INTERACTIVE SPEECH-DRIVEN FACIAL ANIMATION

by

WARREN HODGKINSON

DISSERTATION

submitted in the fulfilment
of the requirements for the degree

MASTER OF SCIENCE

in

COMPUTER SCIENCE

in the

FACULTY OF SCIENCE

at the

UNIVERSITY OF JOHANNESBURG

SUPERVISOR: MR A HARDY
CO-SUPERVISOR: PROF S VON SOLMS

JANUARY 2005

Summary

One of the fastest developing areas in the entertainment industry is digital animation. Television programmes and movies frequently use 3D animations to enhance or replace actors and scenery. With the increase in computing power, research is also being done to apply these animations in an interactive manner. Two of the biggest obstacles to the success of these undertakings are control (manipulating the models) and realism. This text describes many of the ways to improve control and realism aspects, in such a way that interactive animation becomes possible. Specifically, lip-synchronisation (driven by human speech), and various modeling and rendering techniques are discussed. A prototype that shows that interactive animation is feasible, is also described.

Audio processing

In order to perform lip-synchronisation, speech processing techniques are required. The more widely used of these techniques are described in this text. These include feature extraction and classification. One chapter also describes the difficulties that need to be overcome, in order to perform audio processing successfully (especially human speech for lip-synchronisation).

Facial rasterisation

There are many different techniques that can be used to improve the quality of the animation. Different methods that can be used to represent the models mathematically are presented. Techniques that can be used to improve the quality of the rendered model are also presented, including shading and texturing. Ways to make use of dedicated 3D hardware are also presented.

Implementation

A prototype that evaluates the implications of these techniques all working together, is described. The implementation of methods for performing speech processing and classification are explained. The results of the performance and accuracy tests on the prototype show that interactive speech-driven animation is feasible. The speed at which modeling and rendering techniques implemented in the prototype execute, further illustrate that interactive animation is feasible.

Samevatting

Een van die velde in die vermaak industrie wat die vinnigste groei is dié van digitale animasie. Televisie programme en rolprente gebruik dikwels 3D animasie om akteurs en toneelstukke te verbeter of vervang. Met die verbetering van rekenaar krag, word baie navorsing gedoen om hierdie animasies in 'n wisselwerkende manier te implementeer. Twee van die grootste probleem areas in die sukses van die ondernemings, is die beheer en die akkuraatheid van die modelle. Hierdie verhandeling beskryf baie van die metodes wat gebruik word om die probleme te oorkom, om wisselwerkende animasie moontlik te maak. Spesifiek word lip-sinchronisasie, en die verskillende uitvoerings tegnieke vir modelleerwerk en grafika-vertolking bespreek. 'n Prototipe wat wys dat wisselwerkende animasie doenbaar is, word ook beredeneer.

Klank prosessering

Om lip-sinchronisasie metodes te implementeer, is die gebruik van spraak prosessering tegnieke noodsaaklik. Dié verhandeling beskryf die metodes wat die meeste gebruik word in klank prosessering applikasies. Dit sluit in kenmerk berekeninge en klassifikasie. Een hoofstuk hanteer ook die moeilikhede wat 'n mens moet oorkom om spraak prosessering suksesvol te implementeer.

Gesigs Vertoning

Daar is baie metodes wat gebruik word om die kwaliteit van animasie te verbeter. Verskillende maniere wat die modelle op 'n wiskundige manier weergee word verduidelik. Beskrywings van die tegnieke wat die voorkoms van die model verbeter is ook in hierdie verhandeling ingesluit. Die gebruik van 3D hardeware vir die implementering van die stelsel word ook bespreek, asook skadueering end tekstuur.

Implementasie

'n Prototipe wat die implikasies van die samewerking van die tegnieke evalueër, word in hierdie verhandeling beskryf. Die implementasie van die metodes wat spraak prosessering en klasifikasie hanteer word verduidelik. Die resultate van die spoed en akkuraatheids toetse op die prototipe, wys dat wisselwerkende spraak-aangedrewe animasie uitvoerbaar is. Die spoed van die gebruikte modelleerwerk and grafika-vertolkings tegnieke dui verder aan dat wisselwerkende animasie wel doenbaar is.

Contents

Contents iv

List of Figures ix

List of Tables xi

I Introduction 1

1 Introduction to Audio Driven Facial Animations 2

1.1 Problem Statement . . . 3
1.2 Objectives . . . 3
1.3 Approach . . . 3
1.4 A Brief Overview of Previous Work . . . 4

II Audio Processing 6

2 Introduction to Speech Analysis 7

3 Human Speech 8

3.1 Introduction . . . 8
3.2 Wave Theory . . . 8
3.3 Noise . . . 10
3.4 The Human Vocal System . . . 10
3.4.1 Voiced Sounds . . . 11
3.4.2 Unvoiced Sounds . . . 12
3.4.3 Classifying Sounds as Voiced or Unvoiced . . . 12
3.5 Phonemes . . . 13
3.5.1 Classification of Phonemes . . . 14
3.6 Graphs of Speech . . . 17
3.6.1 Wave Patterns . . . 17
3.6.2 Spectrogram . . . 18

3.6.3 Formants . . . 20
3.7 Co-articulation . . . 21
3.8 Additional References . . . 21
3.9 Summary . . . 22

4 Digital Signal Processing 23
4.1 Introduction . . . 23
4.2 Sampling . . . 23
4.2.1 Sampling Problems . . . 24
4.2.2 Sampling Theorem . . . 25
4.3 Energy . . . 26
4.4 Digital Filters . . . 27
4.4.1 How Digital Filters Work . . . 27
4.4.2 Notation of Digital Filters . . . 28
4.5 Mathematical Foundations for DSP . . . 30
4.5.1 The Geometry of Functions . . . 30
4.5.2 The Inner Product . . . 31
4.5.3 Spanning the L_2 Space . . . 34
4.5.4 The Importance of Orthogonality . . . 35
4.5.5 Raising e to a Complex Power . . . 36
4.5.6 The Delta Function . . . 36
4.6 Change-of-Base Transforms . . . 37
4.6.1 The Reason for Doing Change-Of-Base Transforms . . . 38
4.7 The Fourier Transform . . . 39
4.7.1 Introduction to the Fourier Transform . . . 39
4.7.2 The Discrete Fourier Transform . . . 40
4.7.3 The Fast Fourier Transform . . . 41
4.7.4 Other FFT Algorithms . . . 43
4.7.5 Problems with Fourier Transforms . . . 44
4.8 Wavelets . . . 46
4.8.1 A Brief History of Wavelets . . . 46
4.8.2 Understanding Wavelets . . . 47
4.8.3 The Mathematics of Wavelets . . . 49
4.8.4 The Discrete Wavelet Transform . . . 51
4.8.5 More Useful Wavelets . . . 53
4.9 Feature Extraction Techniques . . . 54
4.9.1 Processing of Fourier Data . . . 54
4.9.2 Processing of Wavelet Information . . . 55
4.10 Frames and Windows . . . 56

4.11 Additional DSP References . . . 58
4.12 Summary . . . 58

5 Classification 59
5.1 Introduction . . . 59
5.2 Neural Networks . . . 60
5.2.1 Training Neural Networks . . . 63
5.2.2 Identifying Phonemes Using Neural Networks . . . 67
5.2.3 Additional References . . . 68
5.2.4 Summary . . . 68
5.3 Hidden Markov Models . . . 69
5.3.1 How Hidden Markov Models Work . . . 69
5.3.2 HMM Notation . . . 70
5.3.3 Solving HMM Problems . . . 71
5.3.4 The Three HMM Problems . . . 73
5.3.5 Applying HMMs to Speech Recognition . . . 77
5.3.6 Recognising Words using HMMs . . . 78
5.4 Other Classification Techniques . . . 79
5.4.1 Additional References . . . 79
5.5 Summary . . . 79

6 Speech Recognition Challenges 80
6.1 Acquiring Training Data . . . 80
6.2 Dealing with Incorrectly Labelled Phonemes . . . 81
6.3 Phoneme Borders . . . 81
6.4 Summary . . . 82

III Facial Rasterisation 83

7 Introduction 84

8 Facial Modeling 86
8.1 Different Ways of Modeling Faces . . . 86
8.1.1 Performance-Based Models . . . 87
8.1.2 Parameterised Models . . . 89
8.1.3 Muscle Based Models . . . 93
8.1.4 Manipulating Skin Vertices Based On Muscle Movements . . . 94
8.2 Automated Model Creation . . . 98
8.3 An Alternative To Anatomical Correctness . . . 98
8.4 Additional References . . . 99

8.5 Summary . . . 99

9 Techniques For Improving Rasterisation 100

9.1 Shading Techniques . . . 101

9.1.1 Flat Shading . . . 101

9.1.2 Gouraud Shading . . . 101

9.1.3 Phong Shading . . . 102

9.1.4 PN Triangles . . . 103

9.1.5 Shading Techniques for Facial Animation . . . 105

9.2 Advanced Texturing . . . 105

9.2.1 Bump Mapping . . . 106

9.2.2 Height Maps . . . 106

9.2.3 Dot 3 Bump Maps . . . 107

9.2.4 Displacement Maps . . . 107

9.2.5 Calculating Bump Maps . . . 107

9.2.6 Self Shadow . . . 108

9.2.7 Light Mapping . . . 108

9.2.8 Gloss Mapping . . . 108

9.3 Anti-Aliasing . . . 109

9.4 The Cg Language . . . 110

9.5 Summary . . . 110

10 Video Realism 112

10.1 Introduction . . . 112

10.2 Motion Capture . . . 112

10.3 Physics . . . 114

10.4 Co-articulation . . . 115

10.5 Summary . . . 116

IV Implementation 117

11 Implementation 118

11.1 Digital Signal Processing . . . 118

11.1.1 Sampling . . . 118

11.1.2 Signal Processing . . . 119

11.1.3 Classification . . . 120

11.1.4 Training . . . 121

11.2 Modelling . . . 121

11.3 Rasterisation . . . 122

11.4 Driving the Model Parameters 124

12 The Prototype System **125**

12.1 The Overall Design 125

12.2 Problems Encountered 125

12.3 Facts about the Prototype 126

13 Results **128**

13.1 System Specification 128

13.2 Model Information 128

13.3 Phoneme Recognition Performance 129

13.3.1 The DWT 129

13.3.2 Classification results of log cepstral coefficients 130

13.4 Mesh Manipulation Performance 130

13.5 Conclusion 131

14 Conclusions and Contributions **134**

14.1 Speech Classification 134

14.2 Facial Modeling 135

14.3 Entire Solution 135

15 Additional Applications **137**

15.1 Speech Recognition Techniques 137

15.1.1 Speaker Identification 137

15.1.2 Speech Samples for Speaker Identification 138

15.1.3 Reliability of Speaker Identification 138

15.2 Wavelet Transforms 139

A Data Structures Used **142**

A.1 Windowing Algorithm 142

A.2 Fast Fourier Transform 143

A.3 Cepstral Coefficients Algorithms 146

A.4 Neural Network Data Structure 148

A.5 Phoneme Identifier 152

A.6 Geometry Structures 153

A.7 Mesh Structure 156

A.8 Skeleton Data Structures 158

A.9 Muscle Data Structures 159

References **162**

List of Figures

3.1 A low frequency sound wave 9
3.2 A high frequency sound wave 9
3.3 The superposition of two sound waves 9
3.4 Voiced sound wave 11
3.5 Unvoiced sound wave 12
3.6 Monophthong sound wave 15
3.7 Diphthong sound wave 15
3.8 Approximant sound wave 15
3.9 Nasal sound wave 16
3.10 Fricative sound wave 16
3.11 Plosive sound wave 16
3.12 Africate sound wave 17
3.13 A wave-graph (phonetics produced by Microsoft Liset [60]) 18
3.14 A spectrogram for the word “election” 19

4.1 Undersampling 24
4.2 Periodic undersampling 25
4.3 Raising e to a complex power 36
4.4 Symmetry in fourier transforms 41
4.5 The working of the Cooley-Tukey FFT algorithm 43
4.6 Frequency leakage 45
4.7 The difference between Fourier and Wavelet transformations 48
4.8 The effects of the Blackman windowing function [5] 58

5.1 A neural network structure 62

8.1 The effect of a muscle on vertices - [93] 95
8.2 Muscle vector parameters 96

9.1 Normal calculations for shading techniques 102
9.2 Flat shading 102
9.3 Gouraud shading 103

9.4 Phong shading 103
9.5 Subdividing a PN triangle 104
9.6 Curving the PN triangles 104
9.7 Output from PN triangles 105
9.8 Aliasing 109
9.9 Anti-aliasing 110

10.1 Exploring facial performance capture 113

11.1 Implication of normals on shading 123

15.1 The original fingerprint [8] 140
15.2 The fingerprint compressed using the JPEG standard [8] 141
15.3 The fingerprint compressed using WSQ [8] 141

List of Tables

3.1 The American English phonemes 14

4.1 Bit-reversing indices 42
4.2 Definitions of some common windowing functions 57

5.1 Activation functions 63
5.2 HMM state transition matrix 70
5.3 HMM observation matrix 70
5.4 HMM state sequence probabilities 72

13.1 The accuracy of phoneme identification using the DWT 130
13.2 The effect of cepstral coefficient counts 131
13.3 The effect of training data composition 132
13.4 The speeds of the different modules 132
13.5 The effect of polygon count 132
13.6 The effect of muscles 133

Part I

Introduction

Chapter 1

Introduction to Audio Driven Facial Animations

One very profitable industry in the world today is the entertainment industry. Companies seeking an edge on competitors are continually looking for innovative and creative new techniques to communicate their messages. One of the more popular techniques employs computer animation to convey the message.

Computer animation for entertainment is not a new idea - even early movies sometimes used computers to generate special effects like fireballs and spaceships. Over time the quality of these pre-made animations has been steadily improving. This can be observed by movies like ‘Lord of the Rings’, where it is nearly impossible to differentiate between reality and animation. One application of computer animation in the entertainment industry that has been less used is real-time animation.

Real-time animation is computer animation, but the models being animated are controlled at the time they are displayed. A prime example is a model interacting with a live audience. In traditional animation (off-line animation) it is possible for an artist to perfect aspects such as motion and shape before rendering. Very often several cycles of rendering → reviewing → adjustment to models, may be required before the artist is satisfied with the animation. In a real-time environment this is not possible - the movements of the model must be accurately, flexibly and quickly controllable. Also the rendering process itself must be real-time.

When animating the facial features of a character, the spoken message of that character is conveyed more clearly when one can see the actual lip movements of that character [3]. In off-line scenarios, a recorded sound file is analysed in detail, to identify segments of speech. This information is then applied to a model’s face so that it appears that the model is speaking the contents of the sound file. This analysis process may

take quite a while and require several passes. Furthermore, the entire sound file is on hand at the time of animation. This is not the case in a real-time environment. The requirement for real-time facial animation is that the model appears to speak the sounds *while* those sounds are being recorded and output - there can be no (or very little time) for pre-processing of this sound information.

1.1 Problem Statement

The problem being solved in this document is to determine whether real-time human speech can be effectively used to drive a facial animation.

1.2 Objectives

Our objectives are:

- To determine whether real-time human speech can be used to drive a facial animation.
- To build a prototype system to demonstrate this.
- To investigate ways to improve the performance of such a system.

1.3 Approach

In order to solve the above-mentioned problem, we utilise the following methods:

1. We first conduct a literature survey to determine to what extent other work has solved the problem. This includes real-time scenario and offline scenario work.
2. We also use our findings from the literature survey to determine which of the techniques that could be used provide the most benefits (accuracy, speed, effectiveness and such).
3. Having completed our survey, we progress to design and build a prototype to demonstrate our findings.
4. Once a working prototype has been completed, we work to implement some of the potential speed, accuracy and effectiveness enhancements identified in the literature survey.

1.4 A Brief Overview of Previous Work

In order to accomplish interactive audio-driven facial animation there are several aspects one must consider:

1. Audio-analysis.
2. Facial animation.
3. Making it look appealing.
4. Doing it in real-time.

Audio analysis is a well defined area of research. Early techniques included reading spectrograms [15], Fourier Transforms [45] and Wavelet transforms [35] applied together with classification systems.

Facial animation is also a mature field of research. In the early days, creative people's hand-drawn artwork was used [24]. Later people learnt to create animations by stitching together appropriate photographic still images. One example of this approach is described in [30]. In later years still images were morphed into one another to correct minor defects in the animation and to improve the synchronisation of the animation with the audio channel [31]. 3D Rendered models have also been applied, sometimes to great effect. Keith Waters [93] describes using muscle simulations to animate a model of a human face, and later work expanded on many of his original ideas.

Advanced texturing and rendering techniques [63] have also allowed the quality of rendered models to be vastly improved. Many of these techniques can even be applied in real-time rendering scenarios by dedicated 3D hardware [59].

Different combinations of these above requirements have been achieved with great success: real-time facial animation (items 2 and 4) [99], photorealistic facial animations (items 2 and 3) [98], and even audio-driven facial animation (items 1 and 2) [34].

Putting them *all* together has been achieved, but with marginal success. The hindrances of real-time speech analysis are probably the largest stumbling block. The reason for this is that sounds that are *about* to be spoken have an effect on sounds *currently* being spoken [18].

This document studies each of the above-mentioned problems in detail:

- Human speech (chapter 3) - we describe human speech, illustrating key factors to consider at audio processing time.

- Digital Signal Processing (chapter 4) - this section describes the techniques used to analyse digital signals (in specific audio signals), together with optimisations that allow for feature extraction in real-time.
- Classification (chapter 5) - the state-of-the-art techniques for classifying data, specifically human speech are studied.
- Facial Modelling (chapter 8) - different modeling techniques and their advantages and disadvantages are shown in this section.
- Techniques for Improving Renderings (chapter 9) - this section describes some of the more popular techniques that can (a) be used for facial animation and (b) that can be executed in real-time, that contribute to improving the overall quality of rendered animation.
- Video Realism (chapter 10) - in this section techniques are described which improve the quality of the movement of the models to be animated. Co-articulation effects (one of the major drawbacks to real-time facial animation) is one of the major focuses of the chapter.

For each section an attempt is made to show how the technique is applied in a real-time environment.

The document also includes a description of our implementation of a real-time facial animation system (chapters 11 and 13). The last chapter then describes future research that could be conducted.

Part II

Audio Processing

Chapter 2

Introduction to Speech Analysis

When we hear human speech, our brains process the audio information in many ways before we actually *understand* what is being said. Our brains are however extremely complex and can be compared to a vast system of parallel computers all doing their share of work at the same time. Modern computers still primarily rely on a single CPU that can do only one instruction at a time. The problem is that the computer must still do all the same work as the human brain if it aims to gain the same accuracy of understanding of speech. For purposes of this document we need not comprehend the meaning of the speech itself, but we do need to process it sufficiently to identify the words (or at least sub-words or phonemes) being spoken.

This part of the document will explain:

- Human Speech (chapter 3) - This chapter shows how the human vocal system produces sounds. This will help us to identify desired features of sounds and potential classifications of sounds. It also provides some of the history of our modern speech-related knowledge including speech-graphs.

- Digital Signal Processing (chapter 4) - This chapter deals with processing the digital signal (recorded spoken sound) in various ways. Focus is placed on extracting important features of the signals for use in a classification system. Performance details are explained as well, as these play a large role in performing the calculations in real-time.

- Classification (chapter 5) - This chapter deals with using artificial intelligence and learning techniques to classify audio features.

- Speech Recognition Challenges (chapter 6) - This chapter explains some obstacles that one can expect to encounter when writing speech recognition applications.

Chapter 3

Human Speech

3.1 Introduction

In order to accurately analyse speech it is important to first understand how speech is produced. This chapter will describe the various parts of the human vocal system and the roles they play, as well as providing an important introduction to audio theory. This information helps once we start using some digital signal processing techniques (which are described in chapter 4).

Firstly, the theory of sound will be briefly explained, as this information is crucial to anybody undertaking speech analysis at any level.

3.2 Wave Theory

Sound vibrations are caused by some disturbance in a medium (air). These vibrations travel in a wave-like motion through this medium. Should the vibrations reach our ears, we hear the sound. The higher the frequency of the vibrations, the higher the pitch of the sound. The higher the amplitude of the vibrations, the louder the sound. Several sound waves passing each other are superimposed. (See figures 3.1 to 3.3).

When one hears sounds, one is really hearing the sum of many vibrations that have reached our ears. Any given sound may be composed of many different frequencies and amplitudes of vibrations. The ear separates these superimposed vibrations into their component vibrations. The way this works is as follows: inside the human ear there is a spiral structure called the cochlea. Inside the cochlea is a membrane (called the basilar membrane) that resonates as the sound causes it to vibrate. High pitch sounds cause the front of the membrane to resonate, while lower pitch sounds cause it to resonate nearer to the back. Tiny hair-like cells sense this resonance and convey the message to the

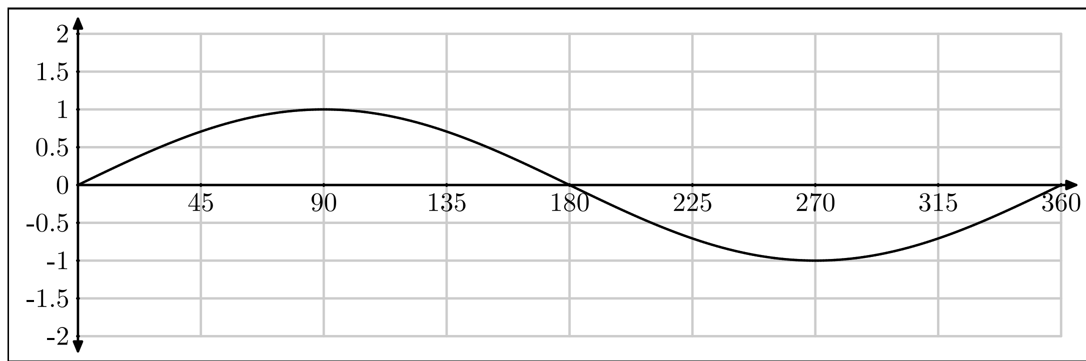


Figure 3.1: A low frequency sound wave

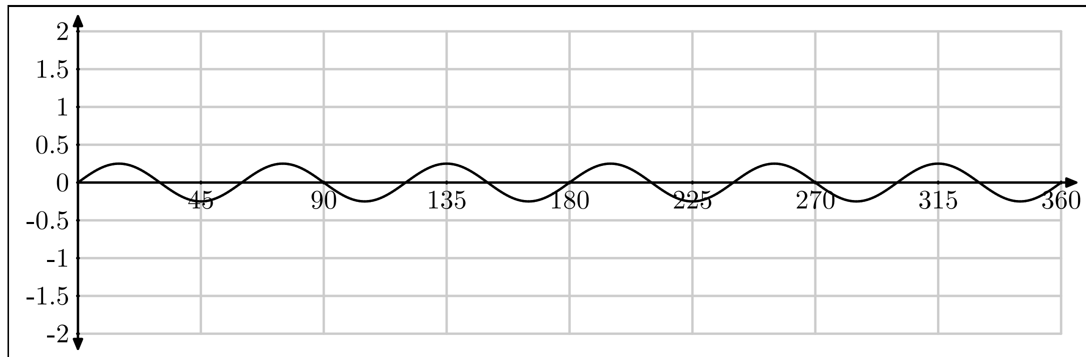


Figure 3.2: A high frequency sound wave

brain via the auditory nerve. The intensity with which the hair vibrates is determined by the amplitude of the vibration. This process is simulated mathematically by Fourier Transforms (chapter 4.7). The brain interprets these signals and we ‘hear’ the sound.

We have seen in this section how the ear interprets sounds, but we need to be aware that our brains ‘filter’ out a lot of unimportant sounds (noise).

