ');
94         tc = TC*period;
95         te = TE*tc;
96         tp = TP*tp;
97         ta = TA*ta;
98 end
99
100 % Vocal Tension modulation. This is a rough calculation of the Speed
101 % Quotient (SQ) achieved by adding/subtracting a percentage of the
102 % original
103 % timing parameters
104 vocalTension = input('Enter a vocal tension value (+/- 0.3): ');
105 te = te + te*vocalTension;
106 tp = tp + tp*vocalTension;
107 ta = ta - ta*vocalTension;
108
109 vocalFoldReturnRate = input('Enter a vocal fold return rate value (-
    0.9 - 1.0): ');
110 ta = ta + (ta*vocalFoldReturnRate);
111
112 SQ = tp/(te+ta-tp); % Speed Quotient
113
114 % Oversampling
115 period = period/overSample;
116 tc = tc/overSample;
117 te = te/overSample;
118 tp = tp/overSample;
119 ta = ta/overSample;

```

```

120
121 % LF coefficient calculation
122 tn = te - tp;
123 tb = tc - te;
124
125 wg = pi/tp;
126 Eo = Ee;
127
128 areaSum=1.0;
129 peakChange=0.001;
130 optimumArea=1e-14;
131 epsilonDiff=10000.0;
132 epsilonOptimumDiff=0.1;
133
134 % solve iteratively for epsilon
135
136 epsilonTemp = 1/ta;
137
138 while abs(epsilonDiff)>epsilonOptimumDiff
139     epsilon = (1/ta)*(1-exp(-epsilonTemp*tb));
140     epsilonDiff = epsilon - epsilonTemp;
141     epsilonTemp = epsilon;
142
143     if epsilonDiff<0
144         epsilonTemp = epsilonTemp + (abs(epsilonDiff)/100);
145     end
146     if epsilonDiff>0
147         epsilonTemp = epsilonTemp - (abs(epsilonDiff)/100);
148     end
149 end
150
151 end
152
153 % iterate through area balance to get Eo and alpha to give A1 + A2 = 0
154
155 % original line in JM's code. causes MATLAB to keep running until Eo
is
156 % NaN:
157 % while (areaSum<-optimumArea)|| (areaSum>optimumArea)
158
159 % use this line for MATLAB
160 while (areaSum > optimumArea)
161     alpha = real(( log(-Ee/(Eo*sin(wg*te))))/te);
162
163     Area1 = ( Eo*exp(alpha*te)/(sqrt(alpha*alpha+wg*wg)))...
164         * (sin(wg*te-atan(wg/alpha)))...
165         + (Eo*wg/(alpha*alpha+wg*wg));
166
167     Area2 = ( -(Ee)/(epsilon*epsilon*(ta)))...
168         * (1 - exp(-epsilon*tb*(1+epsilon*tb)));
169
170     areaSum = Area1 + Area2;
171
172     if areaSum>0.0
173         Eo = Eo - 1e5*areaSum;
174     elseif areaSum<0.0
175         Eo = Eo + 1e5*areaSum;
176     end
177
178 end
179
180 % Calculate length of one pitch period in samples (dataLength)
181 dataLength = floor(overSample*Fs*period);
182 output = zeros(1,dataLength); % Output waveform for one pitch period

```

```

183
184
185 % Noise parameters
186 noiseAmt = input('Enter a turbulent noise amount (0-1): ');
187 if noiseAmt > 0.0;
188     noiseDuration = input('Enter a turbulent noise duration: ');
189     noiseStart = input('Enter a turbulent noise start position: ');
190     noiseDuration = noiseDuration*dataLength;
191     noiseStart = noiseStart*dataLength;
192
193
194 % Calculate filter coefficients for HPF @ 2 kHz
195 %     Fc = 2000; % Filter cut-off freq.
196 %     Wn = Fc/(Fs/2); % Normalised Fc between 0-1 where 1 corresponds
    to Nyquist
197 %     [b, a] = butter(2, Wn, 'high'); % Computes b and a coefficients
    for 2nd order butterworth HPF
198 %     b0 = b(1);
199 %     b1 = b(2);
200 %     b2 = b(3);
201 %     a1 = a(2);
202 %     a2 = a(3);
203 end
204
205 % playback section
206 durationSeconds = input('Enter a duration (s): ');
207 durationSamples = durationSeconds*Fs;
208 waveform = zeros(1,durationSamples);
209 j = 1; % counter
210
211 % checkNoise = zeros(1,durationSamples);
212
213 for i = 1:durationSamples
214
215     t = j*period/dataLength;
216
217 %     This is where the LF waveform is calculated
218     if t<te
219         LFSample = Eo*(exp(alpha*t)) * sin(wg*t);
220     end
221     if t>=te
222         LFSample = -((Ee)/(epsilon*t))*(exp(-epsilon*(t-te))...
223             - exp(-epsilon*(tc-te)));
224     end
225     if t>tc
226         LFSample = 0.0;
227     end
228
229     LFSampleQ = trapz(LFSample);
230
231     if noiseAmt == 0.0;
232         waveform(i) = LFSample;
233         waveformQ(i) = LFSampleQ;
234     end
235
236 %     Add noise
237     if noiseAmt > 0.0;
238         noiseSample = rand(1,1)*noiseAmt;
239
240 % Could HPF noise here if found necessary
241
242 % This adds white noise to LF-waveform at noiseStart for
    noiseDuration.
243 % need to cycle around to the start of pitch period if noiseStart +

```

```

244 % noiseDuration is bigger than length of period (dataLength):
245
246     if noiseStart + noiseDuration < dataLength;
247         if j >= noiseStart && j <=noiseStart+noiseDuration
248             waveform(i) = LFSample + noiseSample;
249         end
250         if j < noiseStart || j > noiseStart+noiseDuration
251             waveform(i) = LFSample;
252         end
253     end
254     if noiseStart + noiseDuration > dataLength;
255         remainder = (noiseStart+noiseDuration)-dataLength;
256         if j <= remainder || j >= noiseStart
257             waveform(i) = LFSample + noiseSample;
258         end
259         if j > remainder && j < noiseStart
260             waveform(i) = LFSample;
261         end
262     end
263 end
264
265 % Increment counter and cycle back to start of waveform if counter
266 % > dataLength
267
268     j=j+1;
269
270     if j > dataLength
271         j = j - dataLength;
272     end
273
274 end
275
276 plot(waveform);
277
278 p = input('press 1 to play sound: ');
279
280 if p == 1
281     sound(waveform,Fs)
282 end
283
284 s = input('press 1 to save sound: ');
285
286 if s == 1
287     filename = input('Enter .wav filename: ');
288     audiowrite(filename, waveform, Fs);
289 end
290
291
292 q = input('press 1 to start or 0 to quit: ');
293
294 hold on
295
296 end

```

Appendix B – ‘ViewController.h’ LFGGen App Header File

```
1 //
2 //  ViewController.h
3 //  LFGGen
4 //
5 //  Created by Jacob Harrison on 11/12/2013.
6 //  Copyright (c) 2013 Jacob Harrison. All rights reserved.
7 //
8
9 #import <UIKit/UIKit.h>
10 #import "AudioEngine.h"
11
12 @interface ViewController : UIViewController{
13     AudioEngine *_ae;
14 }
15
16 - (IBAction)startStop:(UIButton *)sender;
17
18 - (IBAction)setF0:(UISlider *)sender;
19
20 - (IBAction)setTension:(UISlider *)sender;
21
22 - (IBAction)setTa:(UISlider *)sender;
23
24 - (IBAction)setAmplitude:(UISlider *)sender;
25
26 @end
```


Appendix C – ‘ViewController.m’ LFGGen App Main File

```

1 //
2 //  ViewController.m
3 //  LFGGen
4 //
5 //  Created by Jacob Harrison on 11/12/2013.
6 //  Copyright (c) 2013 Jacob Harrison. All rights reserved.
7 //
8
9 #import "ViewController.h"
10
11 @interface ViewController ()
12
13 @end
14
15 @implementation ViewController
16
17 - (void)viewDidLoad
18 {
19     [super viewDidLoad];
20     // Do any additional setup after loading the view, typically from
    a nib.
21     _ae=[[AudioEngine alloc] init];
22
23 }
24
25 - (void)didReceiveMemoryWarning
26 {
27     [super didReceiveMemoryWarning];
28     // Dispose of any resources that can be recreated.
29 }
30
31
32 //Start and Stop audio playback
33 - (IBAction)startStop:(UIButton *)sender {
34
35     if([_ae isPlaying]){
36         [_ae stopPlayback];
37     }
38     else{
39         [_ae startPlayback];
40     }
41
42 }
43
44
45
46
47 - (IBAction)setTa:(UISlider *)sender {
48     [_ae setTa:[sender value]];
49 }
50
51 - (IBAction)setF0:(UISlider *)sender {
52     [_ae setF0:[sender value]];
53 }

```

```
54
55 - (IBAction)setTension:(UISlider *)sender {
56     [_ae setTension:[sender value]];
57 }
58
59 - (IBAction)setAmplitude:(UISlider *)sender{
60     [_ae setAmplitude:[sender value]];
61 }
62
63 - (IBAction)setTypical:(UIButton *)sender {
64     [_ae setTypical];
65 }
66
67 - (IBAction)setModal:(UIButton *)sender {
68     [_ae setModal];
69 }
70
71 - (IBAction)setBreathy:(UIButton *)sender {
72     [_ae setBreathy];
73 }
74
75 - (IBAction)setVocalFry:(UIButton *)sender {
76     [_ae setVocalFry];
77 }
78
79 - (IBAction)setFalsetto:(UIButton *)sender {
80     [_ae setFalsetto];
81 }
82
83 - (IBAction)vowelOn:(UIButton *)sender {
84     [_ae vowelOn];
85 }
86
87 - (IBAction)pitchSlide:(UIButton *)sender {
88     [_ae pitchSlide];
89 }
90
91 - (IBAction)autoVoice:(UIButton *)sender {
92     [_ae autoVoice];
93 }
94
95
96
97 @end
```

Appendix D – ‘AudioEngine.h’ LFGGen App Header File

```

1 //
2 // AudioEngine.h
3 // LFGGen
4 //
5 // Created by Jacob Harrison on 11/12/2013.
6 // Credit to Dimitrios Zantalis for initial implementation of
   EffectState structure
7 // Copyright (c) 2013 Jacob Harrison. All rights reserved.
8 //
9
10 #import <Foundation/Foundation.h>
11 #import <AudioToolbox/AudioToolbox.h>
12 #import <AVFoundation/AVFoundation.h>
13
14 #define MY_PI 3.14159265359
15
16 // in order to declare arrays in the EffectState structure, the sizes
   need to be defined first (there is probably a way of declaring a
   variable size array here).
17 // for the pressure and impedance arrays to be made available in the
   processing loop, they will be declared here, meaning that the
   'sizeXMax' and 'sizeYMax' variables will also be fixed here
18
19 #define SIZE_X_MAX 15
20 #define SIZE_Y_MAX 4
21
22 typedef struct{
23     Float64    _hardwareSampleRate;
24     AudioUnit  _rioAU;
25     AudioStreamBasicDescription _clientASBD;
26
27     // coefficients for LF waveform calculation
28     double _f0;
29     double _period;
30     int _dataLength;
31     double _Eo;
32     double _Ee;
33     double _alpha;
34     double _wg;
35     double _epsilon;
36     double _taVal;
37     double _tcVal;
38     double _tpVal;
39     double _teVal;
40     double _ta;
41     double _tc;
42     double _tp;
43     double _te;
44     int _k;
45     int _kk;
46     // noise coefficients
47     double _noiseAmount;
48     double _noiseDuration;
49     double _noiseStart;
50     BOOL _noiseOn;
51 //     double _noiseFilter[490];
52
53     double _vocalTension;
54     // vibrato coefficients

```