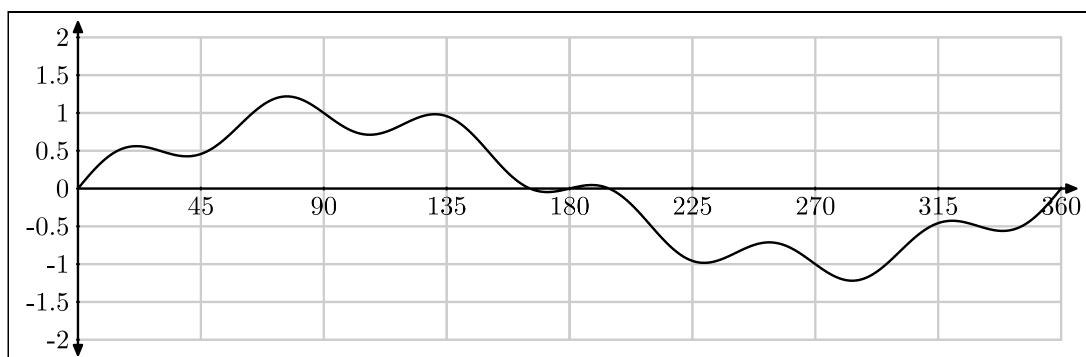


Figure 3.3: The superposition of two sound waves

3.3 Noise

Unfortunately the ear hears all sounds entering it, not just the ones that are important. Noise can be superimposed with the original vibration, and would then be heard by the ear as well. The brain fortunately is extremely complicated and has been trained over years to ignore the noise. This is more difficult for a computer to accomplish.

Noise needs to be mathematically defined before it can be filtered out. If the noise signal can be predicted, then a simple subtraction of the noise from the signal would result in the uncorrupted signal, but this case of prediction is extremely rare in real-world scenarios. Noisy information occurs with varying pitches and intensities, often at unpredictable times. To eliminate this noise therefore usually involves searching for chaotic features in the sound signal and using filtering algorithms to rebuild the original signal without these features.

By gaining an understanding of the specific source of the original sounds (for example knowing that human voice ranges between 300 and 3400Hz [6]), one can filter out sounds that are undesirable.

One effective way of coping with noise is to pass the audio signal through dedicated hardware filters that remove the noise, but in this document we focus on using software for this task.

We have now seen how sounds are transmitted and perceived. We have also described that understanding the source of the sounds can help filter out noisy information. In the next section, the source of human speech sounds will be explained: the human vocal system.

3.4 The Human Vocal System

As mentioned in the previous section, knowledge of the source of a sound signal can help filter out noisy information from that signal. Some of the other techniques (Linear Predictive Coding inter alia) used to identify features of the signal are also enhanced by an understanding of the actual workings of the human vocal system.

By knowing how the different parts of the human vocal system create the vibrations in the air, we can calculate what sounds to expect. These sounds have certain properties by which they can be identified. This in turn helps to classify the individual sounds.

The human vocal tract is a complex set of structures each of which cause different vibrations and disturbances in the air as the air passes them. These structures are:

- the nasal cavity,
- the mouth cavity,
- the tongue,
- the glottis (voice-box),
- the trachea and lungs and
- the velum (soft palate at the back of the mouth).

Each of these structures adds certain features to the sound. One of the most important distinctions in sound classification is whether the sound is voiced or unvoiced - ie. whether the vocal chords vibrate or not.

3.4.1 Voiced Sounds

Voiced sounds are caused by air from the lungs flowing over the vocal cords. This causes vibrations of a specific frequency in the air (ie. a very regular vibration in the air is created). These sounds are known as glottal pulses - so called because they are produced by the glottis (voice-box) in the throat. They are usually vowels of high energy levels and have distinct formant frequencies [6] (formants are described in section 3.6.3). This frequency can be clearly seen in figure 3.4. For now all we need know about formants is that the vibrations causing them have large amplitudes and are thus easily identified from the sampled wave.

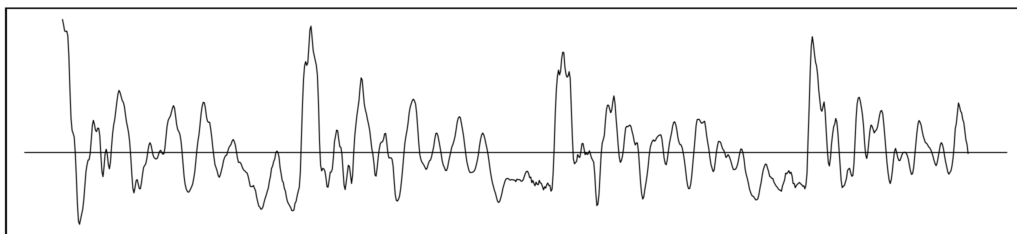


Figure 3.4: 850 samples of the voiced sound ‘ao’ (22050 samples/sec)

Not all sounds are caused by the glottis - in the next section another class of sounds called unvoiced sounds will be described.

3.4.2 Unvoiced Sounds

In the previous section we saw that vowels and similar sounds are caused by air flowing over the vocal chords, and the resulting vibrations were regular. Some structures cause a more chaotic set of vibrations. These are known as unvoiced sounds and are typically created by turbulence in the air flowing through the vocal tract. Examples are:

- air flowing through the tight space between the tongue and the teeth - eg. the ‘s’ sound,
- air flowing through the space between the tongue and the roof of the mouth - eg. the ‘sh’ sound,
- air flowing through the constriction in the lips while whistling and
- explosions of air that is totally trapped by obstructions in the vocal tract being released - eg. the ‘p’ sound.

According to Bradbury [6], pitch is not an important characteristic of unvoiced sounds. This is due to the fact that the turbulent air produces vibrations of many different frequencies, typically each of which are of a relatively low amplitude. This can be seen in figure 3.5.

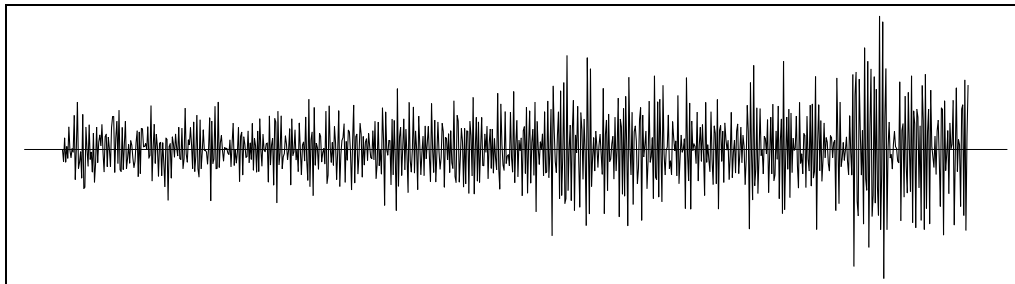


Figure 3.5: 850 samples of the unvoiced sound ‘s’ (22050 samples/sec)

One of the first steps of speech classification is usually to distinguish between the two above-mentioned classes of sounds. This process will be explained next.

3.4.3 Classifying Sounds as Voiced or Unvoiced

In the previous two sections, we classified sounds into voiced and unvoiced categories. This is the first step of a hierarchy of classifications. As a result we need to first identify in which of these classes a sound belongs.

It is relatively easy to classify sounds as voiced or unvoiced (compared to the rest of the

analysis process at any rate). By observing the number of times the wave signal crosses the x-axis (see figures 3.4 and 3.5) we can get a fairly accurate indication whether or not the sound is a voiced sound. In our experiments we found that with a sample rate of 22050 samples / second - taking 4096 consecutive samples resulted in an average of 607 crossings of the x-axis given unvoiced input as opposed to 197 crossings with voiced input. The reason behind this observation is that unvoiced sounds result in higher frequencies which would in turn cause the signal to cross the axis more frequently.

Using this metric results in a fairly definite classification that is irrespective of sound volume (sound volume is another one of the metrics that is sometimes used to determine whether a sound is voiced or unvoiced).

In the past few sections we have shown two classes of sounds and how signals are placed into these two classes. In the next few sections we will go one step further and describe common sub-classifications for both voiced and unvoiced sounds.

3.5 Phonemes

The most common way to classify human speech is to break up the sounds into *phonemes*. A phoneme is considered a basic unit of human speech [7, 26, 35]. To see an example of phonemes, one can observe phonetic spellings for words. Phonetic spelling shows how words are pronounced by showing the set of phonemes that make up a word. Each language has their own set of phonemes - American English consists of the phonemes listed in table 3.1. For example, the word ‘phoneme’ spelt using phonetic spelling is “/f/ /ow/ /n/ /iy/ /m/”. Sometimes different symbols are used to represent these letters also, such as using a horizontal bar over a letter.

There is a set of symbols which represent the phonemes used not only in English but in all commonly spoken languages. This is called the International Phonetic Alphabet.

As seen in section 3.4 phonemes are first classified as voiced and unvoiced. Dividing sounds into these categories is an important part of the analysis process, as the two sounds exhibit different features due to the way they are formed, but this is insufficient. Sounds within these categories needs to be further classified as described in the following section.

	Symbol	Example
1	-	Syllable boundary
2	!	Sentence terminator
3	&	Word boundary
4	,	Sentence terminator (comma)
5	.	Sentence terminator (period)
6	?	Sentence terminator (question mark)
7	_	Silence
8	1	Primary stress
9	2	Secondary stress
10	aa	father
11	ae	cat
12	ah	cut
13	ao	dog
14	aw	foul
15	ax	ago
16	ay	bite
17	b	big
18	ch	chin
19	d	dig
20	dh	the
21	eh	pet
22	er	fur
23	ey	ate
24	f	fork
25	g	gut
26	h	help
27	ih	fill
28	iy	feel
29	jh	joy
30	k	cut
31	l	lid
32	m	mat
33	n	no
34	ng	sing
35	ow	go
36	oy	toy
37	p	put
38	r	red
39	s	sit
40	sh	she
41	t	talk
42	th	thin
43	uh	book
44	uw	too
45	v	vat
46	w	with
47	y	yard
48	z	zap
49	zh	pleasure

Table 3.1: The American English phonemes

3.5.1 Classification of Phonemes

Phonemes are initially classified into voiced or unvoiced [15], which approximately map to the concept of vowel and consonant respectively.

Vowels are further classified into monophthongs and diphthongs. Monophthongs are vowels where the sound remains relatively unchanged (example ‘oo’ in ‘b*oo*k’), while diphthongs are vowels that change as the vowel is spoken (example ‘i’ in ‘b*i*te’).

If one looks at figure 3.6 one can see that the wave pattern remains relatively unchanged while speaking a monophthong compared to figure 3.7 where the shape of the wave changes as a diphthong is spoken.

Consonants are divided into 5 sub-categories:

- Approximants - these sounds are midway between vowels and consonants. This is caused by restriction in the vocal tract. Look at figure 3.8 to observe the vowel

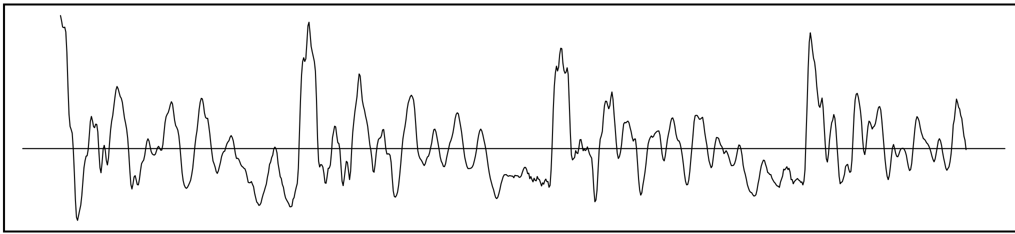


Figure 3.6: 850 samples of the monophthong 'ao' (22050 samples/sec)

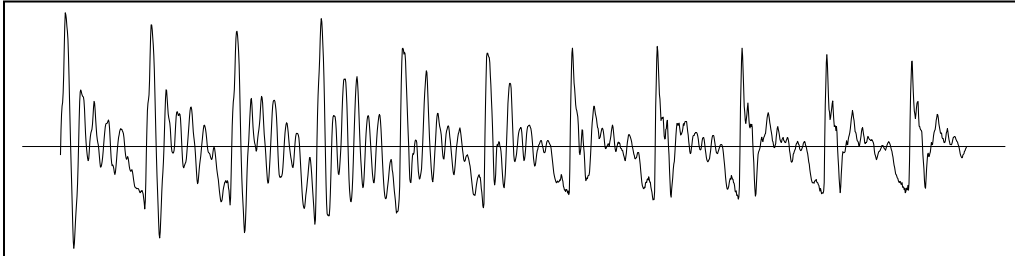


Figure 3.7: 1700 samples of the diphthong 'oy' (22050 samples/sec)

features (low frequency wave) together with the consonant features (high frequency wave). Examples of approximants are '*w*' and '*l*' sounds.

- Nasals are produced by air being completely blocked from escaping from the mouth (either by the tongue, lips or velum), thereby forcing the air out of the nasal cavity. Examples of nasals are '*n*', '*m*' and '*ng*'. By studying figure 3.9, one can see that nasals cause an additional, lower-amplitude, double frequency wave to the standard wave caused by the glottis (voice box). This is noticeable by looking at the smaller spike roughly half-way between the tallest spikes in the figure.

- Fricatives are caused when there is severe turbulence in the air passing through the vocal tract. This is due to the sound articulators being very close to one another. Sounds like '*s*' and '*h*' are examples of fricatives. Due to the severe turbulence in the air, the resulting wave has a broad range of frequencies, many of which

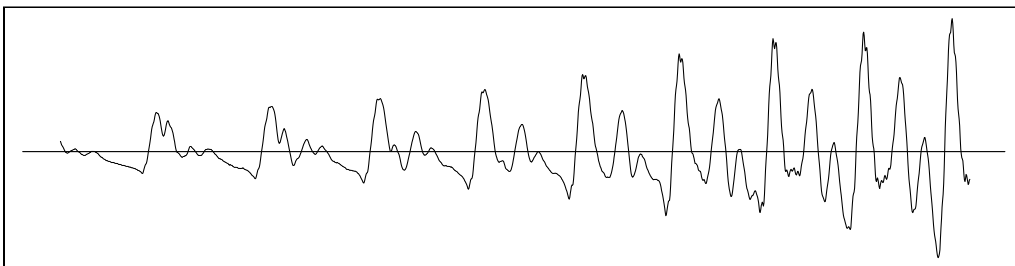


Figure 3.8: 1700 samples of the approximant 'w' (22050 samples/sec)

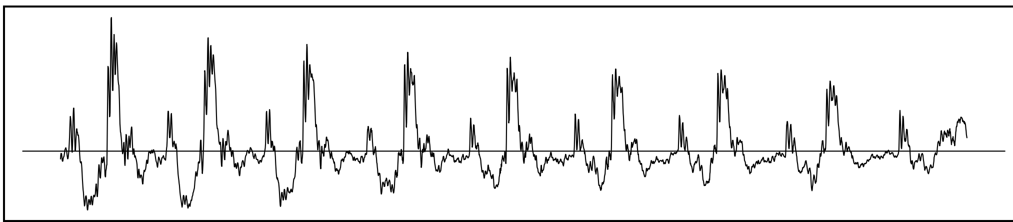


Figure 3.9: 3400 samples of the nasal ‘n’ (22050 samples/sec)

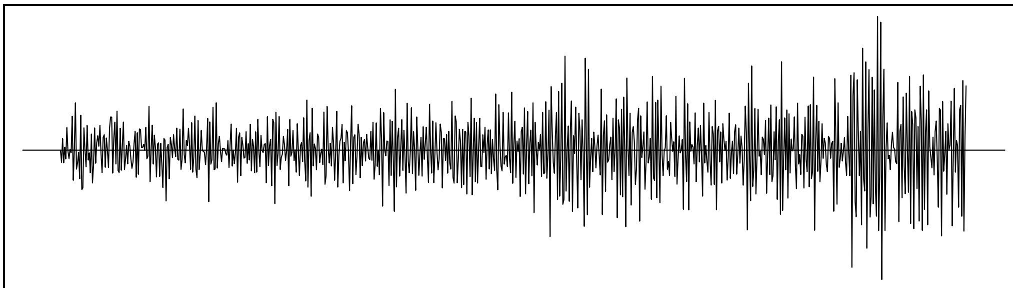


Figure 3.10: 850 samples of the fricative ‘s’ (22050 samples/sec)

are very high. This can be seen in figure 3.10. The fricative *h* is also called an *aspirate* because of the puff of air released while saying it (to aspirate means to breathe). Another example of an aspirate is the *t* in the word ‘kit’, because after saying the ‘*t*’ there is an extra release of air.

- Plosives are caused by a complete sealing of the vocal tract for a short period of time, causing air pressure to build up, then folowed by a sudden and complete release of the air. This creates an explosive sound. The English language has six plosives each of which are caused by blocking off different volumes of the vocal tract or releasing air at different pressure. These are *‘p’, ‘t’, ‘k’, ‘b’, ‘d’, ‘g’*. These six plosives can be divided into two groups - ‘p’, ‘t’ and ‘k’) and (‘b’, ‘d’ and ‘g’). The latter group builds up less pressure in the vocal cavity than their counterparts (which block off the same volume of air) in the former group. By looking at the plosive in figure 3.11, one can see how the sound is restricted (no vibrations) then suddenly released (sudden start to the vibrations). The largest

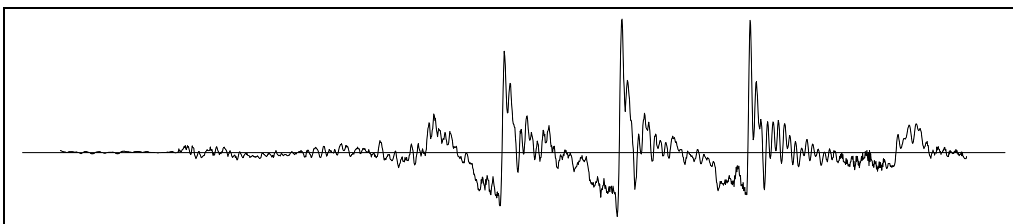


Figure 3.11: 1700 samples of the plosive ‘p’ (22050 samples/sec)

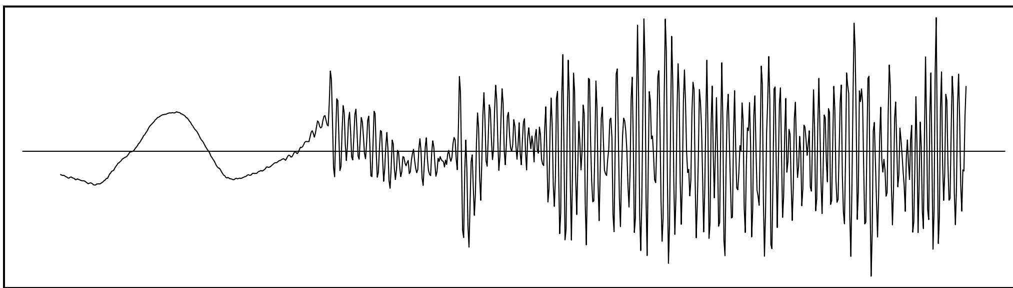


Figure 3.12: 850 samples of the affricate ‘j’ (22050 samples/sec)

amplitude vibrations only last until the air has been released, then the intensity of the vibrations drops.

- Affricates are similar to plosives except that when the air is finally released, the vocal tract still remains slightly obstructed. The phonemes ‘*j*’ and ‘*ch*’ are the only two affricates in the English language. By looking at figure 3.12 one can see how the air is initially trapped (very little vibration) followed by a turbulent (high frequency wave) release of air.

Understanding what phonemes are and how they are produced is only part of the knowledge we need to be able to classify sound into phonemes. In the next section some graphs that are studied by human sound-analysers will be shown. These graphs are extensively used to gain better understandings of sound features (one of the most noteable being formants).

3.6 Graphs of Speech

In this chapter we have described the different aspects that make up sounds focussing on human speech. We mentioned that analysis techniques can be improved if we have knowledge of the signals being analysed. We continue in this section by showing some of the graphs that are used to gain a better understanding of sound features, to aid in the analysis process.

The specific graph used depends on the features that should be highlighted. This section will explain the roles of the different graphs that are commonly used.

3.6.1 Wave Patterns

Wave patterns are the easiest graph to obtain, as no processing is required. The samples from the microphone are simply plotted. The x-axis is time, and the y-axis is amplitude. One will notice that the samples have both positive and negative values. (One can think

of these samples as electrical current passed to an electromagnet in a hi-fi speaker. As these samples go above and below zero this would cause attractive and repulsive forces between the electromagnet and the permanent magnet attached to the speaker membrane - thus producing vibrations in the membrane and hence sound).

By studying the wave patterns, one can identify certain patterns in the sound with the naked eye. If one looks at figure 3.13, one can see that the unvoiced sound ‘ch’ from the word “each” has high frequency with low amplitude, while the ‘ea’ sound has a lower, less chaotic frequency, with a higher amplitude.

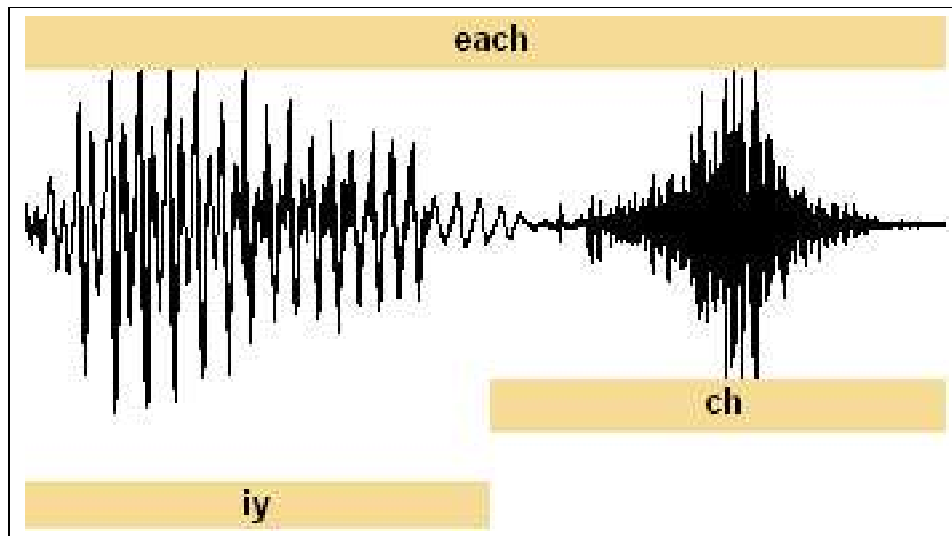


Figure 3.13: A wave-graph (phonetics produced by Microsoft Liset [60])

Unfortunately the patterns which we can identify easily with our brains are much harder to spot using purely mathematical techniques. For this reason very little success has been achieved by analysing the wave pattern alone. There are also other graphs which can highlight more obscure features. The next section will describe one such graph: the spectrogram.

3.6.2 Spectrogram

Speech analysis is frequently aided by using a spectrogram as the simple wave data (mentioned in the previous section) seldom reveals enough by itself. This is a graph where the x-axis is usually time, the y-axis is frequency, and the amplitude of the component of the wave of the given frequency is represented by the brightness (or darkness) of the dot (pixel) on the graph. The louder the sound at a given frequency and time, the brighter (or darker - depending on the graph palette) the appropriate pixel would be. At the top of figure 3.14), is a spectrogram of the word “election”. Notice the darker

coloured linear patches near the bottom on the left. These lines are high-energy areas representing formants (see next section) of the vowels ‘ih’ and ‘eh’ shown in the wave-graph below the spectrogram.

Often a spectrogram will use the logarithm of the energy of the sound at the given frequencies as opposed to the energy itself. This is known as the *decibel* scale. As with

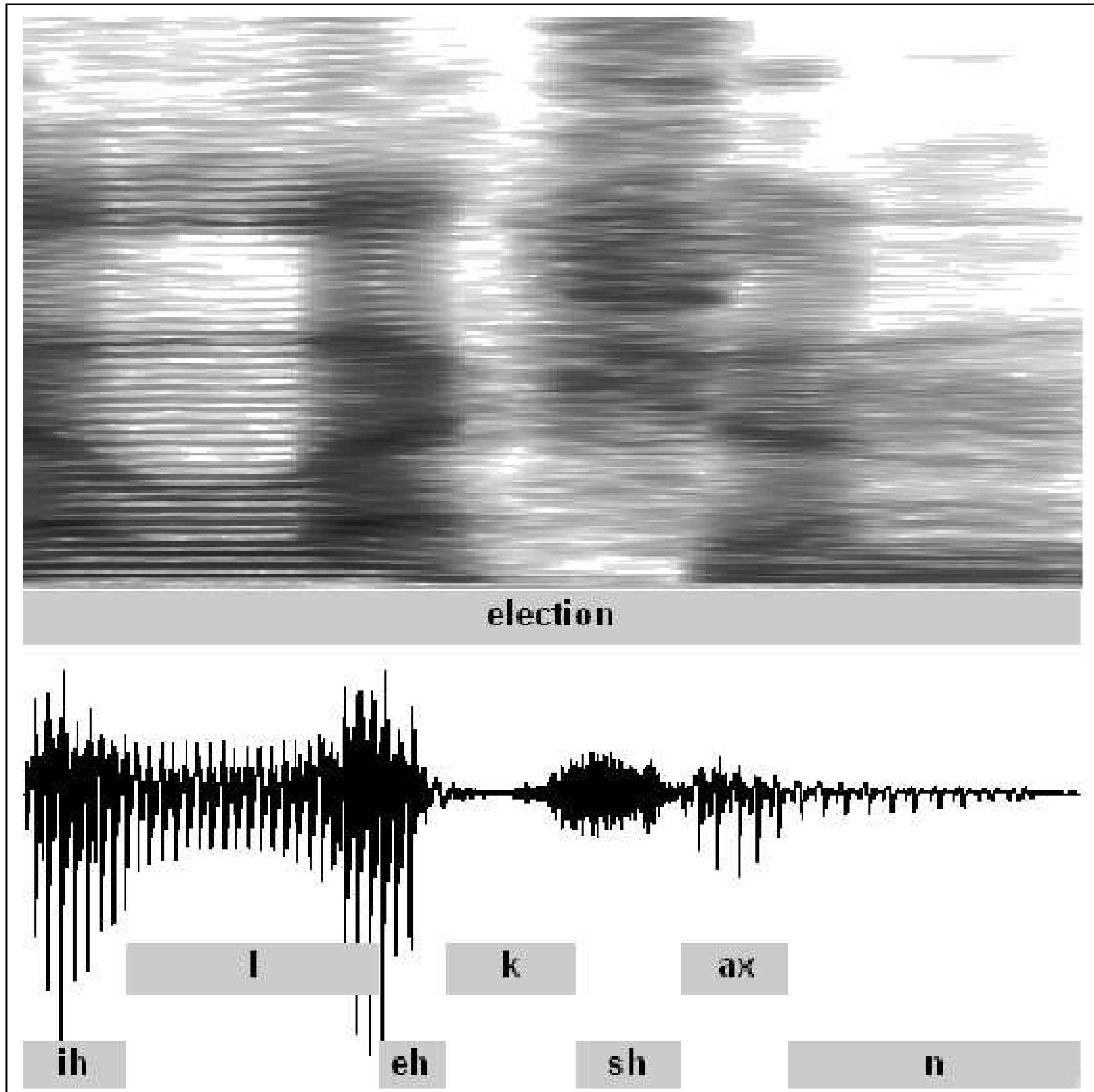


Figure 3.14: A spectrogram for the word “election” (with phoneme breakdown produced by Microsoft Liset [60])

most sound analysis techniques, the spectrogram relies on the Fourier Transform (see chapter 4.7 for more information about the Fourier Transform).

3.6.3 Formants

As mentioned previously in this chapter, phonemes are initially classified as voiced or unvoiced phonemes. The reason for this is that the two are analysed differently: voiced sounds, when viewed on a spectrogram exhibit high amounts of energy clustered around certain frequency levels, while unvoiced sounds exhibit more of a general low-intensity cloud of energy spread across many different frequencies [15]. (To observe this phenomenon notice the dark lines near the bottom of the spectrogram in figure 3.14 while the two vowels '*ih*' and '*eh*' are spoken). In this section we will focus on the features of voiced sounds: formants. These are the clusters of high energy (observed on a spectrogram) for voiced sounds.

Notice that there may be more than one formant at a given point in time. This is observed in the dark patches on the spectrogram above one another. These formants are numbered starting at one (F1) - the lowest frequency formant, and increasing in number as the frequency range increases (F2, F3, ...).

F1 ranges from 300Hz to 1000Hz, and is caused by proximity of the tongue to the roof of the mouth. The closer the proximity, the higher the pitch of the formant.

F2 ranges from 850Hz to 2500Hz, and is caused by the position of the tongue from the back of the mouth together with the rounding of the lips. The further back in the mouth the tongue, the smaller the oral cavity, the higher the pitch of the formant. Rounding the lips lowers the pitch of F2.

The first two formants are the most important in identifying phonemes. The latter formants are primarily used to determine quality of the phoneme [15].

Due to the chaotic nature of unvoiced sounds, there is no unvoiced counterpart to formants. One must use the properties described in section 3.5.1 to categorise unvoiced sounds.

In this section we looked at different graphs that can help us analyse sound signals. We conclude this chapter by studying the effects that one phoneme has on those spoken just before or after that phoneme: co-articulation.

3.7 Co-articulation

Up until now in this chapter we have studied phonemes which we assumed to be atomic, ie. they could be totally isolated from sounds adjacent (in time) to them. This is unfortunately only an approximation - in practise there is interference between phonemes called co-articulation [18], [67].

Co-articulation is the interpolation on the vocal tract shape between uttering one phoneme and uttering the next one [3]. What this means is that the sounds one produces before and after any sound, influence the mouth position of that sound. An example given by Albrecht et al. [3] is in the mouth shape during the '/k/' sound in the words "coin" and "cow". In the first instance the mouth is round, while in the second instance the exact same sound is formed by a wide-open mouth. It can therefore be seen that the sounds (the '/oi/' and '/ow/') adjacent to a given sound (the '/k/') influence the mouth (and the rest of the vocal tract) shape for that given sound.

Co-articulation is one of the greatest hurdles to cross in real-time facial animation, because the phonemes that a person is about to say (in the future) will influence their mouth shape during phonemes they are currently saying. For this reason one of two courses of action needs to be taken (a) ignore the sounds following a given phoneme, and use exclusively those sounds preceding the one currently being uttered (to predict co-articulation effects), or (b) delay output of the sound by some relatively short time, say $\frac{1}{2}$ second.

Solution (a) would lead to inaccuracies and should only be applied when real-time is essential (as opposed to mere continuous-time). If this inaccuracy is tolerable then this method is suitable. Solution (b) would result in more accurate predictions of mouth positions at the expense of real-time output. If a delay in output is acceptable then this solution is the more suitable of the two.

In section 10.4 we will discuss more about co-articulation, focusing on the effect it has on video-realism.

Additional resources about human speech are provided in the next section.

3.8 Additional References

Speech analysis is a vast field that spans many disciplines including medical, psychological, language and several others. As such one can find a lot of information about

specific details of speech recognition. For more tutorial information about speech analysis see [37] and [52].

3.9 Summary

In this chapter we discussed human speech so as to understand its production. Wave theory, the articulators, phonemes and graphs of speech were discussed to improve our understanding of speech. This in turn should help in the design of speech processing applications.

The next section will describe how to use digital signal processing techniques to process these human speech patterns.

Chapter 4

Digital Signal Processing

4.1 Introduction

Digital Signal Processing (DSP) is a collection of techniques for analysing any form of digital signal. In this document the signals referred to will be sounds recorded using a microphone and sampled using the sound card of a computer, but the techniques apply to other signals as well.

In this chapter basic sampling theory will be introduced followed by a short mathematical foundation for DSP techniques. Then an in-depth study of some of the more useful DSP techniques will be conducted. The chapter will then conclude with a description of ways of actually using the results provided by those techniques (feature extraction).

4.2 Sampling

Digital signals are usually samples from a continuous analog source sampled at regular time intervals. This sampling usually occurs using a microphone when dealing with audio signals. What this means is that at some interval (44100 samples per second for cd-quality sound) the signal is measured. These measurements are called samples and are proportional to the amplitude of the sound vibration at that time.

The convention that this document uses when dealing with sampled signals is that there is a discrete function f where $f[n]$ returns the amplitude of sample n. For example, a continuous, single-pitch sound will be produced by a sinusoidal function say $f(t) = cos(t)$ where $0° \leq t < 360°$. If this signal is sampled every $45°$, then the resultant values would be:

- $f[0°] = 1$

- $f[45°] \approx 0.707$
- $f[90°] = 0$
- $f[135°] \approx -0.707$
- etc.

Having briefly introduced sampling principles, it is important to describe some issues with the usage of sampling techniques. In the next section, some potential problems with sampling techniques will be illustrated.

4.2.1 Sampling Problems

One should be careful to collect sufficient samples to gain an accurate understanding of the function being sampled. Taking samples too sparsely (figure 4.1) can result in certain 'details' of the signal being ignored. These details would be the high-frequency portions of the signals.

Taking samples at a rate too closely matching the period of a sample (figure 4.2) will produce a completely incorrect representation of the signal - even low frequency samples would be excluded.

The opposite extreme - oversampling - does not produce accuracy problems, it wastes resources (it would need larger storage buffers) and processing time.

Now that we have seen some of the potential problems with sampling techniques, it would be valuable to find the balance between undersampling and oversampling. In the next section, a theorem will be described that allows one to place an upper bound on sampling frequency so as to minimise the number of samples required (prevent oversampling problems) without compromising the accuracy of signal representation (prevent undersampling problems).

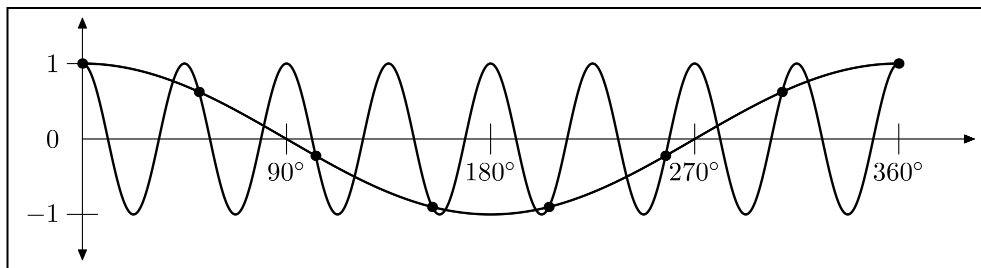


Figure 4.1: $f(x) = cos(8x)$ sampled every $\frac{360°}{7}$ appears to be $f(x) = cos(x)$

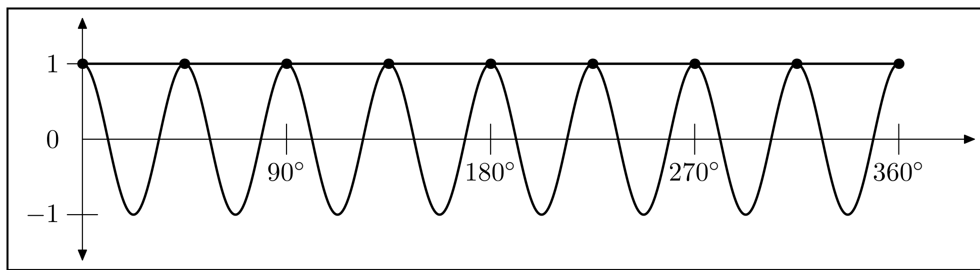


Figure 4.2: $f(x) = cos(8x)$ sampled every $45°$ appears to be $f(x) = 1$

4.2.2 Sampling Theorem

In the previous section, we saw that there are several problems that can arise when sampling a signal. This section will describe a theorem that can be used to prevent these problems.

There is a theorem (called the Nyquist Limit [89]) that can be used to place an upper bound on the required sampling rate. When a continuous function $f(t)$ has a maximum frequency of ω_m then that function can be accurately reproduced by its samples taken at a frequency of $2\omega_m$ to give samples $f(\frac{n\pi}{\omega_m})$ where $n \in \mathbb{Z}$. The original continuous function (signal), $f(t)$, can then be derived by:

$$f(t) = \sum_{n=-\infty}^{\infty} f[nT] sinc_T(t - nT), \tag{4.1}$$

where

$$sinc_T(t) = \frac{sin(\pi t/T)}{\pi t/T}.$$

For the proof of this theorem see [89]. The importance of this theorem (as mentioned above) is that it provides an upper bound on the sampling rate needed to accurately represent a signal containing a specific maximum frequency. This is useful because we know that humans can only hear sounds between 20Hz and 20kHz, and of that most of the phonetic information is concentrated below 8kHz [15].

So if 8000Hz (8kHz) is taken to be the maximum frequency of the signal it follows that sampling the signal at 16kHz is sufficient to represent the signal accurately. For other sources of information about sampling issues see [5, 73].

This concludes the section on sampling techniques, where we observed how poor application of sampling techniques can provide innaccurate results, and we showed that the Nyquist Limit can be used to improve sampling results.

Energy calculations (which are typically a first phase of digital signal processing) will be described in the next section.

4.3 Energy

In the previous section, sampling theory was discussed including several issues and their solutions. In this section energy calculations will be described within the context of sampled signals.

The intensity of the signal (how loud you speak) is an important part of digital signal processing. This intensity is also referred to as energy (E). Energy can be measured over the entire signal $f(x)$ (see equation 4.2) or over a given sub-band of frequencies (set of adjacent frequencies) within the signal. In order to measure energy within these sub-bands, the signal itself will need to be split up into signals of different frequencies. This is accomplished using Fourier Transforms (See section 4.7) or some other change-of-base transform.

The equation for calculating the energy of the entire discrete function f is [89]:

$$E = \sum_{x=-\infty}^{\infty} |f[x]|^2. \tag{4.2}$$

For regions of the infinite domain over which the signal is undefined, it is assumed to equal 0.

This value for energy is typically one of the features that is extracted and passed to a signal classification system. Most often the energy over the entire signal as well as the energy for each sub-band are used as inputs to the classification system. In this way, if the overall intensity of the signal has no bearing on the signal (only relative intensity between subbands) then the classification system can normalise these features.

The use of energy as a feature set of a signal will be explored in more detail in the section about the analysis process (section 4.9) and again in the chapter on signal classifications (chapter 5).

In this section, the energy calculations for a discrete (sampled) signal were described including sub-band energy level calculations. Next, the concept of digital filters will be described because they are used in nearly all aspects of digital signal processing (DSP) and hence form one of the necessary foundational sections of DSP in this document.

4.4 Digital Filters

In the previous section, energy calculations were briefly described. A related, yet different concept is the digital filter which also performs a function on samples of a signal. The difference is that the energy calculation outputs a single value, while the digital filter outputs a new signal.

Digital filters form a huge part of the theory of Digital Signal Processing. Digital filters are used to extract only the desired features from a signal. Typical uses of digital filters are to strip high or low frequency information out of a signal (low pass or high pass filters respectively), to strip noisy data out of a signal and to produce delays, echoes, distortion and other effects.

Digital filters can be implemented in hardware or in software, but the filters that will be discussed in this document are all implemented in software. Software filters tend to be very stable (always produce the same output given the same input), whereas hardware filters can be severely affected by temperature and other factors [80, ch14]. Software filters are also a lot more versatile and are faster to alter, due to the dynamic nature of software.

This section of the document describes some of the more important aspects of digital filters, from both an intuitive and a mathematical viewpoint.

4.4.1 How Digital Filters Work

Digital Filters perform some transformation on a signal, resulting in a new signal. In many documents discussing digital filters the original signal is denoted

$$\mathbf{X} = (x_0, x_1, x_2, ..., x_{n-1}),$$

the filter itself is denoted $\mathbf{H}$ and the resultant signal is

$$\mathbf{Y} = (y_0, y_1, y_2, ..., y_{m-1}).$$

Notice that the transformed result need not be the same length as the original signal, nor does y_i need to occur at the same time that x_i occurs. For the remainder of this document, we will only use y_i occurring at the same time that x_i occurs.

In many applications the signal is continuously being sampled and processed. This means that to compute the output value for y_i, the only input values available are those

that occurred before y_i in time. These include previously-recorded x values and already-calculated y values.

Some example filters are:

- *Amplification filters*:

$$y_i = kx_i \tag{4.3}$$

  Values of k from $0 < k < 1$ will reduce the signal intensity (attenuation) while k values of $k > 1$ will increase the signal intensity (amplification).

- *Delay filters*:

$$y_i = x_{i-l} \tag{4.4}$$

  The output signal after being passed through a delay filter is the original signal shifted by l samples. Assuming sampling starts at $i = 0$, values of $x_{i<0}$ (which would be undefined) are defined to be $x_{i<0} = 0$.

- *Echo filters*:

$$y_i = x_i + kx_{i-l} \tag{4.5}$$

  This results in the original signal, together with a (typically attenuated) copy of the signal as it was l samples earlier.

- *Running total*:

$$y_i = x_0 + x_1 + ... + x_i \tag{4.6}$$

  or more efficiently (once again taking $x_i = 0$ when $i < 0$):

$$y_i = x_i + y_{i-1} \tag{4.7}$$

In this section the workings of digital filters on an intuitive level were shown. In the next section, formal mathematical notation will be introduced.

4.4.2 Notation of Digital Filters

Now that the basic workings of digital filters have been shown intuitively, some of the notation and formulae will be described in a more formal manner.

Digital filters typically operate on samples that were recently sampled. The *order* (o) of a digital filter is defined as one less than the greater of (a) number of previous samples x_i and (b) the number of previous outputs y_i required as input to the filter. (Note that x_b for $a < b < i$ is assumed to be used if x_a is used). So the amplification filter (equation 4.3) is of order 0, the delay filter (equation 4.4) is of order l. The first running-total filter

(in equation 4.6) is of order i, while the second one (in equation (4.7) is of order 1.

The order of the filter is related to the number of previous samples that need be stored in a processing buffer for the filter to work. Obviously the smaller the buffer the less memory would be used (and the less work the CPU must do to execute the filter).

Another way of thinking about a digital filter is to think of it as a type of discrete signal itself. The convention is to represent the filter as $\mathbf{H} = (h_0, ..., h_{o-1})$. A function called *convolution* (which will be explained shortly) is applied - the original signal $\mathbf{X}$ is *convoluted* with $\mathbf{H}$ to produce $\mathbf{Y}$ [80, ch6].

The formal formula for convolution of two *discrete* signals $\mathbf{X}$ and $\mathbf{H}$ in mathematical notation ($\mathbf{Y} = \mathbf{X} * \mathbf{H}$) is:

$$y_i = \sum_{k=-\infty}^{\infty} x_k h_{i+k}.$$

Usually a digital filter H is defined as zero for all points outside of some active domain (the domain for which it has non-zero values). In those cases it is possible to modify the interval over which the summation occurs to apply only to the appropriate filter values. This is very important for implementation purposes, as it is impossible to execute an infinite summation in a loop on a CPU. Therefore:

$$y_i = \sum_{k=0}^{n-o-1} x_k h_{i+k}, \tag{4.8}$$

where o is the order of the filter as define at the start of this section.

Filters that only use the original samples (no previously calculated filter outputs) as their input are called *Finite Impulse Response* filters (FIR filters), while those that also make use of previously calculated filter output values are called *Infinite Impulse Response* filters (IIR filters). For the change-of-base transforms described in this paper, the IIR filters will often be used.

So far the workings of digital filters have been described and the applicable formulae given. In the next section this notation will be written in a more compact form to facilitate understanding some of the more advanced concepts that will be described later.

A more compact representation of filters

A FIR filter of order l written as $y_i = a_0x_{i-0} + a_1x_{i-1}... + a_lx_{i-l}$ is often written as $(a_0, a_1, ..., a_l)$. For example, a filter that takes the average of every two consecutive samples would be written $(\frac{1}{2}, \frac{1}{2})$.

A related filter to the one described above is $(\frac{1}{2}, \frac{-1}{2})$. Together these two filters applied to a signal calculate the average shape of the signal (the first filter of the two), while the latter filter calculates the detail of the signal (it represents the amount by which the original signal differs from the average signal). These two filters together (but applied only to non-overlapping sets of samples) make up the discrete Haar wavelet, which will be explained later.

In this section a compact representation for digital filters was given. This representation will be used for the remainder of this document. In the next section, we will continue to discuss some of the other mathematical foundations needed for digital signal processing.

4.5 Mathematical Foundations for DSP

In previous sections, sampling, energy calculations and digital filters were described. In this section, some additional mathematical foundations for DSP techniques will be established. These will include:

- Geometry of functions (a comparison between vector geometry and function geometry, required for the change-of-base transforms described later).
- Using $e^{i\theta}$ to represent sinusoids, which is used (amongst others) in Fourier Transforms, also described later.

4.5.1 The Geometry of Functions

In the next few sections, some function geometry will be described. Function geometry is merely an extension of some of the geometrical rules defined for vectors so they can be applied to functions and still (intuitively) behave the same way. This is important to us because signals can be thought of as functions, and so all the techniques described in the following few sections can also be applied to signals. The specific geometrical rules that will be described are: inner products, norms and angles.

4.5.2 The Inner Product

The inner product is a very useful function defined for some given space $\mathbb{S}$. The inner product for $\mathbb{S}$ must fulfil the properties described below [47], [89, p19]:

- The inner product function takes two inputs ($x \in \mathbb{S}$ and $y \in \mathbb{S}$) and outputs a single scalar (usually a real or a complex number). It is denoted $\langle x, y \rangle$
- Rule of linearity: $\langle ax + by, v \rangle = a\langle x, v \rangle + b\langle y, v \rangle$, $a \in \mathbb{R}$, $b \in \mathbb{R}$ and $v \in \mathbb{S}$
- Rule of symmetry: $\langle x, y \rangle = \overline{\langle y, x \rangle}$ (where $\overline{c}$ is the complex conjugate of c)
- Zero Vector : There exists $x \in \mathbb{S}$ such that $x + x = x$. x is then called the zero vector.
- Positivity: $\langle x, x \rangle \geq 0$ and $(\langle x, x \rangle = 0)$ implies that x is the zero vector.
- Cauchy Schwartz inequality: $|\langle x, y \rangle| \leq \|x\|\|y\|$, where $\|x\| = \sqrt{\langle x, x \rangle}$ (equation 4.12).
- Triangle inequality: $\|x + y\| \leq \|x\| + \|y\|$

Having described the inner product function in general, we will show how it is applied to vectors, then extend the application to functions (and signals) as well. In later sections we describe how we can use the inner product to perform additional calculations on functions.

The Inner Product Applied to Functions

The inner product, as mentioned above, is very useful in digital signal processing. In this section, inner products as they apply to vectors (dot product), will be extended to apply to functions as well. Because signals are functions themselves, we would then be able to apply this theory to our DSP techniques.

For two equal-dimensioned vectors $\mathbf{a} = (a_0, a_1, ..., a_{n-1})$ and $\mathbf{b} = (b_0, b_1, ..., b_{n-1})$, the dot product (a function that fulfils inner product rules) is defined as:

$$\mathbf{a}{\cdot}\mathbf{b} = \sum_{i=0}^{n-1} a_i b_i,$$

for $a_i \in \mathbb{R}$ and $b_i \in \mathbb{R}$ or for complex numbers ($a_i \in \mathbb{C}$ and $b_i \in \mathbb{C}$):

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=0}^{N} a_i \overline{b_i}, \tag{4.9}$$

where $\overline{b_i}$ is the complex conjugate of b_i.

In the same way as we can denote a vector $\mathbf{v}$ with a finite number of dimensions as:

$$\mathbf{v} = (v_0, v_1, ..., v_{n-1}),$$

a discrete function $f[x]$ defined over some domain $0 \leq x \leq (n-1), x \in \mathbb{N}$ can be denoted:

$$F = (f[0], f[1], ..., f[n-1]).$$

The dot product defined for vectors then also works for these functions:

$$F \cdot G = \sum_{i=0}^{n-1} F_i \overline{G_i} = \sum_{i=0}^{n-1} f[i]\overline{g[i]} \tag{4.10}$$

The formula is only slightly different for continuous functions. For the family of all continuous functions $f_i(x)$ which are defined over the domain $d_{min} \leq x \leq d_{max}$, the standard inner product is defined as:

$$\langle f_j, f_k \rangle = \int_{d_{min}}^{d_{max}} f_j(x)\overline{f_k(x)}dx. \tag{4.11}$$

This is also called the L_2 inner product [47] - see section 4.5.3 for more details.

We have applied basic inner product theory to functions, and can now start using these inner products in other formulae. Specifically *norm* and *angle* calculations will be shown.

The Norm of a Function

It was shown in the previous few sections how inner products can be applied to functions. These principles will be applied in this section to show how to calculate the norm of a function, as well as showing what use this norm value is to us.