```

55     double _vibratoFreq;
56     double _vibratoDepth;
57     double _vibratoOut;
58     int _WTSize;
59     BOOL _vibratoOn;
60     double _dt;
61
62 //     pressure and impedance arrays for 2D DWM
63     Float32 _pNPlus[SIZE_Y_MAX*SIZE_X_MAX];
64     Float32 _pNMinus[SIZE_Y_MAX*SIZE_X_MAX];
65     Float32 _pEPlus[SIZE_Y_MAX*SIZE_X_MAX];
66     Float32 _pEMinus[SIZE_Y_MAX*SIZE_X_MAX];
67     Float32 _pSPlus[SIZE_Y_MAX*SIZE_X_MAX];
68     Float32 _pSMinus[SIZE_Y_MAX*SIZE_X_MAX];
69     Float32 _pWPlus[SIZE_Y_MAX*SIZE_X_MAX];
70     Float32 _pWMinus[SIZE_Y_MAX*SIZE_X_MAX];
71     // Impedance
72     Float32 _zNorth[SIZE_Y_MAX*SIZE_X_MAX];
73     Float32 _zEast[SIZE_Y_MAX*SIZE_X_MAX];
74     Float32 _zSouth[SIZE_Y_MAX*SIZE_X_MAX];
75     Float32 _zWest[SIZE_Y_MAX*SIZE_X_MAX];
76
77     double _impData[SIZE_X_MAX];
78
79     double _reflectionGLottis;
80     double _reflectionLips;
81     double _reflectionWalls;
82
83
84     int _sizeXMax;
85     int _sizeYMax;
86
87     BOOL _vowelOn;
88
89     double _amp;
90
91     BOOL _pitchSlide;
92
93     int _count;
94     int _sampleCount;
95     int _f0Count;
96
97     BOOL _autoVoice;
98
99     int _voiceType;
100
101
102
103 }EffectState;
104
105 @interface AudioEngine : NSObject{
106     AVAudioSession *_AudioSession;
107     BOOL _playing;
108     BOOL _recording;
109
110     //USING FS
111     Float64 _samplingRate;
112
113     NSFileManager *_FileManager;
114     NSURL *_audioDirURL;
115 }
116
117 @property (assign) EffectState effectState;
118

```

```
119 //Public interface
120 -(OSStatus)startPlayback;
121 -(OSStatus)stopPlayback;
122 //-(OSStatus)exportAudioWithName:(NSString*)fileName;
123 -(BOOL)isPlaying;
124
125 -(void)setF0:(Float32)f0Value;
126
127 -(void)setTa:(Float32)taValue;
128
129 -(void)setModal;
130 -(void)setVocalFry;
131 -(void)setBreathy;
132 -(void)setTypical;
133 -(void)setFalsetto;
134 -(void)setAmplitude:(Float32)ampValue;
135 -(void)vowelOn;
136 -(void)pitchSlide;
137 -(void)autoVoice;
138
139 -(void)setVoiceType;
140
141
142 -(void)setTension:(Float32)tensionValue;
143
144 //-(void)makeNoise;
145
146 - (void) checkErr: (OSStatus) error
147     withMessage:(const char *) message;
148
149 @end
```

Appendix E – ‘AudioEngine.m’ LFGGen App Main File

```

1
2 //
3 // AudioEngine.m
4 // 'LFGGen' Liljencrants-Fant (LF) glottal source modelling App.
5 // This app creates an acoustic model of the human voice source via
   the LF Equation
6 // A 2D Digital Waveguide Model of the vocal tract based on Jack
   Mullen's VocalModel
7 // software is used to synthesis vowel formants
8 //
9 // 2D DWM port based on Amelia Gully's MATLAB implementation
10 //
11 // Generic Audio App template provided by Dimitrios Zantalis
12 // (credited in code where appropriate)
13 //
14 // Created by Jacob Harrison on 11/12/2013.
15 // Copyright (c) 2013 Jacob Harrison. All rights reserved.
16 //
17
18 #import "AudioEngine.h"
19
20 // Main Processing Loop
21
22 static OSStatus playbackCallback(void *inRefCon,
23                                 AudioUnitRenderActionFlags
24                                 *ioActionFlags,
25                                 const AudioTimeStamp *inTimeStamp,
26                                 UInt32 inBusNumber,
27                                 UInt32 inNumberFrames,
28                                 AudioBufferList *ioData
29                                 ) {
30     // Get the tone parameters out of the view controller
31     EffectState *fxs=(EffectState*)inRefCon;
32     Float32 output;
33
34     // Get LF waveform coefficients
35     double t;
36     double LFcurrentSample;
37     double LFcurrentSample1;
38     double period = fxs->_period;
39     int dataLength = fxs->_dataLength;
40     double te = fxs->_te;
41     double Eo = fxs->_Eo;
42     double alpha = fxs->_alpha;
43     double wg = fxs->_wg;
44     double Ee = fxs->_Ee;
45     double epsilon = fxs->_epsilon;
46     double ta = fxs->_ta;
47     double tc = fxs->_tc;
48     int k = fxs->_k;
49     int kk = fxs->_kk;
50
51     // Turbulent noise coefficients
52     BOOL noiseOn = fxs->_noiseOn;
53     double noiseAmount = fxs->_noiseAmount;
54     double noiseDuration = fxs->_noiseDuration*dataLength;
55     double noiseStart = fxs->_noiseStart*dataLength;
56     // double *noiseFilter = fxs->_noiseFilter;

```

```

57     double noiseRemainder;
58     double noiseAdd;
59     double noiseSample;
60
61     // Pressure arrays
62     Float32 *pNPlus = fxs->_pNPlus;
63     Float32 *pNMinus = fxs->_pNMinus;
64     Float32 *pEPlus = fxs->_pEPlus;
65     Float32 *pEMinus = fxs->_pEMinus;
66     Float32 *pSPlus = fxs->_pSPlus;
67     Float32 *pSMinus = fxs->_pSMinus;
68     Float32 *pWPlus = fxs->_pWPlus;
69     Float32 *pWMinus = fxs->_pWMinus;
70
71     // Pressure arrays
72     Float32 *zNorth = fxs->_zNorth;
73     Float32 *zEast = fxs->_zEast;
74     Float32 *zSouth = fxs->_zSouth;
75     Float32 *zWest = fxs->_zWest;
76
77
78     // 2D DWM coefficients
79     double *impData = fxs->_impData;
80     int sizeXMax = fxs->_sizeXMax;
81     int sizeYMax = fxs->_sizeYMax;
82     double reflectionLips = fxs->_reflectionLips;
83     double reflectionGlottis = fxs->_reflectionGLottis;
84     double reflectionWalls = fxs->_reflectionWalls;
85
86     BOOL vowelOn = fxs->_vowelOn;
87
88     double pJ;
89     double opPressure;
90
91     int y,x;
92
93     double amp = fxs->_amp;
94
95     BOOL pitchSlide = fxs->_pitchSlide;
96
97 //     Set max and min dataLength (pitch period in samples) values for
98 //     automatic f0.
99 //     int dataLengthMax = 490; //490 samples = pitch period 1/90 s
100 //     (90 Hz)
101 //     int dataLengthMax = 1837; //1837 samples = 24 Hz
102 //     int dataLengthMin = 232; //232 samples = pitch period 1/190 s
103 //     (190 Hz)
104 //     int dataLengthMin = 100;
105
106     int count = fxs->_count;
107
108     double f0Data[] = {86.7656730000000,
109                       86.4178010000000,
110                       90.9541950000000,
111                       87.9545930000000,
112                       88.6849350000000,
113                       89.0769560000000,
114                       88.8090830000000,
115                       89.0420040000000,
116                       89.2941220000000,
117                       89.8216050000000,
118                       89.1064770000000,
119                       88.1318270000000,
120                       87.5954940000000,

```

118	87.6932790000000,
119	87.3959820000000,
120	87.4183820000000,
121	88.0824290000000,
122	88.3171960000000,
123	88.5990350000000,
124	88.4426470000000,
125	89.4475360000000,
126	89.8663800000000,
127	90.2665200000000,
128	90.7101400000000,
129	90.9507130000000,
130	90.9308580000000,
131	90.9280150000000,
132	91.2230760000000,
133	91.7717330000000,
134	92.1247940000000,
135	92.2978610000000,
136	92.6453840000000,
137	93.4096030000000,
138	94.1552810000000,
139	94.9410550000000,
140	95.6436490000000,
141	96.6370420000000,
142	97.3720470000000,
143	97.9702550000000,
144	98.9070940000000,
145	99.5896930000000,
146	100.5126750000000,
147	102.1350560000000,
148	103.5951690000000,
149	104.8524450000000,
150	106.0729660000000,
151	107.3970000000000,
152	109.1361660000000,
153	110.1464010000000,
154	111.2791390000000,
155	112.6315980000000,
156	114.1560860000000,
157	115.5265370000000,
158	117.0730300000000,
159	117.8940420000000,
160	118.3342150000000,
161	118.2619410000000,
162	117.1698500000000,
163	115.6659680000000,
164	113.8116160000000,
165	111.9607550000000,
166	110.0236370000000,
167	107.2819440000000,
168	104.2106370000000,
169	100.9198230000000,
170	97.4242660000000,
171	95.2202820000000,
172	93.8211850000000,
173	91.8860980000000,
174	90.6061610000000,
175	89.8349510000000,
176	88.9622060000000,
177	88.1375650000000,
178	87.7454650000000,
179	87.5657920000000,
180	87.1366340000000,
181	87.1113610000000,


```

182             86.9435590000000,
183             86.7299700000000,
184             86.3627780000000,
185             85.9939510000000,
186             85.6657360000000,
187             84.7229040000000,
188             85.2134430000000,
189             84.7120240000000,
190             84.4130280000000,
191             84.3421640000000,
192             84.5041810000000,
193             84.2591300000000,
194             83.7980820000000,
195             83.2793240000000,
196             82.9630780000000,
197             82.6915980000000,
198             82.0845250000000,
199             80.9131000000000,
200             80.6771400000000,
201             79.3941160000000,
202             77.4358050000000,
203             75.9000770000000,
204             76.3403430000000}; // f0 data from 1s pitch slide obtained
from PRAAT
205     int f0sampleRate = 100; // Sample rate at which f0 data is
recorded
206 //     int f0samples = sizeof(f0Data);
207     int f0Update = fxs->_hardwareSampleRate / f0sampleRate; //
interval at which f0 is updated
208     int sampleCount = fxs->_sampleCount;
209     int f0Count = fxs->_f0Count;
210     int Fs = fxs->_hardwareSampleRate;
211     int overSample = 1000;
212
213
214
215
216 for(UINT32 i = 0; i < ioData->mNumberBuffers; i++) {
217
218     for(int j = 0; j < inNumberFrames; j++) {
219
220         // If automatic f0 is not selected:
221         if (pitchSlide == FALSE) {
222
223             // 'k' is sample increment counter. wrap around to 0
if k > dataLength
224                 if (k>dataLength) {
225                     k = k - dataLength;
226                 }
227
228
229                 // Main LF-waveform calculation performed here:
230                 t = (double)k*period/(double)dataLength;
231
232                 if (t<te) {
233                     LFcurrentSample = Eo*(exp(alpha*t)) * sin(wg*t);
234                 }
235                 if (t>=te) {
236                     LFcurrentSample = -((Ee)/(epsilon*ta))*(exp(-
epsilon*(t-te)) - exp(-epsilon*(tc-te)));
237                 }
238                 if (t>tc) {
239                     LFcurrentSample = 0.0;
240                 }

```