The length (which is also called the norm) of a vector $\mathbf{v} = (v_0, v_1, ..., v_{n-1})$, denoted $\|\mathbf{v}\|$, is given by:

$$\|\mathbf{v}\| = \sqrt{\sum_{i=0}^{n-1} v_i^2}.$$

(This is really just the repeated application of Pythagoras' theorem for calculating the length of the hypotenuse of right-angled triangles). This is equivalent to

$$\|\mathbf{v}\| = \sqrt{\mathbf{v}\cdot\mathbf{v}} = \sqrt{\langle\mathbf{v},\mathbf{v}\rangle}. \tag{4.12}$$

This latter definition of norm (equation 4.12) is used verbatim to apply to functions as well [47]. In other words the norm of a function f is:

$$\|f\| = \sqrt{\langle f, f\rangle}. \tag{4.13}$$

The norm value is very useful for normalising calculations in order to preserve energy values after a transformation. The norm will also be used in determining *orthonormality*, which is described towards the end of the next section.

Besides calculating the norm value, the inner product can also be applied to determine angles between vectors. A similar concept for functions will be described in the next section.

'Angles' Between Functions

In the previous sections, the inner product as it applies to functions was explained. One application of the inner product was also shown: norm calculations. In this section another application of inner product calculations will be used to derive the concept of the 'angle' between two functions.

In order to calculate the angle between two vectors $\mathbf{v}$ and $\mathbf{w}$, we solve for θ in the following formula:

$$\mathbf{v}\cdot\mathbf{w} = \langle\mathbf{v},\mathbf{w}\rangle = \|\mathbf{v}\|\|\mathbf{w}\|\cos\theta. \tag{4.14}$$

The concept of the 'angle' between two functions has been defined to be the same as for vectors [47]. By starting with the above formula we can calculate the 'angle' θ between functions v and w [47]:

$$\begin{aligned}
\langle v, w\rangle &= \|v\|\|w\|\cos\theta \\
\frac{\langle v, w\rangle}{\|v\|\|w\|} &= \cos\theta \\
\theta &= \arccos\frac{\langle v, w\rangle}{\|v\|\|w\|}.
\end{aligned}$$

Using the above calculations, *orthogonality* ($\theta = 90° = \frac{\pi}{2}$) can be determined. Orthogonality plays a large role in deciding the effectiveness of many change-of-base transforms (see section 4.6). Two functions f and g are said to be orthogonal if $\langle f, g\rangle = 0$

(because if $\cos\theta = 0$ then $\theta = 90°$ which implies orthogonality). This can be extended to sets of more than two functions: a set of functions $\{f_i\}$ from the same function space is said to be orthogonal if $\langle f_j, f_k \rangle = 0, j \neq k$.

Furthermore, a set of functions $\{f_i\}$ from the same function space is said to be *orthonormal* if the set is orthogonal and $\|f_i\| = 1, \forall i$.

In the next two sections it will be explained why orthogonality is so important and (in an intuitive manner) exactly what it means to span the L_2 space (this was mentioned after equation 4.11 and is a term that is frequently used in DSP techniques). These two concepts are of vital importance to the change-of-base transforms.

4.5.3 Spanning the L_2 Space

It was mentioned in a section 4.5.2, that there exists a space called L_2, whose elements are functions. In this section the concept of spanning this L_2 space will be explained. A good place to start is with vectors on the Cartesian Plane (what we usually think of as the X-Y axis or $\mathbb{R}^2$). By writing co-ordinates as pairs (x, y) we can represent points on that plane.

If we take $(1, 0)$ and $(0, 1)$ as base vectors, we can (for *any* x and y) say that $(x, y) = x(1, 0) + y(0, 1)$. Because this can be done for any x and y, we can say that $(1, 0)$ and $(0, 1)$ are vectors that *span* the plane.

These are not the only possible base vectors, they just happen to be the most commonly used ones to represent points on the Cartesian Plane.

Should we use any additional vectors to attempt to span the plane, then there are too many vectors. Any given point on the plane could then be written in more than one way. We cannot use fewer than two vectors either. Hence we can say that the vectors $(1, 0)$ and $(0, 1)$ *minimally* span the $\mathbb{R}^2$ plane.

Now a parallel between the above concept of spanning (for vectors), and a similar one for functions will be shown. In the same way as the $\mathbb{R}^2$ plane contained vectors (x, y), the L_2 space contains functions $f_i(x)$. In the same way as we wanted to represent *any* point in $\mathbb{R}^2$ space by using the sum of multiples of basis vectors ($(x, y) = x(1, 0) + y(0, 1)$), we want to represent *any* function f in L_2 space by the

sum of multiples (α_i) of basis functions (ψ_i):

$$f = \sum_{i=0}^{\infty} \alpha_i \psi_i. \tag{4.15}$$

The reason this summation stretches to infinity is that in n dimensional space, there are n base vectors. L_2 space has infinite dimensions, unless the domain of the functions has a finite number of elements.

The first problem is to find such a set of basis functions $\{\psi_i\}$. The second is to ascribe some additional meaning to this newly created transform (otherwise the exercise is probably just wasted effort). Unfortunately (as can be seen from the interval of the summation in equation 4.15) the size of any family of functions capable of spanning L_2 happens to be infinitely large. Fortunately these functions are usually strongly related to one another and are therefore usually easy to use. In practise it is usually possible to utilise some subset of these basis functions to represent the original function with appropriate accuracy (this is especially useful in compression techniques), or to limit the functions to discrete functions over finite domains (this would give the function domains a finite number of elements).

Later in this chapter, the problem of finding these basis functions and their meanings will be discussed. For now it is sufficient that one understand the concept of spanning L_2 space.

We continue with this example in the next section to explain the importance of orthogonality.

4.5.4 The Importance of Orthogonality

We know that $(1, 0)$ and $(0, 1)$ are at $90°$ to one another, and this means that should a point move in the direction of one of these base vectors, the coefficient of the other base vector does not change (a horizontal movement does not alter the vertical coefficient). This implies that a movement in the direction of one base vector requires only a single coefficient to represent - all other coefficients are 0. So orthogonality of basis vectors promotes representing features of a given vector using minimally few (non-zero) basis vectors.

For the same reason, orthogonality is important to basis functions. One area of DSP where this is particularly important is wavelet theory.

This concludes our introduction to function geometry, but there are still two more mathematical concepts to cover before the actual DSP transforms will be explained: raising e to a complex power and the delta functions.

4.5.5 Raising e to a Complex Power

DSP calculations often rely on the sinusoidal functions $\sin\theta$ and $\cos\theta$ as a pair. Fourier (and other) transforms (see section 4.7) are one example. You will frequently notice the formula $e^{i\theta}$. In this context, $i^2 = -1$. (Some texts use the symbol j instead of i). While this does not really make intuitive sense using the conventional understanding of exponents, it has been defined to be:

$$e^{i\theta} = \cos(\theta) + i\sin(\theta). \tag{4.16}$$

It can be helpful to think of this as a 2-D vector in the complex plane, at an angle of θ radians and of length 1 (see figure 4.3).

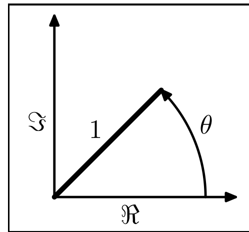


Figure 4.3: Raising e to a complex power ($e^{i\theta}$)

There is one last foundational concept that will be explained in this section - the delta function - before the actual DSP algorithms start.

4.5.6 The Delta Function

Up until now in this chapter, certain foundational mathematical concepts for digital signal processing have been introduced. The final concept we will describe is the delta function.

Many of the techniques in digital signal processing involve extracting a single sample (discrete signals) or an infinitely narrow portion of the domain (continuous functions). To do this, the signal is multiplied by a shifted delta function. The shifting is done to select the correct sample in the domain of the function. The delta function is more useful from a theoretical viewpoint than from an actual application viewpoint.

The continuous delta function is defined as:

$$\delta(x) = \begin{cases} \infty, & x = 0 \\ 0, & x \neq 0 \end{cases},$$

where

$$\int_{-\infty}^{\infty} \delta(x) dx = 1.$$

The discrete delta function is defined as:

$$\delta(x) = \begin{cases} 1, & x = 0 \\ 0, & x \neq 0. \end{cases}$$

This concludes our coverage of the mathematical foundation for DSP techniques. We will now study a set of DSP techniques collectively called change-of-base transforms.

4.6 Change-of-Base Transforms

In the remainder of this chapter, we are going to describe certain DSP techniques. Some of the concepts and notation we use were described in the earlier sections of this chapter.

In order to make sampled signals easier to study, a basis transform is frequently used. What this means is that we would like to write an arbitrary discrete function $f[n]$ as:

$$f[n] = \sum_{k \in \mathbb{Z}} \langle \varphi_k, f \rangle \varphi_k[n], n \in \mathbb{Z} \tag{4.17}$$

where φ_k are the new basis functions.

One of the most commonly used change-of-base transforms, transforms a periodic function based on time (t) to a function based on frequency. This specific transform is known as the Fourier Transform and will be discussed in detail in section 4.7. Another powerful transform is the Wavelet Transform (section 4.8) which has certain advantages over the Fourier technique when applied to rapidly changing signals and signals with very localised features. These are just examples of change-of-base transform and will be explained in detail later.

Before delving into too much detail regarding change-of-base functions, we will explain some of the reasons for using them in the first place.

4.6.1 The Reason for Doing Change-Of-Base Transforms

Changing the basis of a function is a fair amount of effort (perhaps seemingly too much just to represent some function in a different way). This section will explain some of the advantages of converting a function to one using a different basis, so as to justify the effort.

We will begin with an example. Let us say we have a function $f(x) = 3\sin(x) + 5\sin(2x)$. This is the superposition of two sine functions. Should two of our basis functions happen to be $f_1(x) = \sin(x)$ and $f_2(x) = \sin(2x)$, then we could write $f(x) = 3f_1 + 5f_2$ or alternatively (using only the coefficients of these basis functions) $f(x) = (3, 5)$. This has reduced a complicated function defined over the infinite real domain down to a mere two coefficients. This will only work if the original function can be written in this manner. Using the basis functions we will be using later, this will *always* be the case because the set of basis functions span the L_2 space (see section 4.5.3). The two functions used here do not span the L_2 space, but are sufficient to understand the above-mentioned advantages on an intuitive level.

The first advantage of acquiring this representation is compression. Another is function analysis.

Compression

The above-mentioned technique could possibly be useful if we can predict that functions being compressed would always be the sum of very few basis functions. For lossy compression, small coefficients or coefficients of unimportant functions (inaudibly high frequencies in a sound signal, for example) could be zeroed, resulting in better compression, while retaining a sufficiently accurate reproduction of the original signal. Thus change-of-base transforms can be a useful technique in compression algorithms.

One practical example of using DSP techniques for compression is given in chapter 15.

Analysis

Typically the basis functions are structured - the formulae for the basis functions are usually similar to one another (often differing only by a coefficient). This helps a lot when actually implementing the algorithms in software. (A finite subset of the basis functions can often be used to implement the change of base transform with a desired level of accuracy. This is called successive approximation [89, p93]). This can be advantageous not only to compression applications, but also to analysis ones. If we know

that certain basis functions exhibit certain behaviour (eg. some specific frequency) - then if the coefficient for any basis function is significant, then the original function possibly exhibits the same behaviour (to some extent) as that basis function. It is for this reason that change-of-base transforms can be a useful signal analysis tool.

Now that we see why change-of-basis transforms can be useful, we will study some of them. We look first at the *Fourier Transform* followed by *wavelets*. Both of these are widely used transforms in the field of DSP especially audio processing.

4.7 The Fourier Transform

In the previous sections, several goals for DSP were described (analysis and compression, inter alia) and a mathematical foundation for some DSP techniques was established. Now the actual transforms will be described starting with Fourier Transforms.

4.7.1 Introduction to the Fourier Transform

When a function over an infinitely long time domain exhibits periodicity then that function can be represented as the superposition of sine and cosine waves of differing periods and amplitudes. This transform is known as the Fourier Transform, named after its discoverer - Joseph Fourier who discovered it in 1807 [42]. The Fourier Transform for an integrable function $f(\omega)$ is defined by [89, p37]:

$$F(\omega) = \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt. \tag{4.18}$$

Therefore it can be observed that the Fourier Transform results in a set of complex (see section 4.5.5) coefficients of sinusoidal basis functions of differing periods. These coefficients can be used to derive the amplitudes of sine and cosine functions, which when superimposed would result in the original function.

The function in equation 4.18 is also called the Fourier *Analysis* formula because it uses the Fourier technique to study (analyse) the periodicity of the function.

The inverse Fourier Transform is defined as:

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega) e^{i\omega t} d\omega = \frac{1}{2\pi} \langle e^{i\omega t}, F(\omega) \rangle \tag{4.19}$$

This function is also called the Fourier *Synthesis* formula as it can be used to synthesize the original function using Fourier techniques.

The Fourier analysis function was originally designed to work with signals that are continuous, but the Discrete Fourier Transform, which will be discussed next, can be used for discretely sampled signals over an infinite duration (or finite duration in the case of the Short-Time Fourier Transform).

4.7.2 The Discrete Fourier Transform

As mentioned in the previous section, the Fourier Analysis function was originally applied to continuous periodic functions. This has limited use in software-based signal processing environments, as the signals are typically sampled at discrete intervals, or are only periodic over a short period of time (or are sometimes not periodic at all).

The intuitive understanding is that the continuous and discrete versions work the same way, except that the continuous case uses integration, while the discrete case uses a summation.

Given a sequence of samples ($f[n]$), the Discrete Fourier Transform (DFT) is defined by [89, p51]:

$$F[k] = \sum_{n=0}^{N-1} f[n](e^{-i2\pi/N})^{nk}.$$

This is almost as if the entire signal f is being convoluted with a sinusoid, except that the sinusoid changes frequency slightly as k increases. Some texts substitute $W_N = e^{\frac{-i2\pi}{N}}$:

$$F[k] = \sum_{n=0}^{N-1} f[n] W_N^{kn}. \tag{4.20}$$

The W_N^k values are called the N_{th} roots of unity. This is because $(W_N^k)^N = 1$ for all $k \in \{0...N-1\}$. The inverse of this function is:

$$f[n] = \frac{1}{N} \sum_{k=0}^{N-1} F[k] W_N^{-kn}. \tag{4.21}$$

As can be seen from the formulae above, the $F[k]$ values are complex numbers (see also section 4.5.5). The coefficient of the cosine wave can be determined from the real portion, while the coefficient of the sine wave can be determined from the imaginary portion.

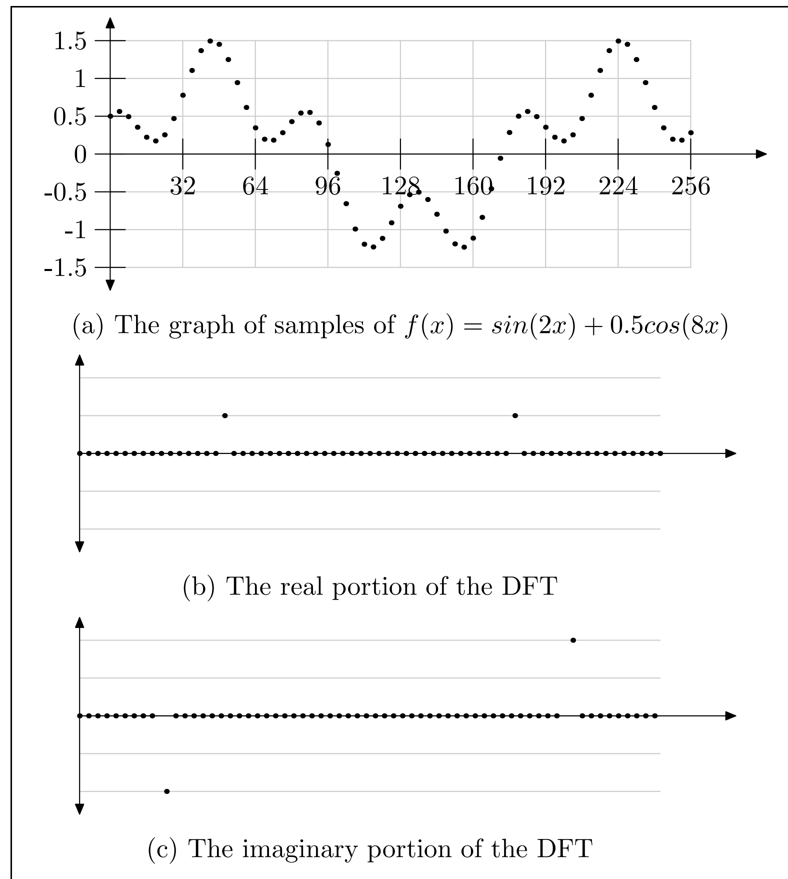
The amplitude of the $F[k]$ values are proportional to both the amplitude of the original signal $f[n]$ as well as the length of the set (N). By doubling the number of samples

of the signal (while still maintaining periodicity), the values $F[k]$ double.

Another interesting result is a complex-conjugate symmetry (See figure 4.4):

$$F[k] = \overline{F[N-1-k]} \tag{4.22}$$

This Discrete Fourier Transform is extremely useful, yet relatively slow to execute.



(a) The graph of samples of $f(x) = sin(2x) + 0.5cos(8x)$

(b) The real portion of the DFT

(c) The imaginary portion of the DFT

Figure 4.4: Symmetry in fourier transforms

In the next section, it will be shown how to optimise the Fourier Transform algorithm complexity from $O(N^2)$ to $O(N\log_2(N))$ using the *Fast Fourier Transform*.

4.7.3 The Fast Fourier Transform

The Discrete Fourier Transform algorithm executes in $O(N^2)$ time. This can easily be seen. There are N $F[k]$ values, and each of those values is an N length summation. This execution time can be reduced given certain assumptions. These optimised algorithms are generally called Fast Fourier Transforms (FFT). The Cooley-Tukey Decimation in Time FFT algorithm is one such algorithm. [5, 73, 89]

Position	Binary	Bits Reversed	New Value
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

Table 4.1: Bit-reversing indices

The Cooley-Tukey Decimation-in-Time FFT

If the number N of samples in the original discrete signal can be written as 2^l, where l is a positive integer, then we can optimise the DFT:

$$\begin{aligned} F[k] &= \sum_{n=0}^{N-1} f(n)e^{-i2\pi kn/N} \\ &= \sum_{n=0}^{N/2-1} f(n)e^{-i2\pi kn/N} + \sum_{n=N/2}^{N-1} f(n)e^{-i2\pi kn/N} \\ &= \sum_{n=0}^{N/2-1} \{f(n) + f(n+N/2)e^{-i\pi k}\}e^{-i2\pi kn/N} \\ &= \sum_{n=0}^{N/2-1} \{f(n) + f(n+N/2)(-1)^k\}e^{-i2\pi kn/N}. \end{aligned}$$

It can be seen that the intervals over which the summation is executed is half that of the original - reducing the complexity of the problem by a factor of two. This whole process can be iterated $l-1$ (remember $N = 2^l$) times, eventually leading to trivial ($N = 2$ sized) DFT's making the complexity of the total algorithm $O(N \log_2 N)$. This can be derived from $N = 2^l$ which means that $l = \log_2 N$ and there are N $F[k]$ values to calculate. [89, p339]

For another explanation of the Cooley-Tukey algorithm see [45]. A diagram representing the workings of this algorithm is shown in figure 4.5. The input values on the left of the diagram (a values) have indices which are bit-reversed. This is a part of the algorithm that can be calculated once and the results reused every time the function is used. Table 4.1 shows how to determine bit-reversed numbers.

The reason one uses this bit-reversal becomes apparent if one fully expands the Cooley-Tukey DFT. By dividing the set in half in the manner described in the technique, the effective transformation to the indices used is the bit-reversal function. The number of bits used in the bit-reversal scheme is dependant on the number of samples over which the DFT is being performed.

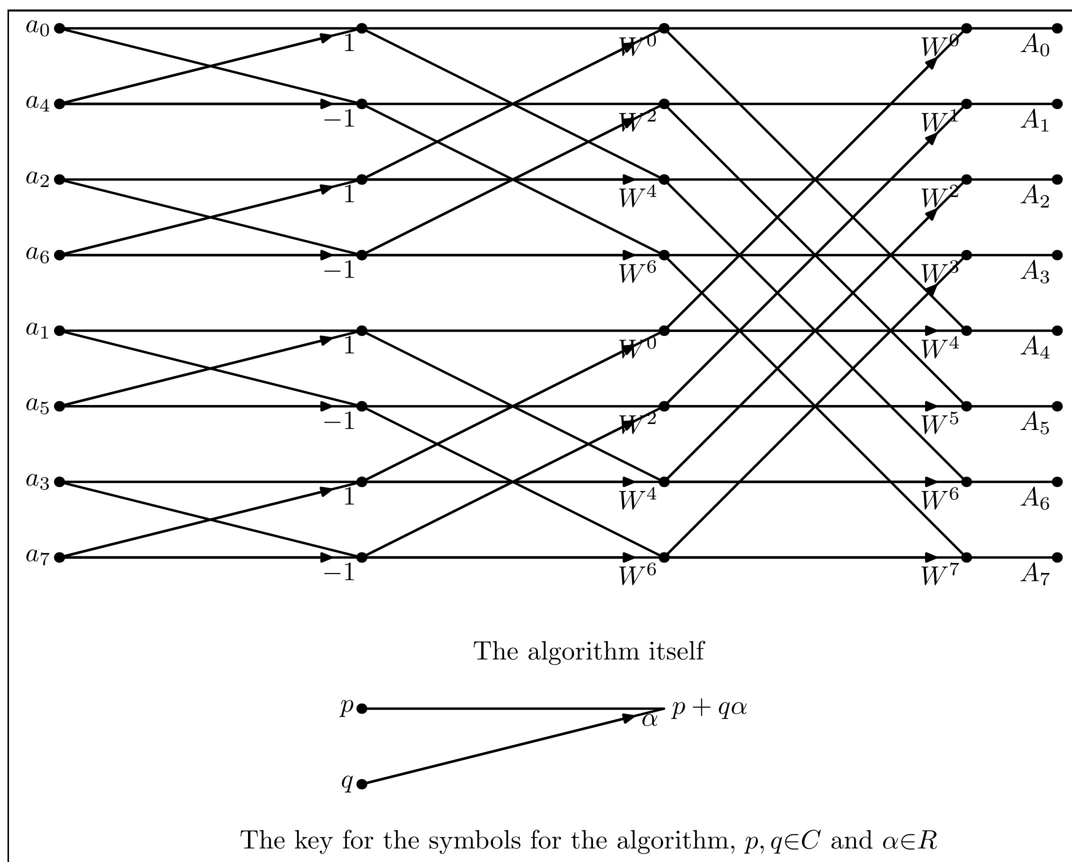


Figure 4.5: The working of the Cooley-Tukey FFT algorithm [45] for $N = 8$ (See table 4.1 for index calculations).

4.7.4 Other FFT Algorithms

There are other FFT algorithms - especially those used when N can be written as $N = N_1 \cdot N_2$, where N_1 and N_2 are coprime. Some examples are the Good-Thomas (or Prime Factor) FFT algorithm and the Winograd FFT algorithm. While these algorithms can result in slightly fewer multiplications, their complexity has lead to mixed success. For this reason the Cooley-Tukey FFT remains the most popular [89, p340].

The saving by reducing the algorithm complexity (reduction from $O(N^2)$ to $O(Nlog_2N)$) is a huge saving, (especially when dealing with large numbers of samples), but there are further optimisations that while not altering the complexity of the algorithm, do reduce the execution time. These are typical computer optimisations such as:

- pre-calculated trigonometric lookup tables,
- pre-calculated bit-reversed indexes (see table 4.1) and
- using fixed-point numbers as opposed to floating point numbers (depending on desired accuracy and CPU architecture).

4.7.5 Problems with Fourier Transforms

Despite its usefulness, the Fourier Transform has several limitations. Two of these are period shifting and frequency leakage:

Period Shifting

One important aspect of the Discrete Fourier Transform is period shifting. Should the DFT be executed twice for the same periodic function, but starting at a different position (relative to the start of the function's period), the results will differ. Should this fact be ignored, then using the result of the Fourier Transform as an input vector to a classification system could reduce the accuracy of classifications. Fortunately, it is possible to alter the results so they remain unchanged when applied to the same signal, irrespective of shifting the function.