```

241
242 // If breathy or falsetto is selected, add noise
243 if (noiseOn == TRUE) {
244     // Creates white noise between -1 and +1
245     noiseSample = rand() % 200;
246     noiseSample = noiseSample - 100;
247     noiseSample = noiseSample/100;
248     // Attenuate noise signal by noise amount
selected
249     noiseAdd = noiseSample*noiseAmount;
250     // Turbulent noise portion of waveform begins at
the closing phase and ends during the opening phase (i.e. crosses
over two pitch cycles)
251     // Need to 'wrap' noise around so that the
opening phase portion of noise begins at the start of the pitch
cycle
252     // Calculate remainder (portion of noise that
extends beyond pitch period):
253     noiseRemainder = (noiseStart + noiseDuration) -
dataLength;
254
255     // if start time + duration is less than length
of period (i.e no remainder)
256     if (noiseStart + noiseDuration < dataLength) {
257         if (k >= noiseStart + noiseDuration){
258             LFcurrentSample1 = LFcurrentSample;
259         }
260         if (k < noiseStart) {
261             LFcurrentSample1 = LFcurrentSample;
262         }
263         if (k > noiseStart & k <= noiseStart +
noiseDuration) {
264             LFcurrentSample1 =
LFcurrentSample+noiseAdd;
265         }
266     }
267
268     // if start time + duration is greater than
length of period (i.e noise wraps round)
269     if (noiseStart + noiseDuration >= dataLength){
270         if (k > noiseRemainder && k < noiseStart) {
271             LFcurrentSample1 = LFcurrentSample;
272         }
273
274         if (k <= noiseRemainder || k >= noiseStart) {
275             LFcurrentSample1 =
LFcurrentSample+noiseAdd;
276         }
277     }
278 }
279
280 // If no noise is needed, the current sample remains
the same.
281 if (noiseOn == FALSE) {
282     LFcurrentSample1 = LFcurrentSample;
283 }
284
285
286 // This section performs the calculations for
pressure and impedance arrays in 2D DWM
287 // Unlike MATLAB implementation, 1D arrays are used
to store the 2D information
288 // MATLAB version: pNPlus[y][x] is equivalent to iOS
version: pNPlus[y*sizeXMax + x]

```

```

289         if (vowelOn == TRUE) {
290             for (y = 0; y<sizeYMax; y++) {
291                 for (x = 0; x<sizeXMax; x++) {
292                     pJ = 2 * ((pNPlus[y*sizeXMax + x] /
zNorth[y*sizeXMax + x])
293                             + (pEPlus[y*sizeXMax + x] /
zEast[y*sizeXMax + x])
294                             + (pSPlus[y*sizeXMax + x] /
zSouth[y*sizeXMax + x])
295                             + (pWPlus[y*sizeXMax + x] /
zWest[y*sizeXMax + x]))
296                     / ( (1 / zNorth[y*sizeXMax + x]) + (1 /
zEast[y*sizeXMax + x])
297                         + (1 / zSouth[y*sizeXMax + x]) + (1 /
zWest[y*sizeXMax + x]));
298
299                     // calculate outgoing pressures from
junction
300                     pNMinus[y*sizeXMax + x] = pJ -
pNPlus[y*sizeXMax + x];
301                     pEMinus[y*sizeXMax + x] = pJ -
pEPlus[y*sizeXMax + x];
302                     pSMinus[y*sizeXMax + x] = pJ -
pSPlus[y*sizeXMax + x];
303                     pWMinus[y*sizeXMax + x] = pJ -
pWPlus[y*sizeXMax + x];
304
305                 }
306             }
307
308             for (y = 0; y<sizeYMax; y++) {
309                 for (x = 0; x<sizeXMax; x++) {
310                     // if x = 0 then we are at the glottis
end, need to take input signal (LFcurrentSample1)
311                     if (x == 0) {
312                         pWPlus[y*sizeXMax + x] =
reflectionGlottis * pWMinus[y*sizeXMax + x] + LFcurrentSample1;
313                     }
314                     else {
315                         pWPlus[y*sizeXMax + x] =
pEMinus[y*sizeXMax + x-1];
316                     }
317
318                     // update eastgoing pressures (towards
lips)
319                     // if x = sizeXMax we are at the lip end
320
321                     if (x == sizeXMax-1) {
322                         pEPlus[y*sizeXMax + x] =
reflectionLips * pEMinus[y*sizeXMax + x];
323                     }
324                     else {
325                         pEPlus[y*sizeXMax + x] =
pWMinus[y*sizeXMax + x+1];
326                     }
327
328                     // if y = 1 we are at the bottom wall
329                     if (y == 0) {
330                         pSPlus[y*sizeXMax + x] =
reflectionWalls * pSMinus[y*sizeXMax + x];
331                     }
332                     else {
333                         pSPlus[y*sizeXMax + x] = pNMinus[(y-
1)*sizeXMax + x];

```

```

334         }
335
336         // if y = sizeYMax -1 we are at the top
wall
337         if (y == sizeYMax - 1){
338             pNPlus[y*sizeXMax + x] =
reflectionWalls * pNMinus[y*sizeXMax + x];
339         }
340         else {
341             pNPlus[y*sizeXMax + x] =
pSMinus[(y+1)*sizeXMax + x];
342         }
343         // printf("%f %f %f \n",
reflectionGlottis, reflectionLips, reflectionWalls);
344     }
345 }
346
347     // initialise current sample output pressure to
zero
348     opPressure = 0;
349
350     for (y = 0; y<sizeYMax; y++) {
351         opPressure = opPressure + (1-
reflectionLips)*pEPlus[y*sizeXMax + sizeXMax-1];
352     }
353
354     // multiply by 125 to normalise to non-vowel
amplitude (value obtained through trial and error - not sure why DWM
reduces amplitude by so much)
355     output = 125 * (opPressure/impData[sizeXMax -
1]);
356     }
357
358     // if no vowel is used
359     if (vowelOn == FALSE) {
360         output = LFcurrentSample1;
361     }
362
363     output = output*0.95*amp;
364
365     // check for clipping
366     if (output > 1.0) {
367 //         printf("output clipped at %f \n", output);
368         output = 1.0;
369     }
370
371     k += 1;
372
373 //     printf("%d %d %d %f %d %f\n", dataLength,
sampleCount, f0Count, period, k, output);
374
375     memcpy(ioData->mBuffers[i].mData+j*fxs-
>_clientASBD.mBytesPerFrame, &output, sizeof(Float32));
376
377     }
378
379     if (pitchSlide == TRUE) {
380         // This is the same as previous section but with
automatic f0 input
381
382         if (kk>dataLength) {
383             kk = 0;
384         }
385

```

```

386         if (sampleCount == f0Update) {
387             sampleCount = 0;
388         }
389
390
391         // Update new value for dataLength based on natural
f0 data
392         if (sampleCount == 0) {
393             period = (1/f0Data[f0Count])/overSample;
394
395             f0Count += 1;
396
397             if (f0Count == 100) {
398                 f0Count = 0;
399             }
400         }
401
402         if (kk == 0) {
403             dataLength = floor(Fs*period*overSample);
404         }
405
406         t = (double)kk*period/(double)dataLength;
407
408         if (t<te) {
409             LFcurrentSample = Eo*(exp(alpha*t)) * sin(wg*t);
410         }
411         if (t>=te) {
412             LFcurrentSample = -((Ee)/(epsilon*ta))*(exp(-
epsilon*(t-te)) - exp(-epsilon*(tc-te)));
413         }
414         if (t>tc) {
415             LFcurrentSample = 0.0;
416         }
417
418         if (noiseOn == TRUE) {
419             noiseSample = rand() % 200;
420             noiseSample = noiseSample - 100;
421             noiseSample = noiseSample/100;
422             noiseAdd = noiseSample*noiseAmount;
423             noiseRemainder = (noiseStart + noiseDuration) -
dataLength;
424
425             if (noiseStart + noiseDuration < dataLength) {
426                 if (kk >= noiseStart + noiseDuration){
427                     LFcurrentSample1 = LFcurrentSample;
428                 }
429                 if (kk < noiseStart) {
430                     LFcurrentSample1 = LFcurrentSample;
431                 }
432                 if (kk > noiseStart & kk <= noiseStart +
noiseDuration) {
433                     LFcurrentSample1 =
LFcurrentSample+noiseAdd;
434                 }
435             }
436
437             if (noiseStart + noiseDuration >= dataLength){
438                 if (kk > noiseRemainder && kk < noiseStart) {
439                     LFcurrentSample1 = LFcurrentSample;
440                 }
441
442                 if (kk <= noiseRemainder || kk >= noiseStart)
{
443                     LFcurrentSample1 =

```

```

LFcurrentSample+noiseAdd;
444         }
445     }
446 }
447
448     if (noiseOn == FALSE) {
449         LFcurrentSample1 = LFcurrentSample;
450     }
451
452
453     if (vowelOn == TRUE) {
454         for (y = 0; y<sizeYMax; y++) {
455             for (x = 0; x<sizeXMax; x++) {
456                 pJ = 2 * ((pNPlus[y*sizeXMax + x] /
zNorth[y*sizeXMax + x])
457                     + (pEPlus[y*sizeXMax + x] /
zEast[y*sizeXMax + x])
458                     + (pSPlus[y*sizeXMax + x] /
zSouth[y*sizeXMax + x])
459                     + (pWPlus[y*sizeXMax + x] /
zWest[y*sizeXMax + x]))
460                 / ( (1 / zNorth[y*sizeXMax + x]) + (1 /
zEast[y*sizeXMax + x])
461                   + (1 / zSouth[y*sizeXMax + x]) + (1 /
zWest[y*sizeXMax + x]));
462
463                 pNMinus[y*sizeXMax + x] = pJ -
pNPlus[y*sizeXMax + x];
464                 pEMinus[y*sizeXMax + x] = pJ -
pEPlus[y*sizeXMax + x];
465                 pSMinus[y*sizeXMax + x] = pJ -
pSPlus[y*sizeXMax + x];
466                 pWMinus[y*sizeXMax + x] = pJ -
pWPlus[y*sizeXMax + x];
467
468             }
469         }
470
471         for (y = 0; y<sizeYMax; y++) {
472             for (x = 0; x<sizeXMax; x++) {
473
474                 if (x == 0) {
475                     pWPlus[y*sizeXMax + x] =
reflectionGlottis * pWMinus[y*sizeXMax + x] + LFcurrentSample1;
476                 }
477                 else {
478                     pWPlus[y*sizeXMax + x] =
pEMinus[y*sizeXMax + x-1];
479                 }
480
481                 if (x == sizeXMax-1) {
482                     pEPlus[y*sizeXMax + x] =
reflectionLips * pEMinus[y*sizeXMax + x];
483                 }
484                 else {
485                     pEPlus[y*sizeXMax + x] =
pWMinus[y*sizeXMax + x+1];
486                 }
487
488                 if (y == 0) {
489                     pSPlus[y*sizeXMax + x] =
reflectionWalls * pSMinus[y*sizeXMax + x];
490                 }
491                 else {

```

```

492         pSPlus[y*sizeXMax + x] = pNMinus[(y-
1)*sizeXMax + x];
493     }
494
495     if (y == sizeYMax - 1){
496         pNPlus[y*sizeXMax + x] =
reflectionWalls * pNMinus[y*sizeXMax + x];
497     }
498     else {
499         pNPlus[y*sizeXMax + x] =
pSMinus[(y+1)*sizeXMax + x];
500     }
501     }
502 }
503
504     opPressure = 0;
505
506     for (y = 0; y<sizeYMax; y++) {
507         opPressure = opPressure + (1-
reflectionLips)*pEPlus[y*sizeXMax + sizeXMax-1];
508     }
509
510     output = 125 * (opPressure/impData[sizeXMax -
1]);
511 }
512
513     if (vowelOn == FALSE) {
514         output = LfcurrentSample1;
515     }
516
517     output = output*0.95*amp;
518
519     if (output > 1.0) {
520 //         printf("output clipped at %f \n", output);
521         output = 1.0;
522     }
523
524     kk += 1;
525     sampleCount += 1;
526
527
528 //         printf("%d %d %d %f %d %f\n", dataLength,
sampleCount, f0Count, period, kk, output);
529
530         memcpy(ioData->mBuffers[i].mData+j*fxs-
>_clientASBD.mBytesPerFrame, &output, sizeof(Float32));
531
532     }
533 }
534 }
535
536 // Assign values for counter and dataLength for next sample
537 fxs->_k = k;
538 fxs->_kk = kk;
539 fxs->_count = count;
540 fxs->_dataLength = dataLength;
541 fxs->_f0Count = f0Count;
542 fxs->_sampleCount = sampleCount;
543 fxs->_period = period;
544
545
546
547 return noErr;
548 }

```