When the signal is shifted, the magnitude of the complex number for each frequency in the result remains constant - only the angle of rotation differs (see figure 4.3). Fortunately for phoneme recognition, only the amplitudes of the signal (magnitudes of the complex numbers) at the different frequencies are important. Using Pythagoras' Theorem, one can calculate this magnitude:

$$M_k = \sqrt{\Re_k^2 + \Im_k^2},$$

where M_k is the magnitude and $\Re_k$ and $\Im_k$ are the real and imaginary portions of the kth coefficient after a Fourier Transform respectively.

One possible optimisation is to ignore the square-root portion of the magnitude calculation, and simply use the energy value of each frequency:

$$E_k = M_k^2 = \Re_k^2 + \Im_k^2.$$

This is faster because the square-root function can take a long time to calculate depending on the CPU architecture.

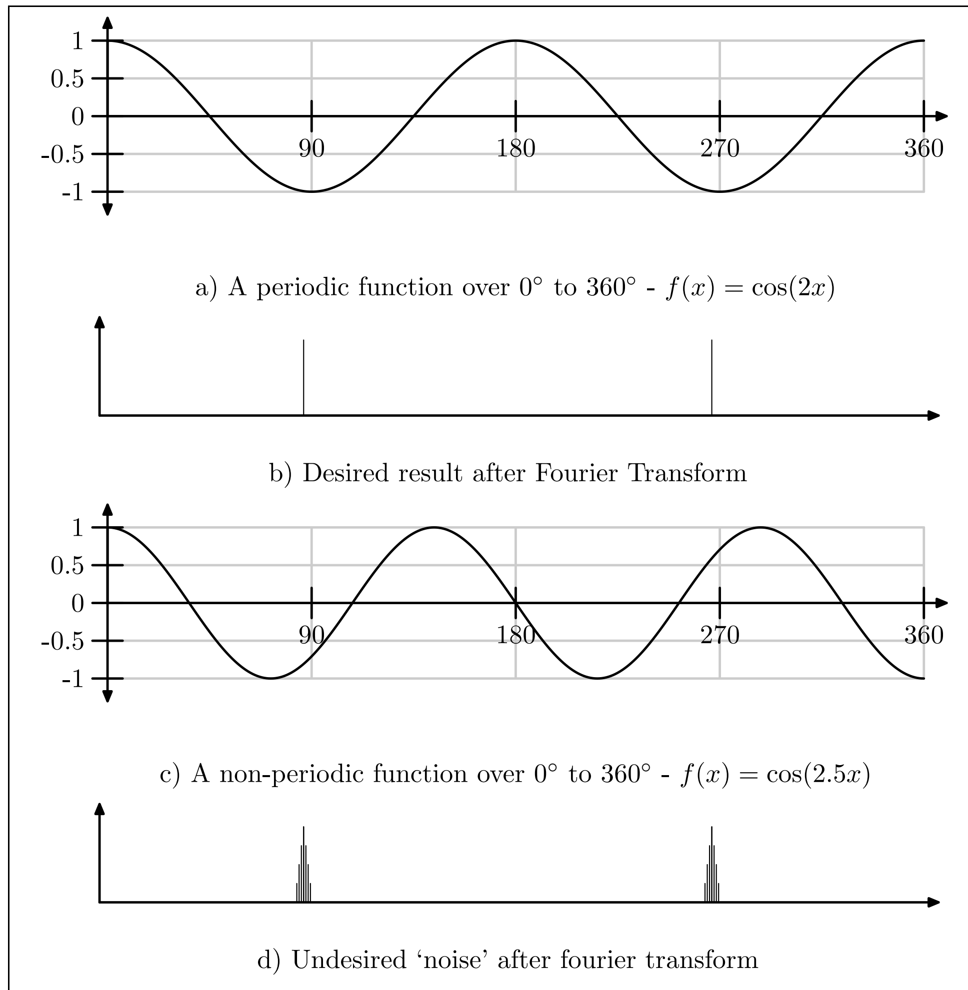
Period shifting is only one of the issues with which one must deal when using Fourier Transforms. Another is that of frequency leakage.

Frequency Leakage

When the signal passed to the Fourier Transform is not exactly periodic within the given domain, *frequency leakage* results. This is observed as 'glitches' - non-zero coefficients

for frequencies that are not actually present in the original signal, or to produce inaccurate amplitude information about a given frequency [5].

Fortunately the errors are localised near the true frequency of the signal (see figure 4.6). This means that using the energy over bands of adjacent frequencies as features for our classification system, we should still be able to classify the signal fairly accurately. The Inverse DFT still results in the original samples despite this non-periodicity.



a) A periodic function over 0° to 360° - $f(x) = \cos(2x)$

b) Desired result after Fourier Transform

c) A non-periodic function over 0° to 360° - $f(x) = \cos(2.5x)$

d) Undesired 'noise' after fourier transform

Figure 4.6: Frequency leakage [5]

In summary the Fourier Tranformation is a useful tool for DSP, but has certain limitations. In the next section a different transform will be described that can overcome or reduce the effect of these limitations.

4.8 Wavelets

In the previous section, the Fourier Transform was discussed, including optimisations for real-time use as well as potential shortfalls when using the technique. This section will describe Wavelet Transforms, which can (to some extent) overcome the following issues:

- The signal is assumed to be periodic. Human speech however is not truly periodic, even over small durations. The DFT typically takes as input, the samples over some fixed-length portion (frame) of the signal. This technique fails when there are sudden changes to the signal part-way through the frame, for example the explosion of released air while uttering the phoneme 'p' may occur in the middle of a frame, thus producing strange results. Wavelets do not require periodicity. They are in fact extremely useful for highlighting localised features in a signal.

- The Short-Time-Fourier-Transform (STFT) cannot be easily reversed to reconstruct the signal. After the signal is transformed in some way while represented in the frequency domain, and the STFT is reversed, the new signal contains discontinuities between time frames. Fortunately this is only a problem when the STFT needs to be reversed. Wavelet Transforms can easily be reversed.

- The narrower the window used for better time resolution the poorer the frequency resolution and vice-versa. Some reasons for this are: (a) in a small window, the lower frequencies do not oscillate over one complete period, and (b) in a large window, the signal's periodicity may change. The effects of this problem can be reduced by using a dynamically sized window that shrinks as the frequency resolution increases. This technique is applied when performing Wavelet Transforms and is the core of multi-resolution analysis.

The list shown above shows some of the reasons why we would want to use wavelets, but their value has not been appreciated until recently. The following brief history will illustrate some of the collaborative efforts that have caused wavelets to be so useful today.

4.8.1 A Brief History of Wavelets

The earliest reference to wavelets were in one of the appendices to Haar's thesis (1909). Since then researchers from many fields of study have used wavelet theory to aid in their work.

Paul Levy, a physicist in the 1930's used the Haar wavelet to investigate Brownian motion (the random motion of particles in gas) because of the superiority of wavelets

for studying localised details of signals.

Over time correlations between Fourier Transforms and other techniques (including quadrature mirror filters, pyramid algorithms and wavelets) were discovered (Some of the most significant work was done by Stephane Mallat in 1985 [42]). This aided in the search for better wavelet basis functions.

In 1988 Ingrid Daubechies, working at AT&T Bell Laboratories discovered several wavelets that have fractaline properties, and are extremely useful for processing many different types of signals [82]. This makes the Daubechies family of wavelets some of the most commonly used wavelet transforms today.

Fortunately for us wavelets are now a well-defined branch of mathematics. The following sections will explain the maths behind wavelets.

4.8.2 Understanding Wavelets

Wavelet Transforms, like Discrete Fourier Transforms (DFTs), are used to represent a function as the sum of multiples of basis functions. Where Fourier Transforms use sinusoids (which have infinite support), wavelets use functions that have more localised support.

Support in a function merely describes the domain of the function over which it returns non-zero values. So sine waves have infinite support. A function $f(x)$ that is zero outside of $a \leq x \leq b$ has localised support over the domain $[a, b]$.

Note: *In many texts (including this one) about wavelets the word* frequency *is used. This term is more of an intuitive understanding, than a mathematically correct one.*

Wavelets differ from the Fourier Transform in that the range of samples over which each value of the transformation is calculated varies depending on the scale (frequency resolution) of the transformation. The higher the scale represented, the fewer the number of samples used. For a diagrammatic comparison between the results of the Discrete Fourier Transform and the Wavelet Transform see figure 4.7. This diagram also shows that (with wavelets) at a given frequency resolution, over a single frame of samples, there may be more than one result, where the Fourier Transform produces a single result per frequency per frame of samples.

First an intuitive understanding of wavelet thinking will be described, using the Haar

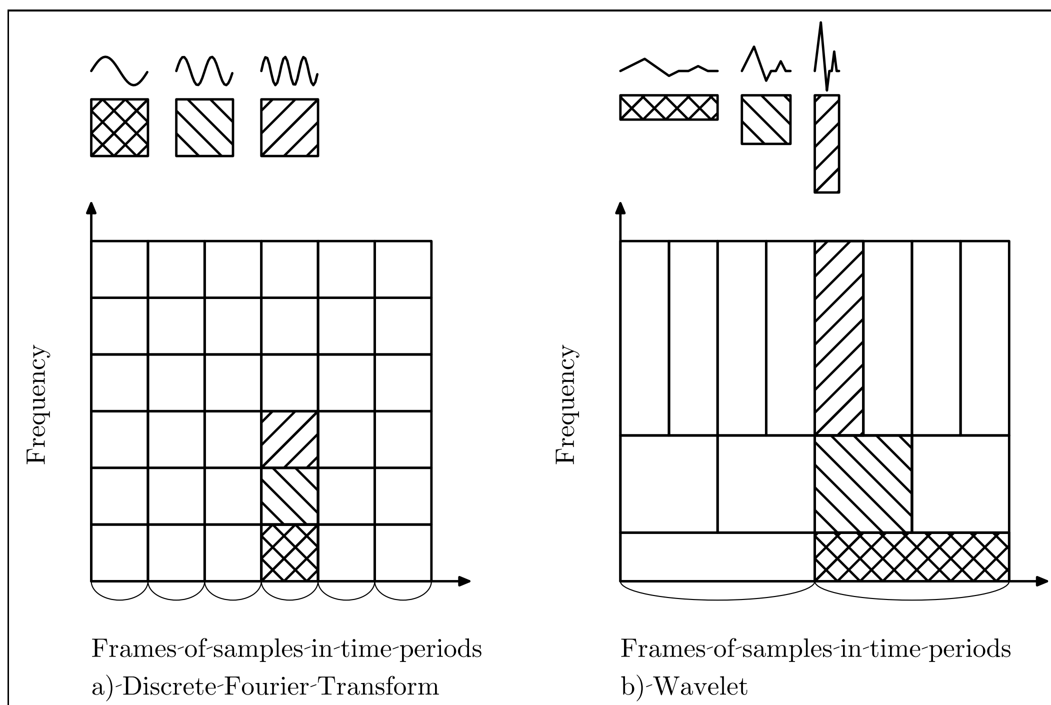


Figure 4.7: The difference between Fourier and Wavelet transformations [42]

wavelet (one of the simplest wavelets to understand) as an example. This will be followed by a more rigorous mathematical treatment.

To understand wavelets properly, it is useful to understand the notion of dimensionality. A vector on the Cartesian plane (x, y) is two-dimensional. Similarly if you use N samples from a signal F, they can be written as $(f_0, f_1, ..., f_{N-1})$. The vector whose elements are those samples is N dimensional. If a function space is 2^j dimensional ($N = 2^j$), then the space is written as V^j. It is important to know that all functions in V^j can also be represented in V^{j+1}. An example of this is $(1, 2, 3, 4)$ can be written as $(1, 2, 3, 4, 0, 0, 0, 0)$. Intuitively this means that these spaces are nested. This nesting is one of the core foundation facts of wavelets.

Given a function space V^j, we define W^j as the orthogonal complement of V^j in V^{j+1}. This means that W^j space is the set of all functions in V^{j+1} not representable in V^j. (Intuitively this means that W^j is the set of functions 'left over' when signals are described using basis functions from V^j). Put another way, if one takes the functions that span V^j and the functions that span W^j, then using all those functions, we can span V^{j+1}.

We will now make this clearer using the Haar wavelet as an example. Take a tiny signal of 4 samples $(4, 6, 22, 20)$. This size 4 can be written as 2^2 so we can think of

the vector as a vector in V^2 space. The Haar wavelet reduces this to a vector in V^1 by averaging every two samples to produce $(5, 21)$.

It is obvious that this is a lossy representation - it would be impossible to reconstruct the original signal using only this information. To reconstruct the true original signal in V^2 we need an additional vector that gives the rest of the data (from V^2) not representable in V^1. These would come from W^1. This is known as the detail basis and in this case represents how far the original samples are from their averages - in this case $(-1, 1)$. So using the vector from V^1 space - $(5, 21)$ - and the one from W^1 space - $(-1, 1)$ - we can represent the original signal. $(((5)+(-1), (5)-(-1), (21)+(1), (21)-(1)) = (4, 6, 22, 20))$. The vectors spanning V^j are typically denoted ϕ and are called scaling functions or father functions. Those spanning W^j are denoted ψ and are called wavelet functions or mother functions.

Now that a vague intuitive understanding of the workings of wavelets has been established, the mathematics behind wavelets will be formalised.

4.8.3 The Mathematics of Wavelets

In the previous section, we gave an example of a Wavelet Transform. In this section we expand on that same example, except that we use continuous functions.

Given a function $\psi(t)$ called the mother function, and a signal $f(t)$, the Morlet-Grossman definition of the Continuous Wavelet Transform (CWT) of a 1-dimensional signal $f(t) \in L_2$ is:

$$W(a, b) \equiv \frac{1}{\sqrt{a}} \int_{-\infty}^{\infty} f(t) \overline{\psi} \left(\frac{t-b}{a} \right) dt, \tag{4.23}$$

where $\overline{\psi}$ is the complex-conjugate of ψ.

The variable a is the scale of the mother function and b is the shift. By altering these variables, one can use the Wavelet Transform for multi-resolution analysis. For the dilations and shifts of the mother function to be orthogonal to each other:

- The shifts of the Wavelet Transform need to have non-overlapping support (see the previous section for a description of support). What this typically implies is that if the mother function $\psi(t)$ has non-zero coefficients only for some domain $x \leq t \leq y$ then the shifts must be larger than $y - x$. If the mother function is scaled to have smaller or larger support, then the shifts may be appropriately smaller or larger respectively.

- The scales of the wavelet function need to be orthogonal to each other. Designing

functions with this property can be tricky.

As can be seen, executing the above-mentioned equation (4.23), there is a single frequency resolution (scale - a value). Multiple executions with different values for a result in a multi-resolution wavelet analysis of the signal.

Converting the Haar wavelet used in the previous section to its continuous form we get:

$$\phi(x) = \begin{cases} \frac{1}{2}, & x \in [0..2] \\ 0, & x \notin [0..2]. \end{cases} \tag{4.24}$$

The Haar wavelet (detail) function is defined as:

$$\psi(x) = \begin{cases} 1, & 0 \leq x < 0.5 \\ -1, & 0.5 \leq x < 1 \\ 0, & otherwise. \end{cases} \tag{4.25}$$

These two functions are orthogonal (their inner products are zero). Furthermore the norm of both functions is equal to 1, so they are orthonormal.

Mother functions are typically dilated by powers of two. Doing this results in an orthogonal *spanning* of L_2 as described in section 4.5.3. It is also possible to dilate the mother wavelet in other ways so as to obtain the desired resolution. Doing so would not result in a minimal set of functions spanning L_2, but can prove useful in signal analysis.

Sometimes one needs to reverse the process (usually after making some adjustments to the resulting signal). The following formula shows a formula to accomplish this and more:

$$f(t) = \frac{1}{C_x} \int_0^\infty \int_{-\infty}^\infty \frac{1}{\sqrt{a}} W(a,b) \chi(\frac{x-b}{a}) \frac{da.db}{a^2}, \tag{4.26}$$

where

$$C_x = \int_0^\infty \frac{\hat{\psi}^*(v)\hat{\chi}(v)}{v} dv = \int_{-\infty}^0 \frac{\hat{\psi}^*(v)\hat{\chi}(v)}{v} dv.$$

Typically $\chi(x) = \psi(x)$, but it can be a different to enhance certain features (in which case the result would be different from the original function).

Continuous Wavelet Transform formulae are useful for deriving new wavelet functions, but are of little use to a software program. The discrete version is extremely important for the actual implementation of Wavelet Transforms, and will be discussed next.

4.8.4 The Discrete Wavelet Transform

To aid in the explanation of the application of Wavelet Transforms, we resort to using the discrete version. We again use the Haar Wavelet to assist our explanations (see section 4.8.2). The earlier explanation of what the Haar Wavelet does will be reinforced, followed by a generalisation to show how other Wavelet Transforms can be implemented in software.

We start our example with a signal consisting of eight samples:

$$f = (4, 8, 7, 13, 0, 12, 6, 2).$$

The first step of the Haar transform acquires the average shape of the signal (it takes the average of every two adjacent samples). This leaves us with:

$$\begin{aligned} \text{average} &= (\frac{4+8}{2}, \frac{7+13}{2}, \frac{0+12}{2}, \frac{6+2}{2}) \\ &= (6, 10, 6, 4). \end{aligned} \tag{4.27}$$

The next step records how far the original signal differs from this average. We take every second sample from the original (even-numbered samples), because using the odd-numbered samples merely results in the negative of these, and can therefore be discarded). This leaves us with:

$$\begin{aligned} \text{detail} &= (4-6, 7-10, 0-6, 6-4) \\ &= (-2, -3, -6, 2). \end{aligned} \tag{4.28}$$

Concatenating these two sets we have:

$$(6, 10, 6, 4, -2, -3, -6, 2).$$

The first portion of the new signal is the average of the original signal, and the latter portion is the detail part.

Digital filters (described in section 4.4) will now be incorporated into the picture, as these form the foundation for actual implementation.

Digital Filters in Wavelets

If one applies the digital filter (0.5, 0.5) this is the same as calculating the average of every pair of samples. The detail portion (of the Haar Wavelet transform) would be achieved using the filter (0.5, -0.5) on each pair of signals. Other wavelets such as the Daubechies 4 and Daubechies 6 wavelets, have longer filters.

Many texts about wavelets give the formulae in a much more compact way, which is described by Edwards [27]. This view of these filters is given by representing the two generic wavelet filters (mother and father functions or analysing and scaling functions) as two formulae taking certain coefficients C_k. These two formulae can be thought of as *low-pass* (other names are: average function, father function, scaling function or function from V^j) and *high-pass* (other names are: detail function, wavelet function, analysing function or function from W^j) filters. The low-pass filter for f can be written as:

$$a_i = \frac{1}{2}\sum_{j=0}^{N-1} C_{2i-j+1} f_j, \tag{4.29}$$

and the high-pass filter as:

$$b_i = \frac{1}{2}\sum_{j=0}^{N-1} (-1)^j C_{j-2i} f_j. \tag{4.30}$$

Using this style of writing, the Haar wavelet can be described as $C_0 = 1, C_1 = 1$ instead of the two filters $(1, 1)$ and $(1, -1)$.

According to Graps [42], these wavelet coefficients must satisfy the following constraints:

$$\sum_{k=0}^{N-1} C_k = 2, \sum_{k=0}^{N-1} C_k C_{k+2l} = 2\delta_{l,0}.$$

where δ is the delta function (see section 4.5.6) and l is the shift.

To utilise the mother wavelet to span the data domain at different resolutions, we apply the following equation [42]:

$$W(x) = \sum_{k=0}^{N-1} (-1)^{k-1} C_k \psi(2x + k), \tag{4.31}$$

where C_k are the wavelet coefficients, and $W(x)$ is the scaling function for the mother function ψ.

Knowing this syntax for representing wavelets may also help to use wavelet definitions acquired from other sources.

To summarise: the theory of wavelets was applied to discrete signals. The result was a finite summation that is implementable in software.

Having the knowledge to calculate a wavelet, however, does not necessarily provide us with clues of how they can be useful. The next section will show some wavelets widely considered more powerful and useful than the Haar wavelet.

4.8.5 More Useful Wavelets

Now that the theory of wavelets has been covered, we can advance to more useful and powerful wavelets. These wavelets are designed to produce near-zero values after the high-pass filter. One such wavelet - which is widely used in today's world - is the Daubechies-4 wavelet, named after Ingrid Daubechies who discovered it [51, 54]. She designed the wavelet function with certain properties in mind:

- The wavelet should have good support for constant signals. This implies that (1, 1, 1, 1) should be representable entirely using the low-pass filter.
- The wavelet should accurately represent linear signals (1, 2, 3, 4).
- Other properties of the different Daubechies wavelets are application specific.

The Daubechies-4 wavelet has coefficients $c_0 = \frac{1}{4}(1+\sqrt{3})$, $c_1 = \frac{1}{4}(3+\sqrt{3})$, $c_2 = \frac{1}{4}(3-\sqrt{3})$, $c_3 = \frac{1}{4}(1-\sqrt{3})$. This wavelet is commonly used, because it is a relatively small filter (and therefore fast to execute), yet very effective for signal analysis and compression.

The Daubechies-6 wavelet is even better than the Daubechies-4 wavelet for studying sounds. It's coefficients are $c_0 = 0.332671$, $c_1 = 0.806891$, $c_2 = 0.459877$, $c_3 = -0.135011$, $c_4 = -0.085441$, $c_5 = 0.035226$. There are other wavelet filters with huge numbers of coefficients designed to represent many common nuances expected in a signal.

Like the Haar wavelet these wavelets are applied to the signal, then shifted two samples on. Doing this results in a new signal the same size as the original. This is slightly different behaviour to standard convolution (see equation 4.8).

Converting a signal using Wavelet or Fourier Transforms is of little use, unless one

knows how to actually interpret the results of the transformation. In the next section, feature extraction techniques (for both Fourier and Wavelet data) will be described.

4.9 Feature Extraction Techniques

Once one has transformed the signal to a different set of bases, it becomes easier to extract features from the signal that can be used for analysis. Extracting features from the two above-mentioned transforms (Fourier Transform and Wavelet Transform) requires some additional processing. This process will first be applied to the Fourier Transform, then to the Wavelet Transform.

4.9.1 Processing of Fourier Data

Fourier Transforms result in complex coefficients - the real portion and imaginary portions being coefficients of sinusoids that are $90°$ out of synchronisation (sine waves and cosine waves). A shift in the original signal could alter the actual values of these complex numbers, but not their norms (magnitudes) (see section 4.7.5).

So if one uses only the norms of these coefficients, or even the energy (square of the norms - chosen due to the fact that we then need not execute the square-root function, which is slow), then the features extracted from a given signal will remain unchanged, regardless of shift.

Merely using these values however, seldom causes a classification system to converge (to consistently classify signals). In order to accurately train a classification network using Fourier transformed information, one more step is required. One needs to extract the cepstral coefficients [88].

The log cepstrum (the word spectrum with the first syllable reversed) is defined as:

$$\mathrm{cepstrum}(x) = \mathrm{FT}(\log(\mathrm{FT}(x))), \tag{4.32}$$

where FT is the Fourier Transform.

Note: some texts incorrectly define the cepstrum as:

$$\mathrm{cepstrum}(x) = \mathrm{IFT}(\log(\mathrm{FT}(x))),$$

where IFT is the *Inverse* Fourier Transform.

Often the $\log$ scale used is the Mel scale. This scale is a scale of pitches judged by human listeners to be equidistant from one another. To convert fHz to the Mel scale we use the following formula:

$$m(f) = 1127.01048 \times \log(1 + f/700). \tag{4.33}$$

The logarithmic cepstrum is not the only type of cepstral coefficient that can be used. Success has been reported by people using the root-cepstrum (using a square-root or other root function instead of the logarithm) as well [77]. Once the cepstrum has been calculated, the first 20-50 coefficients of the result are usually sufficient for most sound classification applications.

In the next section, a similar process for wavelet data will be described.

4.9.2 Processing of Wavelet Information

Wavelet Transforms result in information that is decidedly different to the results of the Fourier Transforms:

- While a single Fourier Transform results in energy levels across the entire spectrum of frequency ranges, and are assumed constant throughout the duration of the given frame (see section 4.10), wavelets typically produce results for frequency bands one octave apart (double the frequency each time), and the results usually vary over the duration of the frame. This is caused by localised features in the original signal.