```

549
550 // Implementation of AudioEngine by Dimitrios Zantalis:
551
552 @implementation AudioEngine
553
554 @synthesise effectState;
555
556 //-----
557 //      init function
558 //-----
559 -(id)init{
560     if(!(self=[super init])) return self;
561
562     OSStatus err=0;
563
564     //=====
565     //STEP 1 - CONFIGURE AND INITIALISE AUDIO SESSION
566     //=====
567     err=[self initialiseAudioSession];
568     [self checkErr:err withMessage:"Failed to configure and
initialise Audio Session!"];
569
570     //=====
571     //STEP 2 - CONFIGURE CLIENT AUDIO STREAM BASIC DESCRIPTION
572     //=====
573     [self configureASBD];
574
575     //=====
576     //STEP 3 - SETUP THE REMOTEIO AUDIO UNIT FOR RECORDING AND
PLAYBACK
577     //=====
578     err=[self setupRemoteIO];
579     [self checkErr:err withMessage:"Failed to setup RemoteIO AU!"];
580
581
582     //=====
583     //STEP 4 - Initialise Wave table
584     //=====
585     [self initValues];
586     [self createLFInput];
587     [self createDWM];
588
589     //=====
590     //STEP 5 - Initialise File Manager
591     //=====
592     //[self initFileManager];
593
594     //=====
595     //STEP 6 - INITIALISE STATE
596     //=====
597     _playing=NO;
598     _recording=NO;
599
600     if(err){
601         [self checkErr:err withMessage:"Failed to initialise
SimpleRecorderDemo!"];
602     }
603     else{
604         fprintf(stdout,"Simple Recorder initialised!\n");
605     }
606     return self;
607 }
608
609 //-----

```



```

610 //      initialiseAudioSession function
611 //-----
612 -(OSStatus)initialiseAudioSession{
613     NSError *err=nil;
614     BOOL success=FALSE;
615
616     fprintf (stdout,"Configuring Audio Session...");
617
618     //Implicit initialisation of audio session.
619     _AudioSession=[AVAudioSession sharedInstance];
620
621     success=[_AudioSession
setCategory:AVAudioSessionCategoryPlayAndRecord
622                                     error:&err];
623     if(!success){
624         [self checkErr:[err code] withMessage:"Failed to set category
for AVAudioSession"];
625     }
626
627     //Check for input availability
628     BOOL hasInput=[_AudioSession isInputAvailable];
629     if(!hasInput){
630         UIAlertView *alert=[[UIAlertView alloc] initWithTitle:@"No
audio input available"
631
632                                     message:@"The
application cannot record because no audio input has been detected!"
633                                     delegate:nil
634                                     cancelButtonTitle:@"OK"
635                                     otherButtonTitles:nil];
636         [alert show];
637     }
638     //Get hardware sample rate
639     effectState._hardwareSampleRate=[_AudioSession sampleRate];
640
641     //Activate audio session
642     success=[_AudioSession setActive:YES
643                                     error:&err];
644     if(!success){
645         [self checkErr:[err code] withMessage:"Failed to activate
audio session"];
646     }
647
648     if(!err)
649         fprintf(stdout,"OK\n");
650
651     return [err code];
652 }
653
654 //-----
655 //      configureASBD function
656 //-----
657 -(void)configureASBD{
658
659     fprintf(stdout,"Configuring client stream format...");
660     //Initialise ASBD structure
661     memset (&effectState._clientASBD, 0, sizeof
(effectState._clientASBD));
662
663     //Set up ASBD for stereo playback
664     effectState._clientASBD.mFormatID = kAudioFormatLinearPCM;
665     effectState._clientASBD.mFormatFlags
=kAudioFormatFlagsNativeEndian|kAudioFormatFlagIsFloat|kAudioFormatF
lagIsNonInterleaved;

```

```

666     effectState._clientASBD.mSampleRate =
effectState._hardwareSampleRate;
667     effectState._clientASBD.mChannelsPerFrame = 1;
668     effectState._clientASBD.mBitsPerChannel = 32;
669     effectState._clientASBD.mBytesPerPacket = 4;
670     effectState._clientASBD.mFramesPerPacket = 1;
671     effectState._clientASBD.mBytesPerFrame = 4;
672     effectState._clientASBD.mReserved=0;
673
674     fprintf(stdout,"OK\n");
675
676 }
677
678 //-----
679 //     setupRemoteIO function
680 //-----
681 -(OSStatus)setupRemoteIO{
682     OSStatus err=0;
683     UInt32 propsize=0;
684
685     fprintf(stdout,"Configuring RemoteIO AU...");
686     //.....
687     //Get RemoteIO AU from Audio Unit Component Manager.
688     //.....
689
690     //Specify RemoteIO Audio Unit Component Description.
691     AudioComponentDescription RIOUnitDescription;
692     RIOUnitDescription.componentType =
kAudioUnitType_Output;
693     RIOUnitDescription.componentSubType =
kAudioUnitSubType_RemoteIO;
694     RIOUnitDescription.componentManufacturer =
kAudioUnitManufacturer_Apple;
695     RIOUnitDescription.componentFlags = 0;
696     RIOUnitDescription.componentFlagsMask = 0;
697
698     //Get RemoteIO AU from Audio Unit Component Manager
699     AudioComponent rioComponent=AudioComponentFindNext(NULL,
&RIOUnitDescription);
700
701     err=AudioComponentInstanceNew(rioComponent, &effectState._rioAU);
702     [self checkErr:err withMessage:"Failed to create a new instance
of RemoteIO AU!"];
703
704     //.....
705     //Set up the RemoteIO AU.
706     //.....
707     //Enable output bus of RemoteIO.
708     UInt32 enableOutput = 1; // to enable output (enabled by
default).To disable set this to zero.
709     AudioUnitElement outputBus = 0; //Bus 0 of RemoteIO AU is the
hardware output.
710     propsize=sizeof(enableOutput);
711
712     err=AudioUnitSetProperty(effectState._rioAU,
kAudioOutputUnitProperty_EnableIO,
kAudioUnitScope_Output,
outputBus,
&enableOutput,
propsize);
713
714     [self checkErr:err withMessage:"Failed to enable output bus of
RemoteIO AU."];
715
716     /*
717
718
719
720

```

```

721 //Enable input bus of RemoteIO.
722 UInt32 enableInput = 1; // to disable input (disabled by
default). To enable set this to one.
723 AudioUnitElement inputBus = 1; //Bus 1 of RemoteIO AU is the
hardware input.
724 propsize=sizeof(enableInput);
725
726 err=AudioUnitSetProperty(effectState._rioAU,
727 kAudioOutputUnitProperty_EnableIO,
728 kAudioUnitScope_Input,
729 inputBus,
730 &enableInput,
731 propsize);
732 [self checkErr:err withMessage:"Failed to disable input bus of
RemoteIO AU."];
733 */
734
735 //Set the stream format of the RemoteIO AU.
736 propsize=sizeof(effectState._clientASBD);
737
738 //Set format for output (outputBus/input scope).
739 err=AudioUnitSetProperty(effectState._rioAU,
740 kAudioUnitProperty_StreamFormat,
741 kAudioUnitScope_Input,
742 outputBus,
743 &effectState._clientASBD,
744 propsize);
745 [self checkErr:err withMessage:"Failed to set StreamFormat
property of RemoteIO AU (output bus/input scope)!"];
746
747 /*
748 //Set format for input (inputBus/output scope).
749 err=AudioUnitSetProperty(effectState._rioAU,
750 kAudioUnitProperty_StreamFormat,
751 kAudioUnitScope_Output,
752 inputBus,
753 &effectState._clientASBD,
754 propsize);
755 [self checkErr:err withMessage:"Failed to set StreamFormat
property of RemoteIO AU (input bus/output scope)!"];
756 */
757
758 //Set up render callback function for the RemoteIO AU.
759 AURenderCallbackStruct renderCallbackStruct;
760 renderCallbackStruct.inputProc=playbackCallback;
761 renderCallbackStruct.inputProcRefCon=&effectState;
762 propsize=sizeof(renderCallbackStruct);
763
764 err=AudioUnitSetProperty(effectState._rioAU,
765 kAudioUnitProperty_SetRenderCallback,
766 kAudioUnitScope_Global,
767 outputBus,
768 &renderCallbackStruct,
769 propsize);
770 [self checkErr:err withMessage:"Failed to set SetRenderCallback
property for RemoteIO AU!"];
771
772 //.....
773 //Initialise RemoteIO AU.
774 //.....
775 err=AudioUnitInitialize(effectState._rioAU);
776 [self checkErr:err withMessage:"Failed to initialise RemoteIO
AU!"];
777

```

```

778     if(!err)
779         fprintf(stdout,"OK\n");
780
781     return err;
782 }
783
784
785 //----- LF-waveform and 2D DWM implementation by Jacob
       Harrison-----//
786
787 // Set initial values (when App is first opened)
788 // Typical voice with automatic f0
789 -(void)initValues{
790     effectState._f0=110.0;
791     effectState._k = 0;
792
793     //TYPICAL voice type
794     effectState._tcVal = 1.0;
795     effectState._teVal = 0.575;
796     effectState._tpVal = 0.457;
797     effectState._taVal = 0.009;
798
799     effectState._vibratoOn = FALSE;
800     effectState._amp = 0.5;
801     effectState._pitchSlide = FALSE;
802     effectState._autoVoice = TRUE;
803
804     effectState._vowelOn = FALSE;
805 }
806
807 // Set f0 manually if auto-f0 is off
808 -(void)setF0:(Float32)f0Value{
809     if (effectState._pitchSlide == FALSE) {
810         effectState._f0=f0Value;
811         [self setVoiceType];
812         [self createLFInput];
813     }
814 }
815
816 // Set 'vocal tension'
817 -(void)setTension:(Float32)tensionValue{
818     effectState._vocalTension=tensionValue;
819     [self createLFInput];
820 }
821
822 // Set 'Ta' (can also be considered as vocal tension parameter)
823 -(void)setTa:(Float32)taValue{
824     effectState._taVal=taValue;
825     [self createLFInput];
826 }
827
828 -(void)setAmplitude:(Float32)ampValue{
829     effectState._amp = ampValue;
830 }
831
832 // Typical voice type (from VocalModel source code)
833 -(void)setTypical{
834     effectState._voiceType = 0;
835     [self setVoiceType];
836 }
837
838 // Modal voice type
839 -(void)setModal{
840     effectState._voiceType = 1;

```

```

841     [self setVoiceType];
842 }
843
844 // Vocal Fry voice type
845 -(void)setVocalFry{
846     effectState._voiceType = 2;
847     [self setVoiceType];
848 }
849
850 // Breathy Voice
851 -(void)setBreathy{
852     effectState._voiceType = 3;
853     [self setVoiceType];
854 }
855
856 // Falsetto Voice
857 -(void)setFalsetto{
858     effectState._voiceType = 4;
859     [self setVoiceType];
860 }
861
862 -(void)setVoiceType{
863     // This function sets the voice type according to pitch or
      selected type
864
865
866     // pitch thresholds for falsetto and vocal fry. between these
      values, interpolate from vocal fry -> breathy/modal -> falsetto
867     // calculated in number of samples for single pitch period
868     // to avoid confusion: 'max' here indicates maximum number of
      samples, which is in fact the 'minimum' pitch.
869     // these are rough estimates based on conversations with
      supervisor
870     int vocalFryMax, vocalFryMin, falsettoMax, falsettoMin;
871     vocalFryMax = 848; // 52 Hz
872     vocalFryMin = 469; // 94 Hz
873     falsettoMax = 213; // 207 Hz
874     falsettoMin = 153; // 288 Hz
875
876     double interpFraction;
877
878     switch (effectState._voiceType) {
879     case 0: // typical voice type
880         effectState._tcVal = 1.0;
881         effectState._teVal = 0.780;
882         effectState._tpVal = 0.600;
883         effectState._taVal = 0.028;
884         effectState._noiseOn = FALSE;
885         effectState._vocalTension = 0.0;
886         [self createLFInput];
887         break;
888
889     case 1: // modal voice type
890         effectState._tcVal = 1.0;
891         effectState._teVal = 0.575;
892         effectState._tpVal = 0.457;
893         effectState._taVal = 0.009;
894         effectState._noiseOn = FALSE;
895         effectState._vocalTension = 0.0;
896         [self createLFInput];
897         break;
898
899     case 2: // vocal fry voice type
900         effectState._tcVal = 1.0;

```

```

901         effectState._teVal = 0.251;
902         effectState._tpVal = 0.19;
903         effectState._taVal = 0.008;
904         effectState._noiseOn = FALSE;
905         effectState._vocalTension = 0.0;
906         [self createLFInput];
907         break;
908
909     case 3: // breathy voice type
910         effectState._tcVal = 1.0;
911         effectState._teVal = 0.756;
912         effectState._tpVal = 0.529;
913         effectState._taVal = 0.082;
914         effectState._noiseOn = TRUE;
915         effectState._noiseAmount = 0.025;
916         effectState._noiseDuration = 0.5;
917         effectState._noiseStart = 0.75;
918         effectState._vocalTension = 0.0;
919         [self createLFInput];
920
921     case 4: // falsetto voice type
922         effectState._tcVal = 1.0;
923         effectState._teVal = 0.770;
924         effectState._tpVal = 0.570;
925         effectState._taVal = 0.133;
926         effectState._noiseOn = TRUE;
927         effectState._noiseAmount = 0.015;
928         effectState._noiseDuration = 0.5;
929         effectState._noiseStart = 0.75;
930         effectState._vocalTension = 0.0;
931         [self createLFInput];
932         break;
933
934     default:
935         break;
936 }
937
938 // When 'autoVoice' is TRUE, voice type automatically changes
939 // depending on pitch
940 // As f0 rises, voice moves from Vocal Fry -> interpolated Vocal
941 // Fry & Breath/Modal -> Breath/Modal -> interpolated Breath/Modal &
942 // Falsetto -> Falsetto
943 if (effectState._autoVoice == TRUE){
944     // Set to Vocal Fry if dataLength is larger than vocalFryMax
945     // (i.e. if f0 < 52 Hz)
946     if (effectState._dataLength > vocalFryMax) {
947         effectState._tcVal = 1.0;
948         effectState._teVal = 0.251;
949         effectState._tpVal = 0.19;
950         effectState._taVal = 0.008;
951         effectState._noiseOn = FALSE;
952         effectState._vocalTension = 0.0;
953         [self createLFInput];
954     }
955
956     // Interpolate from Vocal Fry to Breath/Modal if dataLength
957     // is between vocalFryMax & vocalFryMin (i.e if 52 < f0 < 94)
958     if ((effectState._dataLength < vocalFryMax) &&
959         (effectState._dataLength > vocalFryMin)){
960         interpFraction = ((double)vocalFryMax -
961             (double)effectState._dataLength)/((double)vocalFryMax -
962             (double)vocalFryMin);

```

```

957
958     if (effectState._voiceType == 3) { // if BREATHY voice is
selected, interpolate from vocal fry params to breathy params
959         effectState._noiseOn = TRUE;
960         effectState._noiseDuration = 0.5;
961         effectState._noiseStart = 0.75;
962
963         effectState._noiseAmount = 0.025*interpFraction;
964         effectState._teVal = (0.251*(1-
interpFraction))+0.756*interpFraction);
965         effectState._tpVal = (0.19*(1-
interpFraction))+0.529*interpFraction);
966         effectState._taVal = (0.008*(1-
interpFraction))+0.082*interpFraction);
967
968
969
970     }
971
972     if (effectState._voiceType == 0) { // if TYPICAL voice is
selected
973         effectState._noiseOn = FALSE;
974         effectState._teVal = (0.251*(1-
interpFraction))+0.780*interpFraction);
975         effectState._tpVal = (0.19*(1-
interpFraction))+0.600*interpFraction);
976         effectState._taVal = (0.008*(1-
interpFraction))+0.028*interpFraction);
977
978     }
979
980     if (effectState._voiceType == 1) { // if MODAL voice is
selected
981         effectState._noiseOn = FALSE;
982         effectState._teVal = (0.251*(1-
interpFraction))+0.575*interpFraction);
983         effectState._tpVal = (0.19*(1-
interpFraction))+0.457*interpFraction);
984         effectState._taVal = (0.008*(1-
interpFraction))+0.028*interpFraction);
985
986     }
987     [self createLFInput];
988
989 }
990
991 //     printf("interp fraction: %f \n", interpFraction);
992
993     // Set to Falsetto if dataLength is less than falsettoMin
(i.e if f0 > 288 Hz)
994     if (effectState._dataLength < falsettoMin) {
995         effectState._tcVal = 1.0;
996         effectState._teVal = 0.770;
997         effectState._tpVal = 0.570;
998         effectState._taVal = 0.133;
999         effectState._noiseOn = TRUE;
1000         effectState._noiseAmount = 0.015;
1001         effectState._noiseDuration = 0.5;
1002         effectState._noiseStart = 0.75;
1003         effectState._vocalTension = 0.0;
1004         [self createLFInput];
1005
1006     }
1007

```

```

1008         // Interpolate from Breathy/Modal to Falsetto if dataLength
is between falsettoMin and falsettoMax (i.e if 207 Hz < f0 < 288 Hz)
1009         if ((effectState._dataLength >= falsettoMin) &&
(effectState._dataLength < falsettoMax)) {
1010             interpFraction = ((double)falsettoMax -
(double)effectState._dataLength)/((double)falsettoMax -
(double)falsettoMin);
1011
1012             if (effectState._voiceType == 3) { // if BREATHY voice is
selected, interpolate from breathy params to falsetto
1013                 effectState._noiseOn = TRUE;
1014                 effectState._noiseDuration = 0.5;
1015                 effectState._noiseStart = 0.75;
1016
1017                 effectState._noiseAmount = (0.025*(1-
interpFraction))+0.015*interpFraction;
1018                 effectState._teVal = (0.756*(1-
interpFraction))+0.770*interpFraction;
1019                 effectState._tpVal = (0.529*(1-
interpFraction))+0.570*interpFraction;
1020                 effectState._taVal = (0.082*(1-
interpFraction))+0.133*interpFraction;
1021
1022 //                                     printf("te = %f, tp = %f, ta = %f,
interp = %f \n", effectState._teVal, effectState._tpVal,
effectState._taVal, interpFraction);
1023
1024             }
1025
1026             if (effectState._voiceType == 0) { // if TYPICAL voice is
selected
1027                 effectState._noiseOn = TRUE;
1028                 effectState._noiseDuration = 0.5;
1029                 effectState._noiseStart = 0.75;
1030
1031                 effectState._noiseAmount = 0.015*interpFraction;
1032                 effectState._teVal = (0.780*(1-
interpFraction))+0.770*interpFraction;
1033                 effectState._tpVal = (0.600*(1-
interpFraction))+0.570*interpFraction;
1034                 effectState._taVal = (0.028*(1-
interpFraction))+0.133*interpFraction;
1035
1036             }
1037
1038             if (effectState._voiceType == 1) { // if MODAL voice is
selected
1039                 effectState._noiseOn = TRUE;
1040                 effectState._noiseDuration = 0.5;
1041                 effectState._noiseStart = 0.75;
1042
1043                 effectState._noiseAmount = 0.015*interpFraction;
1044                 effectState._teVal = (0.575*(1-
interpFraction))+0.770*interpFraction;
1045                 effectState._tpVal = (0.457*(1-
interpFraction))+0.570*interpFraction;
1046                 effectState._taVal = (0.028*(1-
interpFraction))+0.133*interpFraction;
1047
1048             }
1049
1050
1051
1052             [self createLFInput];

```



```

1053
1054     }
1055
1056     // if dataLength is between vocalFryMin and falsettoMax, keep
t values the same
1057     if ((effectState._dataLength >= 213) &&
(effectState._dataLength <= 469)) {
1058         if (effectState._voiceType == 3) { // if BREATHY voice is
selected, interpolate from breathy params to falsetto
1059             effectState._noiseOn = TRUE;
1060             effectState._noiseDuration = 0.5;
1061             effectState._noiseStart = 0.75;
1062
1063             effectState._noiseAmount = 0.025;
1064             effectState._teVal = 0.756;
1065             effectState._tpVal = 0.529;
1066             effectState._taVal = 0.082;
1067
1068         }
1069
1070         if (effectState._voiceType == 0) { // if TYPICAL voice is
selected
1071             effectState._noiseOn = FALSE;
1072             effectState._teVal = 0.780;
1073             effectState._tpVal = 0.600;
1074             effectState._taVal = 0.028;
1075
1076         }
1077
1078         if (effectState._voiceType == 1) { // if MODAL voice is
selected
1079             effectState._noiseOn = FALSE;
1080
1081             effectState._teVal = 0.575;
1082             effectState._tpVal = 0.457;
1083             effectState._taVal = 0.028;
1084
1085         }
1086
1087         [self createLFInput];
1088
1089     }
1090
1091 }
1092
1093 }
1094
1095 -(void)vowelOn{
1096     if (effectState._vowelOn == FALSE) {
1097         effectState._vowelOn = TRUE;
1098     }
1099     else {
1100         effectState._vowelOn = FALSE;
1101     }
1102 }
1103
1104 -(void)pitchSlide{
1105     if (effectState._pitchSlide == FALSE) {
1106         effectState._pitchSlide = TRUE;
1107     }
1108     else {
1109         effectState._pitchSlide = FALSE;
1110     }
1111 }

```

```

1112
1113 -(void)autoVoice{
1114     if (effectState._autoVoice == FALSE) {
1115         effectState._autoVoice = TRUE;
1116     }
1117     else {
1118         effectState._autoVoice = FALSE;
1119     }
1120 }
1121
1122 //-(void)createVibrato{
1123 //     effectState._vibratoDepth = 1;
1124 //     effectState._vibratoFreq = 6;
1125 //     int WTSize =
1126 //         effectState._hardwareSampleRate/effectState._vibratoFreq;
1127 //     double dt = 1/effectState._hardwareSampleRate;
1128 //     int i;
1129 //     double t;
1130 //
1131 //     if (effectState._vibratoOn == TRUE) {
1132 //         for (i = 0; i < WTSize; i++) {
1133 //             t = i*dt;
1134 //             effectState._vibratoOut =
1135 //                 cos(2*MY_PI*effectState._vibratoFreq*t);
1136 //             [self createLFInput];
1137 //         }
1138 //     }
1139 //
1140 //}
1141
1142
1143
1144 // This function calculates LF equation coefficients based on
1145 // currently selected voice type
1146 -(void)createLFInput{
1147     double f0;
1148     int Fs;
1149     int overSample;
1150     Float32 period;
1151     double Ee, Eo;
1152     double tc, te, tp, ta, tn, tb;
1153     double wg;
1154     double areaSum, areal, area2;
1155     double optimumArea;
1156     double epsilon, epsilonTemp, epsilonDiff, epsilonOptimumDiff;
1157     double alpha;
1158     int dataLength;
1159     double vocalTension;
1160
1161     Fs = effectState._hardwareSampleRate;
1162     f0 = effectState._f0;
1163     overSample = 1000;
1164     period = 1/f0;
1165     Ee = 1.0;
1166
1167     tc = effectState._tcVal;
1168     te = effectState._teVal;
1169     tp = effectState._tpVal;
1170     ta = effectState._taVal;
1171
1172     vocalTension = effectState._vocalTension;

```