- By using the norm of the complex values returned after a Fourier Transform, one can accurately determine the energy of a given frequency. With the results of the wavelet, the results are typically similar to the original signal, only 'averaged' in some way. For this reason some additional processing is required after a Wavelet Transform before frequency amplitudes can be determined.

- By using the above-mentioned norm (see section 4.9.1) after a Fourier Transform, the results become shift invariant. Wavelets also suffer from shift variance unless corrective measures are taken. Farooq and Datta [35] describe a technique to overcome this problem. They relate acquiring energy distributions using Short-Time Fourier Transforms (STFT) and acquiring bandwidth-limited energy distributions from Discrete Wavelet Transform (DWT) data. By retrieving the energy distributions from a given frequency band, the problem of shift variance is overcome.

Once one has obtained energy readings from the various frequency bands returned by the DWT, those energy values can be used as features for the classification process. The

features are typically passed to a neural-network (see section 5.2) or a Hidden Markov Model (section 5.3) for the actual identification process.

Next the technique of using Frames and Windows will be described, as these can minimise frequency leakage of basis transformation functions especially the Fourier Transform.

4.10 Frames and Windows

Because of the fact that

1. the signals need to be processed in real-time (ie. processing starts before the entire signal has been acquired) and

2. the signal varies (sometimes sharply) over time,

the entire signal is broken up into small processable sections (frames), say perhaps $20ms$ long. Hopefully each frame accurately portrays the desired features of the signal over the domain near the frame. When analysis techniques are used to study these frames, we expect to get a fairly clear idea of the general shape of the signal over time.

Unfortunately, due to the fact that the signal *does* change within a frame, there is often a distinct discontinuity between transformed samples at the borders between adjacent frames. To avoid this behaviour, a commonly used technique is to overlap the frames (eg. take 20ms-long frames once every 10ms). A complementary technique is to diminish the signal at the ends of the frame, so that the samples from the central region of the frame contribute most to the understanding of the features of the frame. The latter technique is called windowing.

Typically the effective domain (support) of a windowing function $w(x)$ is $0 \leq x \leq 1$ (ie. $w(x) = 0, x \notin [0..1]$). Towards 0 and 1 the signal is diminished (attenuated). In so doing, the continuity of signals between frames is enhanced. Bores [5] describes this continuity as: "Put mathematically, a window function has the property that its value and all its derivatives are zero at the ends". By using a windowing function that has the properties Bores describes, the problem of frequency leakage is eliminated, as the function becomes periodic over the duration of the window. This results in better frequency resolution. Windowing functions can be scaled and shifted to operate on different sized windows and windows not starting at time $t = 0$.

The most common windowing function is the *rectangular* window which leaves the

values untouched within the window ($0 \leq x \leq N-1$, where N is the number of samples desired) and makes the values 0 outside the window. (Technically this is merely a frame). This function though, does not fulfill the above mentioned property of good windowing functions. Table 4.2 shows some other common windowing functions [58], [73].

Name	**Formula**
Rectangular	$w(n) = \begin{cases} 1, & 0 \leq n \leq N-1 \\ 0, & \text{elsewhere} \end{cases}$
Bartlett	$w(n) = \begin{cases} 2n/(N-1), & 0 \leq n \leq (N-1)/2 \\ 2 - 2n/(N-1), & (N-1)/2 \leq n \leq N-1 \\ 0, & \text{elsewhere} \end{cases}$
Hanning	$w(n) = \begin{cases} (1 - cos(2\pi n/(N-1))/2, & 0 \leq n \leq N-1 \\ 0, & \text{elsewhere} \end{cases}$
Hamming	$w(n) = \begin{cases} 0.54 - 0.46cos(2\pi n/(N-1)), & 0 \leq n \leq N-1 \\ 0, & \text{elsewhere} \end{cases}$
Blackman	$w(n) = \begin{cases} 0.42 \quad -0.5cos(2\pi n/(N-1)) \\ \qquad +0.08cos(4\pi n/(N-1)), & 0 \leq n \leq N-1 \\ 0, & \text{elsewhere} \end{cases}$

Table 4.2: Definitions of some common windowing functions

Using Windowing Functions

Windowing functions ($\omega(t)$) are multiplied with the original signal ($f(t)$) to return a new function ($g(t)$). Notice that this is not a convolution as with digital filters, as there is a one-to-one mapping of samples of the window and the signal itself. This is the formula for multiplication by a window:

$$g(t) = \omega(N(t - \tau))f(t), \tag{4.34}$$

where τ represents the start of the effective window and N the size of the window.

Of the windowing functions defined in table 4.2, one that results in very little distortion after a Fourier Transform is the Blackman windowing function. (Its derivatives are all zero at $n = 0$ and at $n = 1$). Figure 4.8 shows the effect of applying the Blackman windowing function.

This work by no means covers the entire spectrum of DSP techniques, and further resources on the topic are provided in the next section.

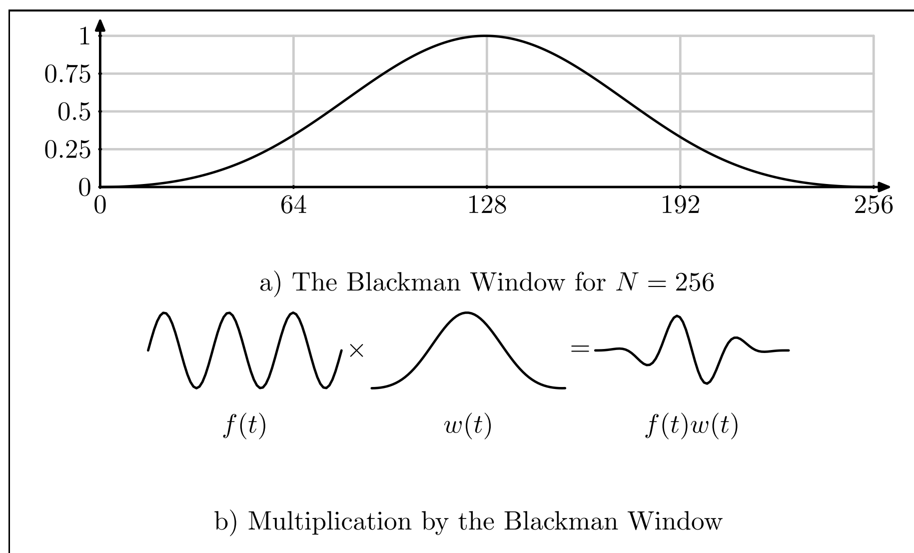


Figure 4.8: The effects of the Blackman windowing function [5]

4.11 Additional DSP References

DSP is a vast subject, and there are many other sources of information. [73] provides an applet which illustrates the DSP functions. [11], [20], [36], [46], and [61] give further insight into Fourier Transforms. [39] is a library of software that provides Fourier Transform functionality in extremely optimised code. [6] and [16] give a linear predictive coding approach to DSP. [10], [41], [82] and [83] give some further background about wavelets, and [35], [43], [57], [84], and [85] give more information about using wavelets for phoneme identification.

4.12 Summary

In this chapter we studied digital signal processing techniques. Focus was placed on the analysis of speech signals. Different change of base transformations were also described. Techniques for extracting feature sets from the signal were also mentioned. The chapter concluded with ways of dividing a signal into smaller units and enhancing those units using windowing algorithms.

Now that DSP techniques have been described, the following chapter will show how to use the results these techniques provide to classify signals.

Chapter 5

Classifi cation

In the previous chapter we illustrated ways to extract feature information from a sound signal. In this chapter we continue the analysis process by illustrating various techniques for classifying those feature sets.

This chapter is divided into the following sections:

- Introduction - this is an introduction to classification systems in general, and describes some of the aspects that are common to all classification systems.
- Neural Networks - this section describes neural networks, which are one of the most popular techniques for classifying phonemes.
- Hidden Markov Models - these are also widely used to solve speech recognition problems.
- Classification Summary

5.1 Introduction

Classification systems are designed to classify a set of input information. For example if one gives a fruit classification system the inputs: “red”, “heart-shaped” and “smooth”, it might place that input vector into the “apple” category.

In order to make such a classification, certain calculations need to be performed on the data. The types of calculations performed vary greatly between classification systems. The actual data used as input also plays a role in deciding which calculations to perform.

Very frequently the formulae of these calculations are fixed in structure, but certain constants of the function are adjusted in order to produce the correct output. This adjustment process is usually the hardest part of creating classification systems due to

the number of calculations and size of input vectors. As a result, adjustment of the constants is seldom accomplished through a manual process. A better technique that is often used is to give the system many sets of input vectors and inform it what the desired output vector for each input vector is. Some learning algorithm is then used to adjust the constants in the formulae to better produce the desired output. By repeating this process sufficiently many times, it is hoped that the system will (eventually) consistently produce the correct output vector given some input vector.

This learning process can be achieved in many different ways, and so forms much of the theory of classification systems.

Unfortunately though, the success of most classification systems depends on the data itself. Designing the structure (formulae) of the classification system often requires some trial and error. The number and types of functions play a large role in the success of the system.

Classifications systems are very diversified in their applications, accuracy, operation and speed. The remainder of this chapter deals with specific classification algorithms, starting with *neural networks*.

5.2 Neural Networks

The first classification system we will discuss is the neural network - modelled after the human brain. At present there is no better classification system than the human brain for solving *general* classification problems. When we look around us, we can immediately identify what objects are present. When seeing people we know, our brains can identify them based on minute details. We usually don't even know that we are thinking about when identifying people, but the calculations performed by our brains are extremely complicated.

There are some systems that can classify certain inputs better or faster, but the human brain consistently performs well for nearly all classifications. Each brain-cell (neuron) takes certain chemical inputs (neurotransmitters), which are processed to produce a certain amount of an output chemical. This process, occurring in a single cell, is called activation. The structure of the brain causes a cell to receive as chemical input, the output chemicals of thousands of other cells. This is the way in which we think and relate ideas [14].

Computer-based neural networks attempt to model the brain's structure mathematically.

The network takes several external inputs (an input vector) which are passed to a number of neurons. These inputs are typically real numbers. The neurons do the following:

- weight the inputs (multiply them by a constant),
- sum the weighted inputs together,
- optionally add a constant value to this sum,
- pass that sum (activation input) to an activation function, and
- output the result of the computation of the activation function.

The above process occurs concurrently in many, many neurons of the human brain. Each applicable neuron receives the same input at the same time. This is why the brain can perform such complex classifications in so little time. A similar concept happens in the computer version, except that the calculations are done for one neuron at a time. A set of neurons where each neuron receives the same input as the others is called a *neuron layer*.

Neural networks are usually created by fully connecting several neuron layers to one another - all the outputs from the current layer form the inputs to the next layer. The size of the output vector of the neural network as a whole is the same as the number of neurons in the final layer of the network. The actual weight values and activation functions for the individual neurons in the structure may differ from neuron to neuron. See figure 5.1 for an illustration of the structure of a neural network.

It is also possible to utilise other structures of neural networks [25], but for purposes of the problems dealt with in this document, the network structure described above is sufficient.

The convention used in this document is that:

- There are L layers.
- The first layer is layer 0, and the last is layer $L-1$.
- Each layer i contains $J(i)$ neurons.
- The first neuron in layer i is $N(i,0)$, the last is $N(i,J(i)-1)$.
- Input k to $N(i,j)$ is $I(i,j,k)$.
- The weight for $I(i,j,k)$ is $W(i,j,k)$.

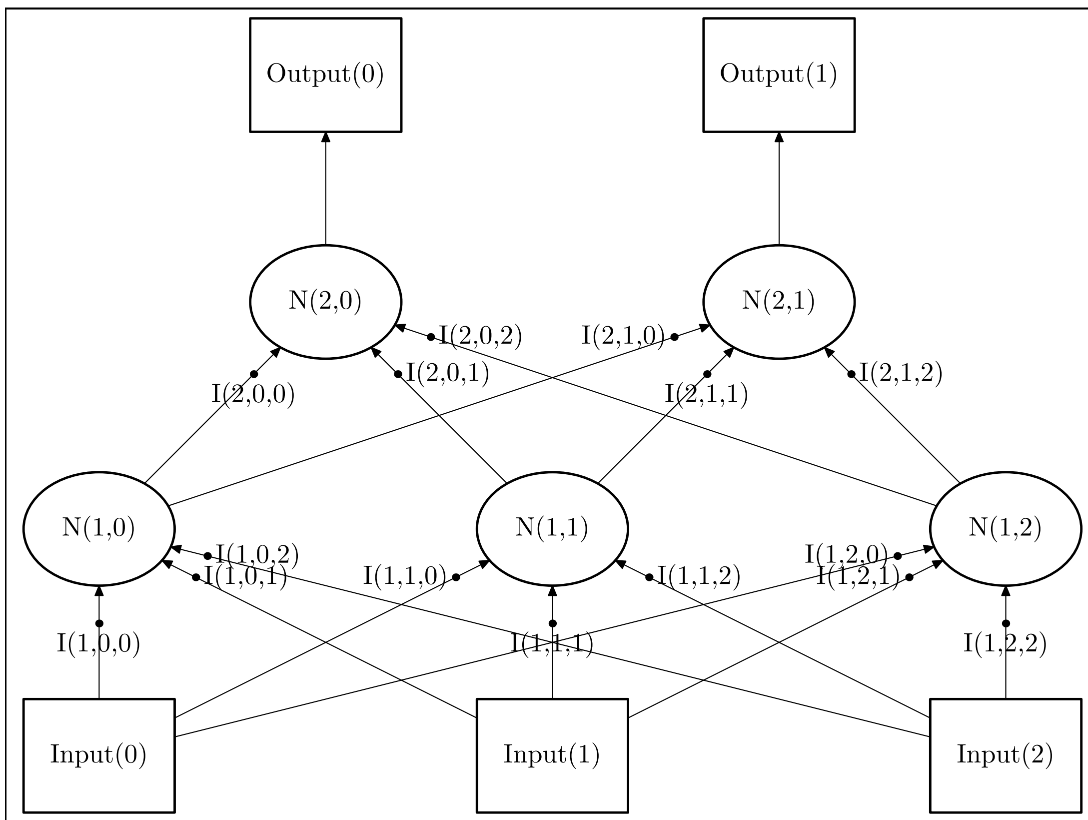


Figure 5.1: A neural network structure

- The output of $N(i,j)$ is $O(i,j)$.
- Because neurons in a layer are grouped, the neurons, inputs, weights and outputs can also be thought of as vectors.
- The set of neurons in layer i can be thought of as $\mathbf{N}(i)$.
- The set of inputs to $N(i,j)$ written as a vector are $\mathbf{I}(i,j)$.
- The set of weights for $\mathbf{N}(i,j)$ written as a vector are $\mathbf{W}(i,j)$.
- The set of outputs for layer i can be written as $\mathbf{O}(i)$.
- Note: there is usually an additional input of constant value 1 (with its own weight value) to each node in the entire network. This implies that there are $J(i-1)+1$ inputs to $N(i,j)$. These are $\mathbf{O}(i-1)$ and 1.

An important aspect of a neural network is the activation function used. Table 5.1 is a table of some of the most commonly used activation functions. Unless otherwise specified, the activation function used in examples in this document is the *log-sigmoid* function.

Name	Formula
Linear	$f(x) = x$
Saturating Linear	$f(x) = \begin{cases} 0, & \text{if } x < 0 \\ x, & \text{if } 0 \leq x < 1 \\ 1, & \text{if } 1 \leq x \end{cases}$
Symmetrical Saturating Linear	$f(x) = \begin{cases} -1, & \text{if } x < -1 \\ x, & \text{if } -1 \leq x < 1 \\ 1, & \text{if } 1 \leq x \end{cases}$
Hard Limit	$f(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$
Symmetrical Hard Limit	$f(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ -1, & \text{if } x < 0 \end{cases}$
Log-Sigmoid	$f(x) = \frac{1}{1+e^{-x}}$
Symmetrical Log-Sigmoid	$f(x) = \frac{2}{1+e^{-x}} - 1$
Hyperbolic Tangent Sigmoid	$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
Competitive	$f(x) = \begin{cases} 1, & \text{if } x \text{ is highest in network layer} \\ 0, & \text{otherwise} \end{cases}$

Table 5.1: Activation functions

Now that we have shown how neural networks are structured, and the general idea behind their workings, we will describe how they are trained.

5.2.1 Training Neural Networks

The weight vectors of a neural network need to be properly initialised in order for the network to give an appropriate output vector, given some input vector. These values are seldom assigned manually (it would take too long) - they are usually assigned through a training process.

The training of a neural network involves providing the network with an input vector and the desired output vector for that input vector. The neural network weight values are then adjusted so that the network better produces the desired output from the given inputs. This process is repeated until the network consistently produces satisfactory classifications for all classes of inputs. There are several training algorithms that can be applied to train neural networks, but most of them are variants of the *gradient-descent* method.

Gradient-descent works by adjusting weights based on how they each contribute to the output error $E(i,j)$ value for neuron $N(i,j)$. The goal is to cause each weight to have no contribution to the error function (which means the error would be zero, which implies that the result would be accurate). Each weight $W(i,j,k)$ is increased or decreased depending on whether the gradient function $\frac{\delta E(i,j)}{\delta W(i,j,k)}$ is increasing or decreasing.

To calculate the error gradient for each weight in the network the following algorithm is executed:

```
//L = number of layers
//therefore L-1 = number of the last layer
//J(i) = number of nodes in layer i
//therefore J(i)-1 is the number of the last node in layer i
//N(i,j) = jth node in layer i
//I(i,j,k) = input k for N(i,j)
//W(i,j,k) = weight k for N(i,j)

//desired_Outputs is an array of the desired outputs for the nodes
//in the output layer
calculate_All_Gradients (desired_Outputs)
{
  for j = 0 to J(L-1)-1
  {
    calculate_Error_Gradients_Output_Node (desired_Outputs [j], j)
  }

  for i = L-2 downto 0
  {
    for j = 0 to J(i)-1
    {
      calculate_Error_Gradients_Non_Output_Node (i,j)
    }
  }
}

//desired_Output is the desired output for the node
//j is the number of the node in the output layer
calculate_Error_Gradients_Output_Node (desired_Output, j)
{
  //calculate dE(L-1, j) / dOutput (L-1, j)
  dE_dOutput = O(L-1,j)-desired_Output

  //Save the result for use in other parts of algorithm
  N(L-1,j).dE_dOutput = dE_dOutput

  //Calculate activation function specific gradient
  //This is the activation function for N(L-1, j)
```

```
  dOutput_dActivation = activation_Function_Gradient (O(L-1,j))

  //Calculate gradients for the individual weights of the node
  //including the constant 1 input
  for k = 0 to J(L-1)
  {
    dOutput_dW = dOutput_dActivation * I(L-1, j, k)
    W(L-1, j, k).gradient = dE_dOutput * dOutput_dW
  }
}

calculate_Error_Gradients_Non_Ouput_Node (i, j)
{
  //Calculate how this whole node contributes to all errors
  //in all outputs
  dE_dOutput = 0
  for k = 0 to J(i+1)-1
  {
    //Calculate activation function specific gradient
    //This is the activation function for N(i+1, k)
    dOutput_dActivationj2 = activation_Function_Gradient (O(i+1,k))

    dE_dOutput += dOutput_dActivationj2 * W(i+1, k)
  }

  dOutput_dActivation = activation_Function_Gradient (O(i,j))
  //Adjust weights for all inputs including the constant 1 input
  for k = 0 to J(i)
  {
    W(i,j,k).gradient = dE_dOutput * dOutput_dActivation * I(i,j,k)
  }
}
```

In the algorithm, it is shown that to calculate $\frac{\delta E(i,j)}{\delta W(i,j,k)}$, we use the following chain rule:

$$
\begin{aligned}
\frac{\delta E(i,j)}{\delta W(i,j,k)} &= \frac{\delta E(i,j)}{\delta O(i,j)} \cdot \frac{\delta O(i,j)}{\delta W(i,j,k)} \\
&= \frac{\delta E(i,j)}{\delta O(i,j)} \cdot \frac{\delta O(i,j)}{\delta Activation(i,j)} \cdot \frac{\delta Activation(i,j)}{\delta W(i,j,k)}.
\end{aligned}
$$

To apply the algorithm described above requires (inter alia) $\frac{\delta output}{\delta activation}$ (or $\frac{\delta O(i,j)}{\delta Activation(i,j)}$) to be calculated. This is the only portion of the gradient function that differs for different

activation functions. Examples in this document use the *log-sigmoid* transfer function. $\frac{\delta O(i,j)}{\delta Activation(i,j)}$ for this function (log-sigmoid) is $O(i,j)(1 - O(i,j))$.

Once the error gradient has been calculated for each weight in the network, we apply a training algorithm that adjusts the weights. We will first describe the *Widrow-Hoff* algorithm, then show a more effective gradient descent algorithm called *resilient propagation*.

Widrow-Hoff

The Widrow-Hoff gradient descent method simply determines the sign of the error gradient for a given weight. If the sign is positive, we add an input-weighted learning value to the weight. If the sign is negative we subtract. In other words the amount we add or subtract is proportional to the magnitude of the input [64, p41].

The Widrow-Hoff method was one of the first methods to be used, but it is dependant on the size of the inputs. This is a problem because weights associated with small input values may need to be adjusted by large amounts. This would not happen – the weights would be adjusted by small amounts, thereby causing the required number of training iterations to be increased. Similarly weights associated with large-value inputs would be adjusted greatly, even if only a small adjustment is needed. This overcompensation may cause the training process to require more iterations before it converges or – in the worst case – to never converge (to be divergent).

In summary the Widrow-Hoff method causes convergence to be relatively slow (for small inputs) and unstable (overcompensation due to large inputs). A better approach would be one where the speed of the training is not dependant on the magnitude of the input values. One such approach is *resilient propagation*.

Resilient Propagation

As mentioned above, the Widrow-Hoff method for adjusting weight vectors does not always produce stable results. A better approach is to give each weight change a velocity, and to accelerate (η^+) them while they move in the same direction, and drop the speed suddenly (η^-) if they change direction. This causes the changes to be irrespective of input magnitude and the algorithm as a whole is therefore more stable. This stability usually results in faster convergence. This technique is called *resilient propagation* (RProp) due to the resilience of changes to weight vectors [72].

Initial velocities of learning rate components Δ_{ij}^t are usually set to 0.1 (although other

values may be used in some cases). Changes to the Δ values are made according to the following formula:

$$\Delta_{ij}^{(t)} = \begin{cases} \eta^+ . \Delta_{ij}^{(t-1)}, & \frac{\delta E^{(t-1)}}{\delta w_{ij}} . \frac{\delta E^{(t)}}{\delta w_{ij}} > 0 \\ \eta^- . \Delta_{ij}^{(t-1)}, & \frac{\delta E^{(t-1)}}{\delta w_{ij}} . \frac{\delta E^{(t)}}{\delta w_{ij}} < 0 \\ \Delta_{ij}^{(t-1)}, & \text{otherwise,} \end{cases} \tag{5.1}$$

where the default values for η^+ and η^- are 1.2 and 0.5 respectively. (These constants are suitable for many applications and should seldom need to be changed [72]).

Once these Δ values have been calculated, the weights are updated as follows:

$$w_{ij}(t) = w_{ij}(t-1) + \begin{cases} -\Delta_{ij}(t), & \frac{\delta E}{\delta w_{ij}}(t) > 0 \\ +\Delta_{ij}(t), & \frac{\delta E}{\delta w_{ij}}(t) < 0 \\ 0, & \text{otherwise.} \end{cases} \tag{5.2}$$

It has been shown that RProp performs very well in most applications of neural networks [72]. For descriptions of additional training algorithms see [72].

In the next section, we will describe a technique that can be applied together with other learning techniques to improve the stability of the learning process - *epoched training*.

Epoched Training

When training a neural network, the weights are often over-adjusted due to some input vector causing a steep error gradient for one or more weights. To overcome this problem, error gradients are averaged over a set of many inputs before adjustment of the weights occurs. This causes only those weights that should be adjusted greatly for *all* classes of inputs to receive large adjustments [72].

Overall this causes a smoother learning curve, with very few over-compensations due to one obscure input set. The technique mentioned here is called epoched training because each training cycle occurs only after a period of time (multiple inputs). The word *epoch* means *a time period*.

5.2.2 Identifying Phonemes Using Neural Networks

Using the descriptions mentioned in the previous sections, we can build a neural network that is capable of classifying phonemes in an audio stream. In chapter 4.9.1, we showed how to extract *cepstral coefficients* from an audio stream using Fourier Transforms. These coefficients make up the input vectors to our neural network.