```

1173
1174     te = te + te*vocalTension;
1175     tp = tp + tp*vocalTension;
1176     ta = ta - ta*vocalTension;
1177
1178
1179
1180     tc = tc*period;
1181     te = te*period;
1182     tp = tp*period;
1183     ta = ta*period;
1184
1185     // These if statements prevent timing parameter values from
going outside of their range with respect to other values.
1186
1187     if (te <= tp) {
1188         te = tp + tp*0.01;
1189     }
1190
1191     if (te >= (tc-ta)){
1192         te = tc-ta - (tc-ta)*0.01;
1193     }
1194
1195
1196
1197     // over sample values (can omit this section if it impacts real
time operation)
1198     period = period/overSample;
1199     tc = tc/overSample;
1200     te = te/overSample;
1201     tp = tp/overSample;
1202     ta = ta/overSample;
1203
1204     // dependant timing parameters
1205     tn = te - tp;
1206     tb = tc - te;
1207
1208     // angular frequency of sinusoid section
1209     wg = MY_PI/tp;
1210
1211     // maximum negative peak value
1212     Eo = Ee;
1213
1214     // epsilon and alpha equation coefficients
1215     areaSum = 1.0;
1216     optimumArea = 1e-14;
1217     epsilonDiff = 10000.0;
1218     epsilonOptimumDiff = 0.1;
1219     epsilonTemp = 1/ta;
1220
1221     // solve iteratively for epsilon
1222     while (abs(epsilonDiff)>epsilonOptimumDiff) {
1223         epsilon = (1/ta)*(1-exp(-epsilonTemp*tb));
1224         epsilonDiff = epsilon - epsilonTemp;
1225         epsilonTemp = epsilon;
1226
1227         if (epsilonDiff<0) {
1228             epsilonTemp = epsilonTemp + (abs(epsilonDiff)/100);
1229         }
1230
1231         if (epsilonDiff>0) {
1232             epsilonTemp = epsilonTemp - (abs(epsilonDiff)/100);
1233         }
1234     }

```

```

1235
1236 // iterate through area balance to get Eo and alpha to give areal
+ area2 = 0
1237
1238 //while ((areaSum< -optimumArea)|| (areaSum>optimumArea)) {
1239 while (areaSum > optimumArea){
1240
1241     alpha = ( log(-Ee/(Eo*sin(wg*te)))/te;
1242
1243     area1 = ( Eo*exp(alpha*te)/(sqrt(alpha*alpha+wg*wg)) ) *
(sin(wg*te-atan(wg/alpha))) + (Eo*wg/(alpha*alpha+wg*wg));
1244
1245     area2 = ( -(Ee)/(epsilon*epsilon*(ta)) ) * (1 - exp(-
epsilon*tb*(1+epsilon*tb)));
1246
1247     areaSum = area1 + area2;
1248
1249     if (areaSum>0.0) {
1250         Eo = Eo - 1e5*areaSum;
1251     }
1252
1253     if (areaSum<0.0) {
1254         Eo = Eo + 1e5*areaSum;
1255     }
1256
1257 }
1258
1259 //calculate length of waveform in samples
1260 if (effectState._pitchSlide == FALSE) {
1261     dataLength = floor(overSample*Fs*period);
1262 }
1263 else{
1264     dataLength = effectState._dataLength;
1265 }
1266
1267 //calculate length of waveform in samples without overSample
1268 // dataLength = floor(Fs*period);
1269
1270 // pass all variables needed for waveform calculation to
effectState
1271 effectState._period = period;
1272 effectState._dataLength = dataLength;
1273 effectState._Eo = Eo;
1274 effectState._Ee = Ee;
1275 effectState._alpha = alpha;
1276 effectState._wg = wg;
1277 effectState._epsilon = epsilon;
1278
1279 effectState._tc = tc;
1280 effectState._te = te;
1281 effectState._tp = tp;
1282 effectState._ta = ta;
1283
1284 // printf("tc = %f, te = %f, tp = %f, ta = %f \n", tc, te, tp,
ta);
1285 }
1286
1287 /*
1288 2D DWM Implementation of Vocal Tract based on Jack Mullen's
VocalModel synthesiser and Amelia Gully's MATLAB port. This function
calculates the impedance maps necessary for a static 'ah' vowel
1289 */
1290
1291 -(void)createDWM{

```

```

1292     int nSlices;
1293     //     double vtLength, vtWidth;
1294     //     int fs;
1295     //     double wgSize;
1296     int sizeXMax;
1297     int sizeYMax;
1298     double reflectionGlottis, reflectionLips, reflectionWalls;
1299     double zPower;
1300     double interpFraction;
1301     double count;
1302     int prevSlice, nextSlice;
1303     double zMin1, zMin, zX, zXPlus, zXY, zXPlusY, zXPlusYPlus,
zXYPlus;
1304
1305     nSlices = 44;
1306     // initialise array manually to avoid reading the vowel .txt
file. May need to find a way around this later if diphthongs are
needed
1307     double areaData[] = {    0.5625,
1308         0.4620,
1309         0.2074,
1310         0.2701,
1311         0.2139,
1312         0.3268,
1313         0.3224,
1314         0.3771,
1315         1.0755,
1316         1.0756,
1317         0.7692,
1318         0.6199,
1319         0.3926,
1320         0.2908,
1321         0.2182,
1322         0.2829,
1323         0.3003,
1324         0.2896,
1325         0.2978,
1326         0.4969,
1327         0.8996,
1328         1.1699,
1329         1.2484,
1330         1.9049,
1331         2.3760,
1332         2.7038,
1333         2.8647,
1334         2.8949,
1335         3.4448,
1336         4.3501,
1337         4.9672,
1338         5.6775,
1339         6.5906,
1340         6.3094,
1341         6.2851,
1342         6.0513,
1343         5.3630,
1344         4.8184,
1345         3.9153,
1346         4.0989,
1347         4.2489,
1348         4.2713,
1349         4.6729,
1350         5.0273};
1351
1352

```

```

1353     /*
1354
1355     - THIS SECTION CALCULATES 'sizeXMax' and 'sizeYMax' BASED ON
LENGTH OF VOCAL TRACT/WAVEGUIDE SIZE.
1356     - THEY ARE #DEFINED IN THE HEADER FILE INSTEAD SO THIS PART IS
COMMENTED OUT (SEE AudioEngine.h)
1357
1358     vtLength = 0.1750;
1359     vtWidth = 0.0500;
1360
1361     fs = effectState._hardwareSampleRate;
1362
1363     // Calculate waveguide size
1364     // waveguide size = sqrt(dimensionality) * speed of sound /
sampling freq.
1365     wgSize = sqrt(2) * 343 / fs;
1366
1367     // Number of waveguides in X and Y directions
1368     sizeXMax = (int)floor(vtLength/wgSize);
1369     sizeYMax = (int)floor(vtWidth/wgSize);
1370     // printf("sizeXMax = %d, sizeYMax = %d, wgSize =
%f",sizeXMax, sizeYMax, wgSize);
1371
1372     */
1373
1374     sizeXMax = SIZE_X_MAX;
1375     sizeYMax = SIZE_Y_MAX;
1376
1377     // initialise empty arrays for pressure and impedance
1378     // Pressure:
1379     Float32 pNPlus[sizeYMax*sizeXMax];
1380     Float32 pNMinus[sizeYMax*sizeXMax];
1381     Float32 pEPlus[sizeYMax*sizeXMax];
1382     Float32 pEMinus[sizeYMax*sizeXMax];
1383     Float32 pSPlus[sizeYMax*sizeXMax];
1384     Float32 pSMinus[sizeYMax*sizeXMax];
1385     Float32 pWPlus[sizeYMax*sizeXMax];
1386     Float32 pWMinus[sizeYMax*sizeXMax];
1387     // Initialise pressure arrays to 0
1388     for (int y = 0; y < sizeYMax; y++) {
1389         for (int x = 0; x < sizeXMax; x++) {
1390             pNPlus[y*sizeXMax + x] = 0;
1391             pNMinus[y*sizeXMax + x] = 0;
1392             pEPlus[y*sizeXMax + x] = 0;
1393             pEMinus[y*sizeXMax + x] = 0;
1394             pSPlus[y*sizeXMax + x] = 0;
1395             pSMinus[y*sizeXMax + x] = 0;
1396             pWPlus[y*sizeXMax + x] = 0;
1397             pWMinus[y*sizeXMax + x] = 0;
1398         }
1399     }
1400
1401     // Impedance
1402     Float32 zNorth[sizeYMax*sizeXMax];
1403     Float32 zEast[sizeYMax*sizeXMax];
1404     Float32 zSouth[sizeYMax*sizeXMax];
1405     Float32 zWest[sizeYMax*sizeXMax];
1406     // Initialise impedance arrays to 1
1407     for (int y = 0; y < sizeYMax; y++) {
1408         for (int x = 0; x < sizeXMax; x++) {
1409             zNorth[y*sizeXMax + x] = 1;
1410             zEast[y*sizeXMax + x] = 1;
1411             zSouth[y*sizeXMax + x] = 1;
1412             zWest[y*sizeXMax + x] = 1;

```

```

1413     }
1414 }
1415
1416 //    Set reflection parameters
1417 reflectionGlottis = 0.97;
1418 reflectionLips = -0.9;
1419 reflectionWalls = 0.94;
1420
1421 //    Impedance map
1422 //    Power for impedance/area function (power to which radius is
raised)
1423 zPower = 2.5;
1424
1425 //    Convert from cm sq to diameter in m
1426 double diamData1[nSlices];
1427
1428 for (int i = 0; i < nSlices; i++) {
1429     diamData1[i] = 0.02*(sqrt(areaData[i]/MY_PI));
1430
1431     //    printf("%f\n", diamData1[i]);
1432 }
1433
1434
1435 //    Might need to find way of performing zero-check here (like
in MATLAB script)
1436
1437 //    Interpolate areaData from original size to new size (no. of
waveguides in X direction)
1438 interpFraction = (double)nSlices/(double)sizeXMax;
1439
1440 double diamData[sizeXMax];
1441 double impData[sizeXMax];
1442
1443 for (int i = 0; i < sizeXMax; i++) {
1444     count = (i+1)*interpFraction;
1445     prevSlice = floor(count)+1;
1446
1447     if (prevSlice > nSlices) {
1448         prevSlice = prevSlice - 1;
1449     }
1450
1451     if (prevSlice == nSlices) {
1452         nextSlice = ceil(count);
1453     }
1454     else {
1455         nextSlice = ceil(count)+1;
1456     }
1457
1458     diamData[i] = diamData1[prevSlice-1] + ((diamData1[nextSlice-
1] - diamData1[prevSlice-1]) * (count-prevSlice));
1459
1460
1461     impData[i] = 1/diamData[i];
1462 }
1463
1464 zMin1 = impData[0];
1465
1466 for (int i=0; i<sizeXMax; i++) {
1467     if (impData[i] < zMin1) {
1468         zMin1 = impData[i];
1469     }
1470 }
1471 zMin = pow(zMin1, zPower);
1472

```

```

1473 //   initialiase array to store raised cosine in
1474 double raisedCosine[sizeYMax];
1475
1476 //   go through each x location and generate raised-cosine
function
1477   for (int x=0; x<sizeXMax; x++) {
1478
1479       //           impedance at edges of raised cosine (max impedance)
1480       zX = pow(impData[x], zPower);
1481
1482       //           max impedance of next slice (reuse final value when
x+1 > sizeXMax)
1483       if (x == sizeXMax-1) {
1484           zXPlus = pow(impData[x], zPower);
1485       }
1486       else {
1487           zXPlus = pow(impData[x+1], zPower);
1488       }
1489
1490       //           fill array with raised cosine of size 'sizeYMax'
1491       for (int i = 0; i<sizeYMax; i++) {
1492
1493           raisedCosine[i] = (0.5 +
0.5*cos(2*MY_PI*((double)i/(double)(sizeYMax-1))));
1494       }
1495
1496
1497       //           cycle through each y-direction sample in the
current slice and assign it an impedance value
1498       for (int y = 0; y < sizeYMax; y++) {
1499
1500           //           calculate impedance value at points
surrounding current x/y
1501           zXY = zMin + (zX - zMin)*raisedCosine[y];
1502           zXPlusY = zMin + (zXPlus - zMin) * raisedCosine[y];
1503
1504           //           again reuse last value for y+1
1505           if (y == sizeYMax-1) {
1506               zXYPlus = zMin + (zX - zMin) * raisedCosine[y];
1507               zXPlusYPlus = zMin + (zXPlus - zMin) *
raisedCosine[y];
1508           }
1509           else {
1510               zXYPlus = zMin + (zX - zMin) * raisedCosine[y+1];
1511               zXPlusYPlus = zMin + (zXPlus - zMin) *
raisedCosine[y+1];
1512           }
1513
1514           //           use averages of the relevant values to
update impedance map
1515           zNorth[y*sizeXMax + x] = (zXYPlus + zXPlusYPlus)/2;
1516           zEast[y*sizeXMax + x] = (zXPlusYPlus + zXPlusY)/2;
1517           zSouth[y*sizeXMax + x] = (zXPlusY + zXY)/2;
1518           zWest[y*sizeXMax + x] = (zXYPlus + zXY)/2;
1519
1520
1521
1522       }
1523   }
1524
1525
1526
1527 //   assign values of pressure and impedance arrays to those in
effectState structure

```



```

1528     for (int x = 0; x < sizeXMax; x++) {
1529         effectState._impData[x] = impData[x];
1530         for (int y = 0; y < sizeYMax; y++) {
1531             effectState._pNPlus[y*sizeXMax + x] = pNPlus[y*sizeXMax +
x];
1532             effectState._pNMinus[y*sizeXMax + x] = pNPlus[y*sizeXMax
+ x];
1533             effectState._pEPlus[y*sizeXMax + x] = pEPlus[y*sizeXMax +
x];
1534             effectState._pEMinus[y*sizeXMax + x] = pEPlus[y*sizeXMax
+ x];
1535             effectState._pSPlus[y*sizeXMax + x] = pSPlus[y*sizeXMax +
x];
1536             effectState._pSMinus[y*sizeXMax + x] = pSPlus[y*sizeXMax
+ x];
1537             effectState._pWPlus[y*sizeXMax + x] = pWPlus[y*sizeXMax +
x];
1538             effectState._pWMinus[y*sizeXMax + x] = pWPlus[y*sizeXMax
+ x];
1539
1540             effectState._zNorth[y*sizeXMax + x] = zNorth[y*sizeXMax +
x];
1541             effectState._zEast[y*sizeXMax + x] = zEast[y*sizeXMax +
x];
1542             effectState._zSouth[y*sizeXMax + x] = zSouth[y*sizeXMax +
x];
1543             effectState._zWest[y*sizeXMax + x] = zWest[y*sizeXMax +
x];
1544
1545
1546         }
1547     }
1548
1549     effectState._sizeXMax = sizeXMax;
1550     effectState._sizeYMax = sizeYMax;
1551
1552     effectState._reflectionGLottis = reflectionGlottis;
1553     effectState._reflectionLips = reflectionLips;
1554     effectState._reflectionWalls = reflectionWalls;
1555
1556
1557 }
1558
1559
1560 //----- End of LF-waveform and 2D DWM implementation by
Jacob Harrison-----//
1561
1562
1563 /*
1564 //-----
1565 //     initFileManager function
1566 //-----
1567 -(void)initFileManager{
1568
1569     fprintf(stdout,"Initialising file manager...");
1570     NSError *err;
1571     //Get an instance of the default file manager.
1572     _FileManager=[NSFileManager defaultManager];
1573
1574     //Get documents directory in app space.
1575     NSArray *dirPaths=[_FileManager URLsForDirectory:NSDocumentDirectory
1576     inDomains:NSUserDomainMask];
1577
1578     NSURL *_docsURL=[dirPaths objectAtIndex:0];

```

```

1579
1580 //Create a directory where all exported audio files are saved.
1581 _audioDirURL =[NSURL URLWithString:@"audio" relativeToURL:_docsURL];
1582
1583
1584 [_FileManager createDirectoryAtURL:_audioDirURL
1585 withIntermediateDirectories:YES
1586 attributes:nil
1587 error:&err];
1588 [self checkErr:[err code] withMessage:"Failed to create audio
1589 directory"];
1589
1590 if(!err)
1591 fprintf(stdout,"OK\n");
1592
1593 }
1594 */
1595
1596 //-----
1597 //      startPlayback function
1598 //-----
1599 -(OSStatus)startPlayback{
1600     OSStatus err=0;
1601
1602     //Reset sample into loop
1603     //_sampleIntoLoop=0;
1604
1605     //_recording=NO; //unless monitor option is available.
1606
1607     err=AudioOutputUnitStart(effectState._rioAU);
1608     [self checkErr:err withMessage:"Failed to start RemoteIO AU!"];
1609     if(!err){
1610         _playing=YES;
1611         fprintf(stdout,"Playing audio...\n");
1612     }
1613
1614     return err;
1615 }
1616
1617 //-----
1618 //      stopPlayback function
1619 //-----
1620 -(OSStatus)stopPlayback{
1621     OSStatus err=0;
1622
1623     //_sampleStoppedRecording=_sampleIntoLoop; //Keep this for
1624     playback and saving file purposes
1625
1626     //Reset smaple into loop
1627     //_sampleIntoLoop=0;
1628
1629     err=AudioOutputUnitStop(effectState._rioAU);
1630     [self checkErr:err withMessage:"Failed to stop RemoteIO AU!"];
1631
1632     if(!err){
1633         fprintf(stdout, "Stopped audio.\n");
1634         _playing=NO;
1635     }
1636
1637     return err;
1638 }
1639 //-----
1640 //      isPlaying function

```

```

1641 //-----
1642 -(BOOL)isPlaying{
1643     return _playing;
1644 }
1645
1646 /*
1647 //-----
1648 //     exportAudio function
1649 //-----
1650 -(OSStatus)exportAudioWithName:(NSString*)outputFileName
1651
1652 {
1653     OSStatus err=0;
1654
1655     //Declare and set-up an ASBD structure for the output file.
1656     AudioStreamBasicDescription outputASBD;
1657     outputASBD.mFormatID = kAudioFormatLinearPCM;
1658     outputASBD.mFormatFlags
1659     =kAudioFormatFlagIsSignedInteger|kAudioFormatFlagIsPacked;
1660     outputASBD.mSampleRate = effectState._hardwareSampleRate;
1661     outputASBD.mChannelsPerFrame = 2;
1662     outputASBD.mBitsPerChannel = 16;
1663     outputASBD.mBytesPerFrame =
1664     outputASBD.mChannelsPerFrame*(outputASBD.mBitsPerChannel/8);
1665     outputASBD.mBytesPerPacket =outputASBD.mBytesPerFrame;
1666     outputASBD.mFramesPerPacket = 1;
1667     outputASBD.mReserved=0;
1668
1669     // Declare and initialise an audio file reference.
1670     ExtAudioFileRef extAudioFileObj = 0;
1671
1672     //Set the type of the output file.
1673     AudioFileTypeID outputFileType=kAudioFileWAVEType;
1674
1675     //Extension of output file.
1676     NSString *fileExtension=@"wav";
1677
1678     // Create full path for output file.
1679     NSString *fullPath=[NSString
1680     stringWithFormat:@"%@/%@.%@",[_audioDirURLpath],outputFileName,fileE
1681     xtension];
1682
1683     // Create file URL.
1684     NSURL *fileURL = [NSURL fileURLWithPath:fullPath isDirectory:NO];
1685
1686     // Get CFURLRef from NSURL.
1687     CFURLRef outputAudioFileURL=(CFURLRef)CFBridgingRetain(fileURL);
1688
1689     // Create an extended audio file object and link it to the reference
1690     above.
1691     err=ExtAudioFileCreateWithURL(outputAudioFileURL,
1692     outputFileType,
1693     &outputASBD,
1694     NULL,
1695     kAudioFileFlags_EraseFile,
1696     &extAudioFileObj);
1697     [self checkErr:err withMessage:"Failed to create a file for
1698     output."];
1699
1700     //Set client ASDB property of extended audio file. The object needs
1701     to know the
1702     //data it will accept so that it converts it accordingly.
1703     err=ExtAudioFileSetProperty(extAudioFileObj,
1704     kExtAudioFileProperty_ClientDataFormat,

```