As was mentioned earlier, the first 20 to 50 of these coefficients is sufficient to accurately classify phonemes in an audio stream (also see chapter 13). To perform this classification however, we need to understand what kinds of problems neural networks can solve.

Due to the fact that the activation function can be altered, a neural network can solve a wide variety of problems. Using the *log-sigmoid* function, a neural network can typically answer a yes/no type question. Given inputs, a network is trained to output a 1 if the answer is yes or a 0 if the answer is no (or vice versa).

Going back to the example described at the start of this chapter, we can use a neural network to tell us whether a fruit is an apple. Given "redness", "roundness" and "smoothness" values each from say 0 to 1, we can output a 0 (not an apple) or a 1 (really an apple). Results are not always exactly 0 or 1, but are sufficiently close.

In a similar manner, we can use neural networks to classify sounds progressively. The first neural network could classify the sound as either voiced or unvoiced. (See chapter 3). Once we know whether the sound is voiced or unvoiced, we can further classify it.

By passing the sound through several steps of the classification process, we can classify the sound. Very often the process may end up deducing that the sound could be one of a set of similar sounding phonemes (each with a specific probability) due to noise, vocal accents or poor pronunciation. If one listens to one of the sound extracts that is inconclusively classified, it is found that even the human brain might be confused. Only when the whole utterance is heard can we be sure which phoneme the extract represented. Therefore the context in which the phoneme occurred (previous phonemetic data acquired) can aid the classification process substantially.

5.2.3 Additional References

Due to the fact that neural networks are typically designed around the problem they are to solve, the field of neural networks is vast. For additional tutorial matter see [68]. For tools to help design neural networks see [71]. For a neural network application written in Java see [4].

5.2.4 Summary

It has been shown that neural networks can be used to fairly accurately classify sounds into categories of phonemes [92]. This process usually follows a hierarchy of classifi-

cations. We also know that sometimes sounds cannot be completely classified (even by a human). In the next section it will be shown how to also use *Hidden Markov Models* for selected speech processing problems.

5.3 Hidden Markov Models

In chapter 3 we showed that phonemes are the building blocks of words, and in the previous sections we showed how to use neural networks to classify audio extracts (frames) into phonemes. We also showed that sometimes audio cannot be exactly classified, it can only be narrowed down to a set of phonemes. In this section we will show how *Hidden Markov Models* (HMMs) can be used to string these phonemes toegether to make words.

There are two main reasons one would want to know what words are being spoken:

- To gain an *understanding* of what is being communicated - or to attribute meaning to the sentences
- To help clarify which phoneme was uttered. Often we can identify an unclear phoneme by hearing the rest of the word in which it was spoken.

This section on HMMs is structured as follows:

- How Hidden Markov Models Work
- The Three HMM Problems
- Applying HMMs To Word Recognition

5.3.1 How Hidden Markov Models Work

Hidden Markov Models are used to determine the probability of a sequence of events occurring, given certain observations. To make this concept clear we will start with an example.

Suppose we want to guess which beverages a certain colleague consumed over the last few days - decaffeinated coffee (D) or standard coffee (S). We don't know for certain, because both beverages look the same, but we can guess based on indirect evidence.

Suppose that research has shown that if a person drinks decaffeinated coffee one day, the chances of them drinking it again the next day is 0.7. If they drink standard coffee

one day, then the chance they'll drink it again the next day is 0.5. Using this information we can draw up a table where the vertical axis is what the person drank today, then horizontal is what they might drink tomorrow, and the values in the table are the probabilities.

	Decaffienated	Standard
Decaffienated	0.7	0.3
Standard	0.5	0.5

Table 5.2: HMM state transition matrix

We know that caffeine stimulates the adrenal gland to produce adrenaline. There could conceivably be a link between coffee consumption and observed energy levels (low, medium and high) or (0, 1 and 2) with the following probabilities:

	Low Energy	Medium Energy	High Energy
Decaffienated	0.7	0.2	0.1
Standard	0.1	0.2	0.7

Table 5.3: HMM observation matrix

Suppose we know that over the years the probability of a person drinking decaffeinated coffee was 0.5.

As we mentioned earlier our goal is to observe whether or not the person drank decaffeinated coffee or standard coffee on a given day. These are called *states* of the Hidden Markov Model, and the process of determining a state given the previous one is called the *Markov Process*.

The state transition matrix we label $\mathbf{A}$, and the observation matrix $\mathbf{B}$. There is an initial state distribution $\pi = [0.5, 0.5]$. We will consider the days Monday to Thursday for our observations, where we observed the following energy levels in our colleague: $O = (Low, Medium, Medium, High) = (0, 1, 1, 2)$.

This problem specification is sufficient to understand the general types of problems solved by HMMs. We leave the actual solution to this problem for a later section, but first we will give some notes about the notation of Hidden Markov Models.

5.3.2 HMM Notation

In HMM formulae the following symbols are used:

- T is the length of the observation sequence - in our case Monday to Thursday = 4.

- N is the number of states possible - in our case decaffeinated or standard = 2.
- M is the number of observation symbols - in our case low energy, medium energy and high energy = 3 symbols.
- **Q** is the vector of the actual states of the Markov Process = $Q = (Q_0, ..., Q_{N-1})$ - in our case the type of coffee consumed each day.
- **V** is the set of all possible observations = $V = (V_0, ..., V_{M-1}) = (0, ..., M-1)$.
- **A** is the state transition matrix - in our case table 5.2.
  $a_{i,j} = P(q_j \text{ at time } t+1 | q_i \text{ at time } t)$ (note that $(a|b)$ means the probability of a occurring given the fact that b has occurred).
- **B** is the observation probability matrix - table 5.3
  $b_{j,k} = P(V_k \text{ at time } t | q_j \text{ at time } t)$.
- π is the initial state distribution of the problem.
- **O** is the actual observation sequence - in our case the energy levels observed - $(0, 1, 1, 2)$.

The HMM is defined by A, B and π and is denoted $\lambda = (A, B, \pi)$.

Now that we have a notational framework from which to work, we can set about showing how to solve HMM problems.

5.3.3 Solving HMM Problems

We will first show step by step how to solve the coffee problem described earlier, then generalise to show how HMM problems are solved in general. We want to calculate the probability of a given sequence of states $\mathbf{X} = (x_0, x_1, x_2, x_3)$:

$$P(\mathbf{X}) = (\pi_{x_0} \times b_{x_0,O_o})(a_{x_0,x_1} \times b_{x_1,O_1})(a_{x_1,x_2} \times b_{x_2,O_2})(a_{x_2,x_3} \times b_{x_3,O_3}) \tag{5.3}$$

So given the fact that we observed $O = (L, M, M, H) = (0, 1, 1, 2)$, the probability of $X = (D, D, S, S) = (0, 0, 1, 1)$ is:

$$P(D, D, S, S) = (0.5 \times 0.7)(0.7 \times 0.2)(0.3 \times 0.2)(0.5 \times 0.7) = 0.001029$$

The other results are listed in table 5.4. In the table, the last column is the normalised values. There are two ways to interpret the 'best' result from this table:

State	Probability	Normalised Probability
D D D D	0.0004802	0.0926883878937616
D D D S	0.0014406	0.278065163681285
D D S D	0.000147	0.0283739962940086
D D S S	0.001029	0.19861797405806
D S D D	0.000147	0.0283739962940086
D S D S	0.000441	0.0851219888820259
D S S D	0.000105	0.0202671402100062
D S S S	0.000735	0.141869981470043
S D D D	0.000049	0.00945799876466955
S D D S	0.000147	0.0283739962940086
S D S D	0.000015	0.0028953057442866
S D S S	0.000105	0.0202671402100062
S S D D	0.000035	0.00675571340333539
S S D S	0.000105	0.0202671402100062
S S S D	0.000025	0.00482550957381099
S S S S	0.000175	0.033778567016677

Table 5.4: HMM state sequence probabilities

- Choose the sequence of the highest probability - in our case (D, D, D, S).
- Choose states in such a way that the expected number of states correctly predicted is maximised.

The latter is the answer looked for when using HMMs. To find this answer we need to take the sum of the probabilities of sequences that have a specific symbol in a specific position. As an example we will decide whether to use D or S as the first state.

We need to sum the probabilities of all the sequences that have a D as the first state. This would be the sum of the normalised probabilities of the first 8 sequences in the table (0.8733786287831989). The probabilities of sequences with an S as the first state sum to 0.1266213712168011. We can therefore see that there should be a D in the first position.

By continuing the process we can see, for this example, that by maximising the expected number of correct states, we also end up with the sequence D, D, D, S. It is not always the case that the optimal state sequence is the same as the optimal state at each position. By maximising the expected number of correct states one might actually end up with a state sequence that is not actually possible.

Now that we have seen one example worked through, we will explain the HMM problems in general, then show how they apply to speech recognition.

5.3.4 The Three HMM Problems

As we saw in the previous chapter, the HMM process maximises the probabilities of specific states being observed at specific places in the sequence. There are however three different problems that can be solved using HMM techniques, depending on what information is already known, and what results are required [70], [81]:

1. If we have been given the HMM model λ, we can use HMM techniques to determine the probabilities of observing sequence $\mathbf{O}$. (In our example we would know what he drank on given days, but would use HMM to determine the probabilities of observing certain energy levels in our colleague).

2. If we have an observation sequence $\mathbf{O}$ and state transition probabilities ($\mathbf{A}$, $\mathbf{B}$ and π), we can maximise the expected number of correctly predicted states at specific positions in the sequence. This is exactly the problem that was solved earlier.

3. Given the observation sequence $\mathbf{O}$ and the state sequence $\mathbf{Q}$, we can determine the HMM model. In other words we can use HMMs to calculate $\mathbf{A}$, $\mathbf{B}$ and π. It is usually assumed that T, N and M are fixed. This is the problem that one needs to solve in speech recognition applications.

Solving the First HMM Problem

The solution to the first problem requires nothing but standard probability theory. We wish to calculate the probability of observing sequence $\mathbf{O}$ given Markov Model λ - $P(\mathbf{O}|\lambda)$. Because the information we seek is irrespective of the states, we basically need to sum all possibilities over all possible state sequences (X).

For more details about the following formulae see [70] and [81].

We start by using the arbitrary state sequence:

$$(X) = (x_0, ..., x_{T-1}).$$

The following two equations are described because they will be used later. By definition:

$$P(\mathbf{O}|\mathbf{X}, \lambda) = b_{x_0,O_0} \times b_{x_1,O_1} \times ... \times b_{x_{T-1},O_{T-1}}. \tag{5.4}$$

We also know that:

$$P(\mathbf{X}|\lambda) = \pi_{x_0} \times a_{x_0,x_1} \times a_{x_1,x_2} \times ... \times a_{x_{T-2},x_{T-1}}. \tag{5.5}$$

We know (by probability theory) that:

$$P(O, \mathbf{X}|\lambda) = \frac{P(O \cap \mathbf{X} \cap \lambda)}{P(\lambda)}, \tag{5.6}$$

and

$$\begin{aligned} P(\mathbf{O}|\mathbf{X}, \lambda) \times P(\mathbf{X}|\lambda) &= \frac{P(\mathbf{O} \cap \mathbf{X} \cap \lambda)}{P(\mathbf{X} \cap \lambda)} \times \frac{P(\mathbf{X} \cap \lambda)}{P(\lambda)} \\ &= \frac{P(\mathbf{O} \cap \mathbf{X} \cap \lambda)}{P(\lambda)}. \end{aligned} \tag{5.7}$$

From equations 5.6 and 5.7 we have:

$$P(\mathbf{O}, \mathbf{X}|\lambda) = P(\mathbf{O}|\mathbf{X}, \lambda) \times P(\mathbf{X}|\lambda). \tag{5.8}$$

And so we have:

$$\begin{aligned} P(\mathbf{O}|\lambda) &= \sum_{\mathbf{X}} P(\mathbf{O}, \mathbf{X}|\lambda) \\ &= \sum_{\mathbf{X}} P(\mathbf{O}|\mathbf{X}, \lambda) \times P(\mathbf{X}|\lambda) \text{ (using equation 5.8)} \\ &= \sum_{\mathbf{X}} (\pi_{x_0} b_{x_0,O_0})(a_{x_0,x_1} b_{x_1,O_1})...(a_{x_{T-2},x_{T-1}} b_{x_{T-1},O_{T-1}}) \text{ (eqn 5.4 and 5.5).} \end{aligned}$$

Unfortunately to iterate over all possible combinations of $\mathbf{X}$ takes N^T cycles, and inside the summation there are roughly $2T$ multiplications (T for a and another T for b). This leaves us with an $O(TN^T)$ complex algorithm, which is terrible.

There is a recursive algorithm that can reduce this to $O(TN^2)$, which is far better.

We define an α pass to be the probability of the observation sequence occurring up until time t:

$$\alpha_t(i) = P(\mathbf{O}_0, \mathbf{O}_1, ..., \mathbf{O}_t, x_t = q_i|\lambda). \tag{5.9}$$

By definition:

$$\alpha_0(i) = \pi_i b_{i,O_0}, \tag{5.10}$$

and

$$\alpha_t(i) = (\sum_{j=0}^{N-1} \alpha_{t-1}(j) a_{ji}) b_{i,O_t}. \tag{5.11}$$

From equation 5.9 we can then see that:

$$P(\mathbf{O}|\lambda) = \sum_{i=0}^{N-1} \alpha_{T-1}(i). \tag{5.12}$$

It can be seen that the two summations cause N^2 complexity, and the recursive function is of T complexity, so the algorithm as a whole executes in $O(TN^2)$ time.

Using these results, specifically the α calculation, we can also solve the second HMM problem.

Solving the Second HMM Problem

The example problem we described at the start of the HMM section happens to be an instance of the second HMM problem. We are given an HMM model λ (fixed values T, M, N, A, B, π) and a sequence of actual observations $\mathbf{O}$. We need to find the most likely state sequence $\mathbf{Q}$. By that we mean the most likely state at each position in the sequence. As with the previous section, one can read [70] and [81] for more details.

In order to solve this problem, we go about defining a recursive function β similar to the α function from the previous section. The only difference is that while the α function calculated the probability of an observation sequence up until time t, the β function is the probability of observing the $\mathbf{O}$ sequence for the sequence after point t. In other words:

$$\beta_t(i) = P(O_{t+1}, O_{t+2}, ..., O_{T-1}|x_t = q_i, \lambda) \tag{5.13}$$

If $t = T-1$ then it can be shown that $\beta_{T-1}(i) = 1$ for all $i, 0 \leq i < (N-1)$. From that result we can calculate β for other values of t by using the following recursive formula:

$$\beta_t(i) = \sum_{j=0}^{N-1} a_{ij} b_{j,O_{t+1}} \beta_{t+1}(j). \tag{5.14}$$

Our problem statement written mathematically is that we wish to calculate:

$$\gamma_t(i) = P(x_o = q_i|\mathbf{O}, \lambda), \tag{5.15}$$

so that at each position t in the state sequence, we can find the maximum value for γ, over all values of i.

We know that α returns the probability of sequence $\mathbf{O}$ up until time t, and β returns

the probability of sequence $\mathbf{O}$ after time t so:

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P(\mathbf{O}|\lambda)}. \tag{5.16}$$

Now that we've shown how to solve the first two HMM problems, we come to the third (and more useful in our field) problem.

Solving the Third HMM Problem

The previous two problems are only of limited application to our field, but their solutions provide us with formulae that we'll be needing to solve this third problem. As with the previous two sections, one can read [70] and [81] for more details.

Our problem statement is this: we have fixed sized matrices (the values T, M and N are fixed), but the contents of the matrices need to be optimised. After calculation, the values of $\mathbf{A}$, $\mathbf{B}$ and π need to maintain their stochastic property (their rows must always sum to 1). Typically the matrices are filled with random information, then their contents are replaced with better and better values.

We can assign values to the initial state distribution π by taking the γ functions from problem 2 at time 0:

$$\pi_i = \gamma_0(i).$$

To calculate the $\mathbf{A}$ and $\mathbf{B}$ values requires a little more work. We define the ϵ function similar to the γ function used to solve the previous problem. ϵ takes three parameters t (a subscript) and i and j:

$$\epsilon_t(i,j) = P(x_t = q_i, x_{t+1} = q_j|\mathbf{O}, \lambda). \tag{5.17}$$

This ϵ function is the probability of transiting from state q_i to q_j between time t and $t+1$. By stochastic theory we can write ϵ as:

$$\epsilon_t(i,j) = \frac{\alpha_t(i)[a_{ij}b_{j,\mathbf{O}_{t+1}}]\beta_{t+1}(j)}{P(\mathbf{O}|\lambda)}. \tag{5.18}$$

We will use the following formulae to calculate $\mathbf{A}$ and $\mathbf{B}$:

$$\sum_{t=0}^{T-2} \gamma_t(i) = \text{expected number of transitions from } q_i, \tag{5.19}$$

and

$$\sum_{t=0}^{T-2} \epsilon_t(i,j) = \text{expected number of transitions from } q_i \text{ to } q_j. \tag{5.20}$$

To get the values for the a_{ij} we take the expected number of transitions from state q_i to state q_j (equation 5.20), and divide that by the number of transitions from q_i to *any* state (equation 5.19):

$$a_{ij} = \sum_{t=0}^{T-2} \epsilon_t(i,j) / \sum_{t=0}^{T-2} \gamma_t(i). \tag{5.21}$$

A similar formula gives the b values. We know that $b_{j,k}$ is the probability of observing k while in state q_j. This can be given by the ratio:

$$\begin{aligned} b_{j,k} &= \frac{\text{expected number of times in state } q_j \text{ and observing symbol } k}{\text{expected number of times in state } q_j} \\ &= \sum_{t=0, \mathbf{O}_t = k}^{T-2} \gamma_t(j) / \sum_{t=0}^{T-2} \gamma_t(j), \end{aligned} \tag{5.22}$$

where the $\mathbf{O}_t = k$ in the first summation affects the α and β formulae.

We will now show how to apply the calculations for this third problem to perform speech recognition.

5.3.5 Applying HMMs to Speech Recognition

In this section we will describe how to use HMMs to recognise phonemes given audio features. We will also introduce the identification of words given phonemes, but the complexity of this topic makes the details beyond the scope of this document. These are not the only ways to apply HMMs to speech recognition calculations. See [70] for descriptions of these other applications of HMMs to speech recognition.

As mentioned in the last section, the way we use HMMs to recognise speech is that we solve the third HMM problem - we approximate the model λ itself.

It is useful to note that while solving this problem, we assumed a state sequence $\mathbf{X}$, but it cancelled itself out of the formulae by the end. (α, β and γ are independent of (X)). The only thing we need in order to be able to perform the calculations are our observations ($\mathbf{O}$).

Recognising Phonemes using HMMs

Recognising phonemes using HMMs is a two part process: (a) training and (b) identification.

In order to recognise phonemes using HMMs, we train one HMM per phoneme to recognise. We pass to each of the HMMs the feature vectors extracted from utterances of that model's specific phoneme. We then update the $\mathbf{A}$, $\mathbf{B}$ and π matrices for that model.

The actual observation sequence is the set of features extracted from the audio signal itself. Unfortunately the HMMs use observations from a discrete (finite) set of possible observations. In order to satisfy this condition, we round (or truncate) all observation values to the nearest integer and clip the observations between specific minimum and maximum values. In our case the features used are the first couple of cepstral coefficients.

Once we have trained the HMMs, we can identify phonemes by passing our observed audio features into the HMMs and checking which HMM returns the highest probability of the observation sequence.

Up until this stage, we have a sequence of identified phonemes. Sometimes at a specific position within a word, the phoneme is not identified precisely, and may require further classification (this problem was also present when using neural networks). As a result we may wish to try identify the word in which the phonemes existed, and then know more certainly which phoneme was uttered.

5.3.6 Recognising Words using HMMs

In order to recognise individual words given our sub-word units (phonemes), the words need to be represented in terms of these sub-units. Once this has been done, one needs to match the given sequence of phonemes to the words to find the most likely match. With a small vocabulary it is possible to use an HMM for each word, and find the most likely HMM, but this approach is impractical with large vocabularies.

In his paper about using HMMs for speech recognition [70], Lawrence Rabiner describes a triple-layer system of HMMs that can be used to identify words. He further mentions other work regarding the stack algorithm and other techniques. Unfortunately many of the methods proposed are of extreme computational complexity, and for purposes of our work regarding animated faces, the additional accuracy given by identify-

ing words is too computationally costly.

Due to the vast size of the field, the interested reader is refered to additional resources in the next section.

5.4 Other Classifi cation Techniques

Neural networks and Hidden Markov Models are by no means the only classification techniques used for speech recognition. Campbell [48] states that Dynamic Time Warping (DTW) and Vector Quantisation (VQ) are also popular. Furthermore the simplistic neural network structure described in this document can also be extended. Time delay neural networks and binary-pair partitioned neural networks have been used to great effect in phoneme recognition applications as well.

5.4.1 Additional References

The methods described above are by no means the only way in which speech can be classified. Marcus Fillipson [37] and George White [94] have written useful tutorials about speech analysis and recognition. Stephen Cook [23] has also set up a useful website that also provides lots of additional information for people wishing to write speech processing applications. Conversational technologies [21] is another useful website providing many useful links of up-to-date research in speech recognition. Salomon et al. [76] show how to use support vector machines for phonetic classification.

5.5 Summary

In this chapter we looked at two methods for classifying sub-word units such as phonemes (or even entire words given a small vocabulary): *Neural Networks* and *Hidden Markov Models*. We showed how to train these networks and apply them to speech recognition problems.

Before this part of the document is concluded, a brief mention of some of the challenges one should expect to encounter when writing speech recognition applications is given.

Chapter 6

Speech Recognition Challenges

Now that we have a basic understanding of how sound signals are recorded, processed and classified it is appropriate that we study the difficulties regarding the implementation of these systems.

The following issues will be dicussed in this chapter:

- Acquiring training data,
- Dealing with incorrectly labelled phonemes, and
- Dealing with training data during the transition from one phoneme to the next in a given word (phoneme borders).

6.1 Acquiring Training Data

In our experience, one of the worst problems with acquiring speech training data is to label the phonemes of that data correctly. To describe the phonemes from which the word is made up is trivial, but noting the location of the phonemes within the utterance is far more difficult. The difficulty arises from the fact that there is a blending between phonemes making up a word - their positions are not distinct.

Perhaps the most typical method of determining phoneme positions is to listen to the utterance and locate the phoneme positions manually. This process can be time-consuming, especially if one considers the number of utterances required to accurately train a typical classification system. A better technique would be to leverage existing work and use automated or semi-automated tools such as Microsoft Liset to identify phonetic positions within words.

If one knows beforehand which word is being uttered, then the process becomes easier.

The set of potential phonemes that can be at a given position within the word is limited. This allows one to build in greater tolerances. If one, for example, has an utterance of the word "monitor" (/m/ /ao/ /n/ /ih/ /t/ /er/), then we know that the word will begin with the phoneme '/m/'. All that is then needed is to determine when the '/ao/' phoneme becomes more dominant than the '/m/'. By repeating the process, we can identify the positions of each phoneme in the utterance.

The next section will explain what happens when the actual phonemes of the word are incorrently labelled.

6.2 Dealing with Incorrectly Labelled Phonemes

Some dictionaries (such as the Carnegie Mellon University (CMU) dictionary) describe phonetic pronounciation of words. Unfortunately the pronounciation of words is culture dependant. One example is the utterance of the word "tomato". Some cultures pronounce the word as "/t/ /ow/ /m/ /ah/ /t/ /ow/" and others pronounce it as "/t/ /ow/ /m/ /ey/ /t/ /ow/".

In order to solve matters like these, Colin and Lua [19] describe a relabelling process based on preliminary training results. They mention that discrepancies arise from one of two causes: incorrect identification (with correct labelling) or correct identification (with incorrect labelling). Unless one can prove the former case to be true, the relabelling process must be manual.

In the next section we describe how to deal with phoneme borders within words.

6.3 Phoneme Borders

As mentioned in section 3.7, speech flows from one phoneme to the next instead of occuring as distinct jumps. For this reason, it is difficult to accurately determine the exact position of transition from one phoneme to the next. Even if this border is precisely known, it remains a question of how to label a single frame, containing data from both sides of the border (one frame with parts of two phonemes in it).