```

1698 sizeof(AudioStreamBasicDescription),
1699 &effectState._clientASBD);
1700 [self checkErr:err withMessage:"Failed to set client ASBD."];
1701
1702 fprintf(stdout, "Writting audio file to disk...");
1703 //Write the data found in _ioBuffer into the file pointed by
    extAudioFileObj.
1704 err=ExtAudioFileWrite(extAudioFileObj,
1705 _smpPerLoop,//This could be a user setting. If user changes this
    size of _ioBuffer has to change accordingly.
1706 _ioBuffer);//Our data is in ioBuffer AudioBufferList.
1707 [self checkErr:err withMessage:"Failed to write file to disk!"];
1708
1709 //We finished with the file so dispose the reference properly.
1710 err=ExtAudioFileDispose(extAudioFileObj);
1711 [self checkErr:err withMessage:"Failed to dispose extended audio
    file reference"];
1712
1713 if(!err)
1714 fprintf(stdout, "OK\n");
1715 return err;
1716 }
1717 */
1718
1719 #pragma mark Utility Functions
1720 //-----
1721 //      checkErr function
1722 //-----
1723 - (void) checkErr: (OSStatus) error
1724     withMessage:(const char *) message{
1725
1726     if(error==noErr) return;
1727
1728     char errorString[20];
1729     *(UInt32*)(errorString+1)=CFSwapInt32HostToBig(error);
1730
1731     if(isprint(errorString[1]) && isprint(errorString[2])
1732         && isprint(errorString[3])&&isprint(errorString[4])) {
1733         errorString[0]=errorString[5]='\ ';
1734         errorString[6]='\0';
1735     }
1736     else sprintf(errorString, "%d", (int)error);
1737
1738     fprintf(stderr, "\nError: %s [Error code:
1739 %s]\n",message,errorString);
1740 }
1741 @end

```

References

- [1] "Siri," *apple.com*. [Online]. Available: <https://www.apple.com/ios/siri/>. [Accessed: 11-Nov-2014].
- [2] W. I. Hallahan, "DECTalk software: Text-to-speech technology and implementation," *Digital Technical Journal*, 1995.
- [3] T. Guerreiro, H. Nicolau, J. Jorge, and D. Gonçalves, "Towards accessible touch interfaces," presented at the the 12th international ACM SIGACCESS conference, New York, New York, USA, 2010, p. 19.
- [4] L. Anthony, Y. Kim, and L. Findlater, "Analyzing user-generated youtube videos to understand touchscreen use by people with motor impairments," presented at the the SIGCHI Conference, New York, New York, USA, 2013, p. 1223.
- [5] J. Harrison, "Exploring the Naturalness of Speech Synthesis using a Real-Time Handheld Controller Device" May 2013.
- [6] D. M. Howard and D. T. Murphy, *Voice Science, Acoustics and Recording*. 2008.
- [7] G. Fant, "Voice source dynamics," *Speech Transmission Labs-Quarterly Progress Status ...*, 1980.
- [8] I. R. Titze, "On the mechanics of vocal-fold vibration," *The Journal of the Acoustical Society of America*, 1976.
- [9] K. E. Cummings and M. A. Clements, "Glottal models for digital speech processing: A historical survey and new results," *Digital Signal Processing*, 1995.
- [10] J. W. Van den Berg and J. T. Zantema, "On the air resistance and the Bernoulli effect of the human larynx," *The Journal of the Acoustical Society of America*, vol. 29, no. 5, p. 626, 1957.
- [11] H. Babinsky, "How do wings work?," *Physics Education*, 38(6) pp. 1–8, Oct. 2003.
- [12] G. Fant, J. Liljencrants, and Q. Lin, "A four-parameter model of glottal flow," *STL-QPSR*, 1985.
- [13] G. Fant, "Preliminaries to analysis of the human voice source," *STL-QPSR*, 1982.
- [14] D. G. Childers and C. K. Lee, "Vocal quality factors: Analysis, synthesis, and perception," *The Journal of the Acoustical Society of America*, vol. 90, no. 5, pp. 2394–2410, Nov. 1991.
- [15] C. Gobl, "The role of voice quality in communicating emotion, mood and attitude," *Speech Communication*, vol. 40, no. 1, pp. 189–212, Apr. 2003.
- [16] G. Chen, J. Kreiman, Y. L. Shue, A. Alwan, and D. Australia, "Acoustic Correlates of Glottal Gaps.," *INTERSPEECH*, 2011.

- [17] J. Sundberg, "Acoustic and psychoacoustic aspects of vocal vibrato," *STL-QPSR*, 1995.
- [18] J. Sundberg, "Vibrato and vowel identification," *Arch Acoust*, 1977.
- [19] S. E. Levinson and C. E. Schmidt, "Adaptive computation of articulatory parameters from the speech signal," *The Journal of the Acoustical Society of America*, 1983.
- [20] L. R. Rabiner, "Digital-Formant Synthesiser for Speech-Synthesis Studies," *The Journal of the Acoustical Society of America*, 2005.
- [21] H. Li, R. Scaife, and D. O'Brien, "LF model based glottal source parameter estimation by extended Kalman filtering," presented at the *Proceedings of the 22nd IET Irish Signals and Systems Conference*, 2011.
- [22] G. Chen, M. Garellek, J. Kreiman, B. R. Gerratt, and A. Alwan, "A perceptually and physiologically motivated voice source model," presented at the INTERSPEECH, 2013, pp. 2001–2005.
- [23] B. Cranen and L. Boves, "Pressure measurements during speech production using semiconductor miniature pressure transducers: Impact on models for speech production," *The Journal of the Acoustical Society of America*, 1985.
- [24] Q. Fu and P. Murphy, "Robust Glottal Source Estimation Based on Joint Source-Filter Model Optimization," *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 14, no. 2, pp. 492–501.
- [25] A. E. Rosenberg, "Effect of Glottal Pulse Shape on the Quality of Natural Vowels," *The Journal of the Acoustical Society of America*, 2005.
- [26] H. Fujisaki and M. Ljungqvist, "Proposal and evaluation of models for the glottal source waveform," *IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 11, pp. 1605–1608, Apr. 1986.
- [27] I. R. Titze, "A four-parameter model of the glottis and vocal fold contact area," *Speech Communication*, vol. 8, no. 3, pp. 191–201, 1989.
- [28] I. R. Titze, "Parameterization of the glottal area, glottal flow, and vocal fold contact area," *The Journal of the Acoustical Society of America*, 1984.
- [29] I. R. Titze and D. T. Talkin, "A theoretical study of the effects of various laryngeal configurations on the acoustics of phonation," *The Journal of the Acoustical Society of America*, 1979.
- [30] J. Mullen, "Physical modelling of the vocal tract with the 2D digital waveguide mesh," 2006.
- [31] D. G. Childers and K. Wu, "Quality of speech produced by analysis-synthesis," *Speech Communication*, vol. 9, no. 2, pp. 97–117, 1990.

- [32] D. H. Klatt, "Software for a cascade/parallel formant synthesiser," *The Journal of the Acoustical Society of America*, vol. 67, no. 3, p. 971, 1980.
- [33] D. G. Childers and C. Ahn, "Modelling the glottal volume-velocity waveform for three voice types," *The Journal of the Acoustical Society of America*, vol. 97, no. 1, pp. 505–519, Jan. 1995.
- [34] J. Pérez and A. Bonafonte, "Automatic voice-source parameterization of natural speech.," *INTERSPEECH*, 2005.
- [35] H. Li, R. Scaife, D. O'Brien, "Automatic LF-Model Fitting to the Glottal Source Waveform by Extended Kalman Filtering," *Signal Processing Conference (EUSIPCO), 2012 Proceedings of the 20th European* pp. 1–5, Oct. 2012.
- [36] M. Airas and P. Alku, "Comparison of multiple voice source parameters in different phonation types.," *INTERSPEECH*, 2007.
- [37] P. Alku, T. Bäckström, and E. Vilkman, "Normalized amplitude quotient for parametrization of the glottal flow," *the Journal of the Acoustical Society of America*, vol. 112, no. 2, pp. 701-710, 2002.
- [38] G. Fant, "Some problems in voice source analysis," *Speech Communication*, vol. 13, no. 1, pp. 7–22, 1993.
- [39] R. B. Mosen and A. M. Engebretson, "Study of variations in the male and female glottal wave," *The Journal of the Acoustical Society of America*, 1977.
- [40] P. Palo, "A review of articulatory speech synthesis". 2006.
- [41] J. L. Kelly and C. C. Lochbaum, "Speech synthesis," *In Proc. Fourth Int. Congr. Acoustics (September 1962)*, pp. 1-4, pp. 1–4, 1962.
- [42] J. O. Smith, *Efficient simulation of the reed-bore and bow-string mechanisms*. 1986.
- [43] J. d'Alembert, *Investigation of the curve formed by a vibrating string*. Acoustics: Historical and Philosophical Development, 1973.
- [44] J. O. Smith, *Physical audio signal processing: For virtual musical instruments and audio effects*. 2010.
- [45] S. A. Van Duyne and J. O. Smith, "Physical modelling with the 2-D digital waveguide mesh," presented at the Proceedings of the International ..., 1993.
- [46] D. Murphy, A. Kelloniemi, J. Mullen, and S. Shelley, "Acoustic Modelling Using the Digital Waveguide Mesh," *Signal Processing Magazine, IEEE*, vol. 24, no. 2, pp. 55–66, Mar. 2007.
- [47] D. T. Murphy and D. M. Howard, "Digital waveguide mesh topologies in room acoustics modelling," 2000.
- [48] M. J. Beeson and D. T. Murphy, "RoomWeaver: A digital waveguide mesh based room acoustics research tool," *Int Conference on Digital*

- Audio Effects (DAFx'04)*, 2004.
- [49] J. Mullen, D. M. Howard, and D. T. Murphy, *Digital waveguide mesh modelling of the vocal tract acoustics*. IEEE, 2003, pp. 119–122.
- [50] “Acoustical simulations of the human vocal tract using the 1D and 2D digital waveguide software model,” 2004.
- [51] J. Mullen, D. M. Howard, and D. T. Murphy, “Real-Time Dynamic Articulations in the 2-D Waveguide Mesh Vocal Tract Model,” *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 15, no. 2, pp. 577–585, Feb. 2007.
- [52] M. D. A. Speed, *Voice Synthesis Using the Three-dimensional Digital Waveguide Mesh*. 2012.
- [53] P. A. Yushkevich, J. Piven, H. C. Hazlett, R. G. Smith, and S. Ho, “User-guided 3D active contour segmentation of anatomical structures: significantly improved efficiency and reliability,” *Neuroimage*, 2006.
- [54] W. von Kempelen, *Mechanismus der menschlichen Sprache*. 1791.
- [55] H. Dudley and T. H. Tarnoczy, “The speaking machine of Wolfgang von Kempelen,” *The Journal of the Acoustical Society of America*, 1950.
- [56] J. Trouvain and F. Brackhane, “The relevance of today Wolfgang von Kempelen's speaking machine,” 2011.
- [57] “Talking Heads: Simulacra,” *haskins.yale.edu*. [Online]. Available: <http://www.haskins.yale.edu/featured/heads/SIMULACRA/kempelen.html>. [Accessed: 11-Nov-2014].
- [58] “Apollo Ensemble,” *apolloensemble.co.uk*. [Online]. Available: <http://www.apolloensemble.co.uk/>. [Accessed: 10-Nov-2014].
- [59] “QTC™ Material Technology,” *peratech.com*. [Online]. Available: <http://www.peratech.com/qtc-technology.html>. [Accessed: 10-Nov-2014].
- [60] C. H. Newell, “Place authenticity and time : A framework for liveness in synthetic speech,” 2009.
- [61] M. Mori, 1970. “*The Uncanny Valley*.” Trans. Karl F. McDorman and Norri Kageki. IEEE, 2012.
- [62] “The Uncanny Valley,” *spectrum.ieee.org*. [Online]. Available: <http://spectrum.ieee.org/automaton/robotics/humanoids/the-uncanny-valley>. [Accessed: 11-Nov-2014].
- [63] R. K. Moore, “A Bayesian explanation of the ‘Uncanny Valley’ effect and related psychological phenomena,” *Scientific Reports*, vol. 2, Nov. 2012.
- [64] A. D. N. Edwards, *Speech synthesis*. Paul Chapman Educational Publishing, 1991.
- [65] J. Han, “Multi-touch Interaction Research,” *cs.nyu.edu*. [Online]. Available: <http://cs.nyu.edu/~jhan/ftirtouch/>. [Accessed: 11-Nov-2014].

- [66] N. Tillmann, M. Moskal, J. de Halleux, M. Fahndrich, J. Bishop, A. Samuel, and T. Xie, "The future of teaching programming is on mobile devices," presented at the the 17th ACM annual conference, New York, New York, USA, 2012, p. 156.
- [67] "Core Audio Overview," *developer.apple.com*. [Online]. Available: <https://developer.apple.com/library/mac/documentation/MusicAudio/Conceptual/CoreAudioOverview/Introduction/Introduction.html>. [Accessed: 11-Nov-2014].
- [68] M. Brooks, "VOICEBOX: Speech Processing Toolbox for MATLAB," *ee.ic.ac.uk*. [Online]. Available: <http://www.ee.ic.ac.uk/hp/staff/dmb/voicebox/voicebox.html>. [Accessed: 11-Nov-2014].
- [69] R. Bristow-Johnson, "Wavetable Synthesis 101, A Fundamental Perspective," Nov. 1996.
- [70] "Core Plot source code and example applications," *github.com*. [Online]. Available: <https://github.com/core-plot/core-plot>. [Accessed: 11-Nov-2014].
- [71] J. Mullen and D. T. Murphy, "Vocal Tract Modelling with the 2D Digital Waveguide Mesh." [Online]. Available: <http://www-users.york.ac.uk/~dtm3/vocaltract.html>. [Accessed: 11-Nov-2014].
- [72] H. Silén, E. Helander, J. Nurminen, and M. Gabbouj, "Parameterization of vocal fry in HMM-based speech synthesis.," *INTERSPEECH*, 2009.
- [73] "Model-View-Controller," *developer.apple.com*. [Online]. Available: <https://developer.apple.com/library/mac/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html>. [Accessed: 11-Nov-2014].
- [74] P. Boersma and D. Weenink, "Praat: doing phonetics by computer," *fon.hum.uva.nl*. [Online]. Available: <http://www.fon.hum.uva.nl/praat/>. [Accessed: 11-Nov-2014].
- [75] "Nyquist Analyze Plug-ins," *wiki.audacityteam.org*. [Online]. Available: http://wiki.audacityteam.org/wiki/Nyquist_Analyze_Plug-ins#Pitch_Detect. [Accessed: 11-Nov-2014].
- [76] "Soundflower," *code.google.com*. [Online]. Available: <https://code.google.com/p/soundflower/>. [Accessed: 17-Feb-2015].
- [77] "Digital Waveguides," 2010. [Online]. Available: https://ccrma.stanford.edu/~jos/pasp/Digital_Waveguides.html#11226.
- [78] "Speech Assessment Methods Phonetic Alphabet," *wikipedia.org*. [Online]. Available: http://en.wikipedia.org/wiki/Speech_Assessment_Methods_Phonetic_Alphabet. [Accessed: 12-Nov-2014].

- [79] T. P. Szynalski, "English vowel chart," *antimoon.com*. [Online]. Available: <http://www.antimoon.com/how/english-vowel-chart.htm>. [Accessed: 12-Nov-2014].
- [80] A. Hunt, D. M. Howard, G. Morrison, and J. Worsdall, "A real-time interface for a formant speech synthesiser," *Logopedics Phoniatics Vocology*, vol. 25, no. 4, pp. 169–175, 2000.
- [81] sensimetrics, "One-Hand-Operated Speech Synthesis," *sens.com*. [Online]. Available: <http://www.sens.com/handsynth/index.htm>. [Accessed: 12-Nov-2014].