Methods to handle this would be:

- label the entire frame as the dominant phoneme, or
- use a weighting factor describing the ratios between the phonemes.

Making this decision requires an understanding of the application itself.

6.4 Summary

This chapter described some of the challenges one should expect to face when training speech-recognition systems. Three of the more common challenges were discussed: acquiring training data, dealing with incorrectly labelled phonemes and dealing with phoneme borders. Possible solutions were given for each problem.

Part III

Facial Rasterisation

Chapter 7

Introduction

In the previous major part of this document, we discussed ways of analysing and classifying audio inputs. In this part we will discuss how to use those inputs to effectively animate facial models.

The need for visual cues to supplement audio signals improves a listener's comprehension of the message by an average of 17% (from 55% accuracy using hearing only, to 72% accuracy using both hearing and vision) [18] and [65]. When building an Avatar (virtual image of a talking head [74]), one needs to consider the quality of the output. This (subjective) measurement is separated into two classifications - videorealism and photorealism.

According to Poggio and Ezzat [33], photorealism is a measurement of how correct the facial features (such as face-shape and skin texture) look in each individual frame of video output. To achieve photorealism, one needs to create biologically accurate models of the face, driven by physics models that produce correct results or utilise imagery of the actual speaker. Texturing methods also play a role in determining photorealism.

Videorealism refers to the accuracy of the motion of the facial structures. To determine videorealism one would ask questions like: "Is the speed of interpolation of, for example, lip positions (between two consecutive phonemes of speech) the same as the speed of the actual speaker's lips?".

The chapters discussing these issues are:

- Modelling Techniques (chapter 8)- This chapter describes techniques for modelling faces. It includes descriptions about how the models can be manipulated, based on various inputs, especially audio streams. The end of the chapter provides a contrasting viewpoint to anatomical correctness.

- Rendering Techniques (chapter 9) - This chapter focuses on way to make the rendered models more visually appealing or effective. Shading techniques, texturing techniques and related concepts are described. The Cg language, which can be used to implement many of these techniques will be also described.

- Video-Realism (chapter 10) - This chapter will focus on ways to accurately animate our models, so that their movements appear to be at the correct speed. Model manipulation will be revised, but speech synchronisation will be the major focus of this chapter.

Chapter 8

Facial Modeling

In this chapter, we will focus on building a model of our animation. This is a mathematical model of that which we wish to display. Such a model must be controlable so it can be positioned in the ways we desire. Furthermore, based on such a model we should easily be able to produce (render) an appropriate image of the face.

This chapter is divided into the following sections:

- Different ways of modeling faces - this section describes the mathematical models used to manipulate the overall shape (geometry) of the facial models. It therefore will describe techniques currently used to *move* facial features.
- Automated Model Creation - in this section we describe ways to automatically create models of faces.
- An alternative to anatomical correctness - this section describes certain techniques that rely on aspects other than realism of facial features to enhance communication effectiveness.
- Additional References.
- Summary.

8.1 Different Ways of Modeling Faces

In this chapter, we present techniques used to display an animated facial image (Avatar). In this first section of the chapter we review the various techniques that have been (and are still) used to accomplish the animations themselves. This is purely a discussion about the geometrical structure (usually 3D) of the face - improving the quality of the displayed image is left for later sections of the chapter.

Over time, techniques for facial modeling have been placed into 3 categories [3], [49] and [50]:

1. Performance-based models (section 8.1.1)
2. Parameterised models (section 8.1.2)
3. Muscle based models (section 8.1.3)

These will be discussed in the following few sections.

8.1.1 Performance-Based Models

Performance-Based Models Explained

According to Irene Albrecht et al. [3], the performance-based models rely on a set of pre-rendered images, or pre-created video segments of the object (in our case a human head). These are played in a sequence best representing the object at key-points in time. Some techniques ([69] and [95]) involve slightly altering the original images to match the audio track, while others [7] pre-create short video clips of the object (in our case mouth animations) for each *triphone*. (A triphone is a set of three consecutive phonemes). The appropriate video sequence is then displayed, based on the given inputs (current triphone from the audio track).

Performance-based models, while usually being extremely photo-realistic, can prove difficult or slow to *animate* accurately. The following section describes some of the criteria that are needed to make the technique possible.

Criteria For Using Performance-Based Models

The success of this model usually relies on having a huge database of original images or video clips to display. This large number of video clips would give the model the flexibility required to exhibit accurate animation. This would unfortunately be at the cost of:

- Storage size of the database - still images and video usually take up quite a lot of hard-drive space, even when compressed.
- Time taken to load these different images / videos (trivial in non-realtime scenarios). Loading images from the hard-drive (and often decompressing on the fly) can be a time-costly exercise.
- The inflexibility of having to pre-create every image that will be displayed. When using performance-based modelling, one is unable to display images that have

not been pre-created. This limits the set of positions into which an object can be deformed.

One extension to performance-based modeling relies on being able to combine two or more original images into a more appropriate one. Examples of this technique are described in [31], [34] and [32]. This would provide a near-limitless set of images to display, but is usually too computationally expensive to be used in real-time environments.

It is (at the time of writing this document) impossible to store an unlimited database of images or videos. This poses a problem because there are a huge number of phonetic sequences that could be uttered at any given time. As a result, performance-based models can potentially suffer from video discontinuity, where consecutive frames are displayed that are so different from one another that a viewer perceives a distinct ‘glitch’ in the video [7], [33]. The next section will explain solutions undertaken to overcome this problem.

Coping With Video-Sequence Discontinuities

We mentioned in the previous section that performance-based models suffer from potential discontinuities. When videos are simply strung together based on the current *triphone*, then this issue becomes noticeable. The issue of discontinuous video is worsened by catering for any possible head movements (lateral movements or rotational movements). To solve this problem, morphing techniques are typically employed. This is done by morphing one viseme (visual imagery corresponding to an audio phoneme) into the next, so that the transition is smooth (typically, doing this is equivalent to simply using the parameterised models that are described later). Bregler et al. [7] also describe the technique of slowing down (or speeding up) frames in the video sequence between two phonemes, so as to synchronise the video with the audio.

Despite the difficulties of using performance-based models they are still used today because they also offer several advantages. The next two sections describe these advantages, then contrasts them with some other issues that have not been sufficiently overcome yet.

Advantages of Performance-Based Models

The greatest advantage of using performance-based models is photorealism. By using actual footage of a real person, subtle nuances of emotion (a sparkle in the eye, or dimples of a smile) that are difficult to create using other techniques are reproduced. Also, the actual skin textures (except when using morphing) are completely accurate.

Another advantage to using pre-created images is that skilled artists can often draw, paint or air-brush original images on paper which can then be scanned in.

Disadvantages of Performance-Based Models

One of the great disadvantages of performance-based models is the time taken to render the models. Because of the fact that still images are relatively large (depending on compression, size and resolution) it can take quite a long time to load still images from a disk fast enough to be displayed in real-time. Caching these images in RAM normally doesn't work either because RAM is typically too small to hold large numbers of images. Morphing techniques are especially slow - they usually perform a per-pixel calculation between two or more images. This requires many computations (depending on the size of the images) and could overwork the primary CPU, unless performed by dedicated hardware.

Another disadvantage of using the performance based technique, is the difficulty of trying to synthesize facial rotations. The images shown have usually been pre-recorded. As soon as there are head rotations, shadows and lighting changes occur, not to mention the fact that previously obscured features for example ears could become visible. To calculate these changes, given only 2D original images is extremely difficult to achieve (especially in real-time), and often requires the use of a 3D model.

In summary we can see that performance-based models can result in photo-realistic images, but video-realism poses a problem. Computational complexity poses problems in real-time environments, and limitations of movement may reduce the effectiveness of the video. In the next section, we describe a technique that overcomes many of these problems - *parameterised models*.

8.1.2 Parameterised Models

In the previous section we described the first of the three techniques for modeling a face - *performance-based modeling* - and explained some of the disadvantages of using that technique. In this section we present a different model, that has become one of the most widely used techniques to animate 3D objects. We will explain how this technique works and show that it can overcome the problems related to performance-based modeling.

Parameterised Models Explained

This technique relies on a model of the face (or any other object) being manipulated by certain parameters (skeletal movements, region locations, vertex node locations, texture parameters ,etc.) [3], [18], [66][ch6]. The model typically stores the co-ordinates of the vertices of several polygons that make up the geometry of the object (and edges between these vertices). Such a model is called a mesh. It is the vertices of the mesh that are adjusted using the given parameters. The vertices may be in two or three dimensional space.

The art of using parameterised models is an old one, and has been successfully implemented by many groups of people each with subtle differences in technique that overcame certain obstacles. Some of the more popular techniques that are used will be explained in the sections that follow.

Vertex Blending

Arguably, the most popular implementation of parameterised models is to use skeletal bones to deform the vertices of a mesh. This technique is known by several names - *skinning, vertex blending, enveloping and skeleton-subspace deformation* [63, ch 3.4]. The bone itself is purely mathematical - it is not actually part of the image that is rendered. The artist models the vertices of the skin mesh and the bone structure in their relaxed positions. The vertices of the skin are then paired up with bones of the skeleton together with a blend-weight. As a bone moves or rotates, the vertices that are influenced by that bone are displaced by an amount related to the influence (blend-weight) that bone has over those vertices.

Part of this technique allows a single vertex to be influenced by many different bones. (For example vertices of an elbow's skin would be influenced by both *upper-arm* and *fore-arm* bones). This gives the entire mesh the property of elasticity. The formula for this vertex-bone transformations is a simple summation:

$$\mathbf{u}(t) = \sum_{i=o}^{n-1} w_i \mathbf{B}_i(t) \mathbf{M}_i^{-1} \mathbf{P}, \tag{8.1}$$

where

$$\sum_{i=0}^{n-1} w_i = 1, w_i \geq 0,$$

and $\mathbf{P}$ is the original vertex in world co-ordinates, $\mathbf{u}(t)$ is the transformed vertex at time t, n bones influence vertex $\mathbf{P}$, and matrix $\mathbf{M}_i$ transforms bone i to world co-ordinates. $\mathbf{B}_i(t)$ is the transformation matrix for bone i at time t. [63, ch 3.4]

This technique for manipulating meshes is particularly effective in real-time applications, due to the speed at which it executes. This speed is further enhanced by many types of modern day graphics acceleration hardware. This leaves the primary CPU of the computer free to do other work. Matrix mathematics additionally helps when bones are arranged in a hierarchy and movement of one bone affects the bones below it in the hierarchy.

Blend Shapes

Another example of vertex blending is that of *blend shapes*. A single mesh for each facial expression is modeled. The vertices of two or more meshes are then blended to create new facial expressions [49]. The difference between vertex blending and blend shapes is that with blend shapes there is no skeleton involved, but there are several source meshes with which to work.

One example of this technique is a dolphin swimming. Meshes are created with the tail up, down and horizontal. From these three meshes a fairly accurate animation of a swimming dolphin can be created.

This technique is most useful when the source meshes are very accurate. Automated techniques (laser scanners or video analysers) are used to create these meshes, which then fairly accurately represent the objects to be displayed. Blend shapes are not used as frequently as traditional vertex blending because of the flexibility that using skeletal structures provides.

In summary, the above two techniques use the parameters to directly transform the vertices of the mesh. The next technique described accomplishes this in a more indirect manner.

Free Form Deformations

In earlier sections we showed how parameters could directly affect the vertices of a model. Free-form deformations adjust the space in which an object resides in some way. The vertices within this space are then adjusted accordingly. This technique is particularly effective when dealing with soft tissued objects like blobs of gel. Yoshizawa et al. [96] describe how to move a skeleton to which nodes of the skin are attached. The mesh deformation occurs using a force caused by the skeleton on the skin itself. This force then deforms the skin to fit the new shape of the skeleton. While this sounds similar to vertex blending it is fundamentally different because the skeleton is used to modify

space, not the vertices.

In essence, most of the other parameterised models are similar to either vertex blending or free-form deformations. These techniques are however pretty useless to us unless we know how we can use the parameters to achieve the desired effects. This will be briefly described before we proceed with the next facial modeling technique.

Driving the Parameters

Once the model's mesh has been created and the parameters that influence that mesh are set up, the challenge lies in controlling the parameters to create realistic or artistic results.

When synthesizing speech imagery, the parameters of the model would normally be driven by the audio track. The phonemes identified would be mapped to some set of parameter values. As the speech progresses from one phoneme to the next, the parameters would be interpolated in such a way that the motion appears realistic (see chapter 10 for more details). There are several different techniques used to identify how the input should drive the parameters, and from there, how the parameters should drive the model itself. These are split into two categories: (a) manual and (b) automatic.

- The manual techniques relies on studying how the different phonemes *should* appear on the model, then manually adjusting the parameters so that the desired output is produced. These values are then stored, and used at run-time.

- The automatic process is seldom easy. One needs to provide some training input for the automated process. People have been successful using a computer to analyse a video (with audio) of a person speaking. The audio is analysed to identify what the person is saying. The video is studied to try to identify the parameter values that (when applied to a model) most closely produce the video imagery. In practise the face in the video has coloured dots on it to help identify how it is being deformed as the speech occurs. [13], [95]

Other parameters such as appropriate head motion and random blinking could be driven by timers or keyboard (or touch-screens, joysticks, mice etc). This would help to create natural looking motion and could contribute to the performance being more believable.

There is another facial synthesis model that while being very strongly related to parameter based models, is sufficiently different to warrant separate discussion: muscle based modeling. This is discussed next.

8.1.3 Muscle Based Models

Muscle Based Models Explained

In the past, many 'tricks' were necessary to create accurate looking 3D models. This was due (in part) to inferior computing power. It is said that computing power doubles roughly every 18 to 24 months. This increase in power over the years has allowed (a) more complex models of 3D characters to be created and (b) the use of more true-to-life models instead of using these 'tricks' to approximate them.

3D modellers have, in recent years, started creating models that are more and more anatomically correct. Instead of the skeletal bone-influence (see section 8.1.2) methods of vertex-blending, modellers are creating their models using layers of muscle, fatty tissue and other layers that exist in real people [3], [29] and [50]. While not a new technique, it is only in recent years that computing power has allowed the muscle modeling technique to be used in real-time rendering applications.

The principle behind muscle based modeling is fairly simple: typically one end of the muscle is fixed to a skeleton (non-visible), and the other to the skin mesh itself. The skeleton can move or rotate around a pivot point, causing the muscles and skin vertices to move or rotate as well. (So far, this exactly like skinning discussed in the previous section). The muscles can contract (and expand) which push or pull the skin vertices away from, or nearer to the skeleton. (It will be shown later how the effects of muscles acting on skin vertices are smoothed out over an area of the mesh).

Another type of muscle is only connected to vertices of the skin mesh. These are typically sphincter-type muscles which are elliptical in shape, and contract towards a central point in 3D space - pulling the skin vertices with it. A good example of a sphincter muscle is the lips of the mouth. (In typical models of mouths the sphincter muscle is combined with other muscles to produce many degrees of motion of the lips).

Some of the earliest work in muscle based modeling was done by P. Eckman et al. [28]. They attempted to develop a comprehensive system which could produce all possible facial movements. This system was the *Facial Action Coding System*, which grouped sets of muscle movements into indivisible *action units*. Many of the formulae used in muscle based modeling today were described in that paper.

The remainder of this section is organised as follows:

- Muscle placement - this describes how muscles and skeletons are placed, so that they can produce the most effective results.

- The maths behind muscle contraction - this section describes the formulae used to deform skin vertices based on muscle movements.
- Skin bulging - this describes the effect of muscles bulging when they contract. In other words, the conservation of muscle volume.
- Using the muscles - this section describes how one uses the muscles to create the desired effects.

Skeleton and Muscle Placement

According to Fatih Erol and Ugur Güdübay [29], there are nine pairs of muscles placed symmetrically through the human face. These are *zygomatic major, angular depressors, labi nasi, inner labi nasi, frontalis outer, frontalis major, frontalis inner, lateral corigator and secondary frontalis*.

Kolja et al. [50] further add to this model by describing the use of sphincter muscles (*orbicularis oris*). They also use a semi-automatic way of allocating a skeletal mesh: the mesh of the skin is slightly shrunken to create a skeleton structure inside the skin that forms a stationary platform for the fixed ends of the muscles. Optionally the mesh is also expanded to create invisible muscles outside the face boundaries that pull the skin outwards as well. This semi-automated method has the advantage of being able to quickly calculate an appropriate skeletal structure for any given mesh.

Yoshizawa et al. [96] describe using Voronoi vertices as an alternative method to approximate an appropriate skeleton of a model.

In summary, the skeleton structure is placed inside of the skin-mesh. Muscles are then connection to the skeleton and the skin, and can contract and relax, thereby pushing and pulling the vertices of the skin appropriately. Should the muscles be connected to only one vertex of the skin mesh, then the deformations to the skin would cause unsightly spikes in the skin as opposed to smooth bulges. As a result multiple vertices are connected to a single muscle, and the muscle effect is smoothed out over the vertices. This smoothing will be explained in the next section.

8.1.4 Manipulating Skin Vertices Based On Muscle Movements

We have shown that rotations and other movements to a bone of a skeleton must are also be applied to all objects connected to that bone. Previously these objects were only skin vertices, but the principle applies to muscle directions and starting locations as well. In

this section we show how to manipulate skin vertices based on contractions (and elongations) of muscles, in such a way that the results are visually 'smooth'. Firstly we will describe the behaviour for linear muscles, then for sphincter muscles.

Mathematically speaking, skin vertices are 'connected' to a muscle, in such a way that as the muscle contracts towards the skeleton, the vertices are also moved towards the skeleton. This is a fairly trivial observation, except that it needs to be determined *which* vertices are connected to *muscles*. If each vertex has its own muscle then the skin effects could be made to be smooth by allowing subtly different contractions for adjacent muscles. This is seldom done in practise. A far better method is to have several vertices near a muscle being affected by that muscle, but reducing intensity of the muscle effects (on the vertices), depending on the distance between the vertex and the muscle. Figure 8.1 shows how skin vertices arranged on a flat plane are moved by varying amounts depending on their proximity to the muscle itself.

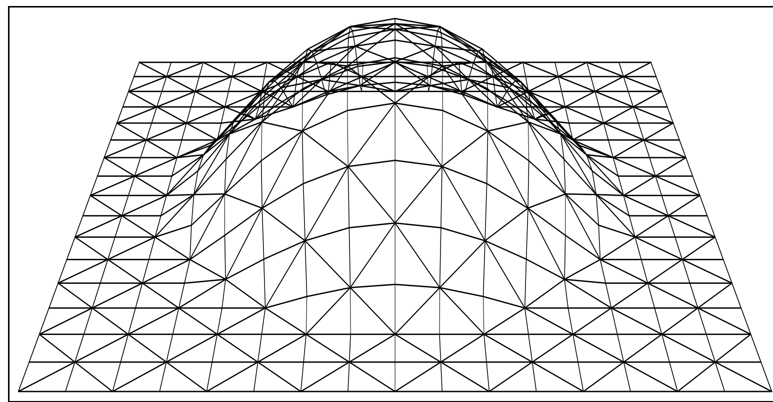


Figure 8.1: The effect of a muscle on vertices - [93]

This technique is described in Keith Waters' [93] paper, and the formulae for this section can also be found in that paper, together with additional explanations. Some of the diagrams from that paper have been included in this one to aid in our explanations.

For the following explanations of the workings of muscle models, one can refer to figure 8.2). A muscle stretches between a start point $\mathbf{V}_1$ (which is constant relative to the skeleton), and target point $\mathbf{V}_2$ (which is indicative of the direction of the muscle). Between these two points are two arcs specifying the effectiveness of the muscle along its length. The radii of these arcs are named R_s (the distance of highest effect) and R_f (the distance after which the muscle no longer effects the vertices). R_s/R_f is the contraction amount. There is also an operational angle from the muscle Ω over which the effectiveness decays laterally. Furthermore a muscle has a spring constant k which determines its strength.

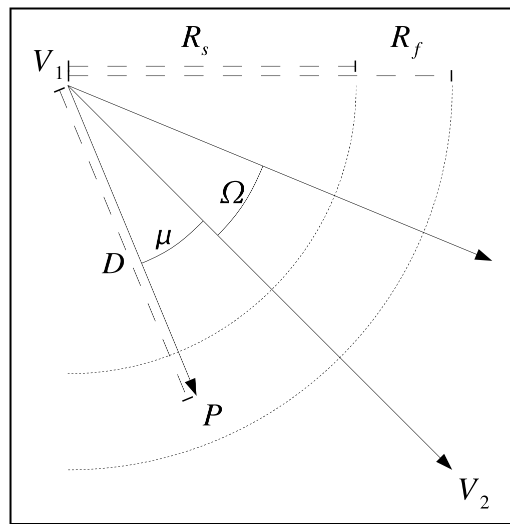


Figure 8.2: Muscle vector parameters

To calculate the effect a muscle movement has on a given point $\mathbf{P}$ we do the following:

- We need to displace $\mathbf{P}$ along the vector $\mathbf{V}_1\,\mathbf{P}$ by some appropriate distance
- So we calculate D, the distance from $\mathbf{P}$ to $\mathbf{V}_1$ and μ the angle between $\mathbf{V}_1\,\mathbf{P}$ (the vertex vector) and $\mathbf{V}_1\,\mathbf{V}_2$ (the muscle vector).
- Using these values we calculate the lateral angular displacement factor A:

$$A = \begin{cases} cos(\mu/\pi \times \pi/2) \times (1 - \mu/\Omega), & \mu \leq \Omega \\ 0, & \mu > \Omega \end{cases}$$

- We also calculate the radial displacement factor R:

$$R = \begin{cases} cos((1 - D/R_s) \times \pi/2), & 0 < D < R_s \\ cos((D - R_s)/(R_f - R_s) \times \pi/2), & R_s \leq D < R_f \\ 0, & \text{otherwise.} \end{cases}$$

- Once we have these values we can calculate a new position P':

$$P' \propto \mathbf{f}(A \times R \times P \times E),$$

  where f is an appropriate function that can model skin elasticity E (the logarithm is sometimes used).

So in summary we can specify a skeleton, then connect muscles to that skeleton with various operational parameters, and place those inside a skin mesh. As we move the

skeleton, the muscles and skin vertices move accordingly. As the muscles contract towards the skeleton, nearby vertices are also pulled towards the skeleton.

We mentioned that not all muscles are connected to a fixed skeleton, and that some pull in towards themselves: sphincter muscles. The mathematics behind sphincter muscles works in very much the same way as linear muscles except that the angular displacement factor A is ignored as all points equidistant from the centre of the muscle are transformed equally. Some sphincter muscles are not circular, but elliptical in nature. The formula can be adjusted to work with horizontal and vertical dimensions of the ellipse.

This concludes our mathematical explanation for skin manipulations using muscles. Before we explain how to create and use muscle models, we will briefly explain one last aspect of muscle modelling that can add to the realism of the models: skin bulging.

Skin Bulging

In reality true skin is not perfectly elastic but is visco-elastic, [29, 50] and should therefore bulge and flatten as the muscles under it bulge. To accurately produce this effect layers of muscle, fat and other skin tissues that are inserted under the skin have volumes that are preserved as they are manipulated [50].

Muscle fibres tend to bulge in the centre, and remain flat towards the end points, while sphincter muscles bulge evenly. To simulate how skin slides smoothly over fatty tissue, 'muscles' of low stiffness are used.

While skin bulging can add to the effectiveness of the model, there are many aspects to be considered, such as intersection testing. The interested reader might read the papers by Fatih Erol and Ugur Güdübay [29] and Kolja Kähler and Jörg Haber [50] mentioned earlier for more details.

So far we have explained the formulae used to manipulate models based on muscles and skeletons, but what remains to be shown is how to go about actually creating these models with skeletal and muscular layers. Techniques to do this are shown next.

Using the Muscles

One way of manipulating the muscles is to have an artist with knowledge of the workings of muscles manually manipulate the muscles to create the desired effect for each key phoneme. This usually takes quite a long time and can prove to be expensive.

Tony Ezzat [30] describes a more automated approach where video clips can be compared to 3D models of faces. These images are 'reverse engineered' to determine how the muscles in the face could be manipulated to produce the same effects. Once these images are linked to the phonemes extracted from the audio stream (the image is then called a viseme - *vis*ual representation of a phon*eme*), phoneme strings can be extracted from a new audio source and be used to synthesize new video.

This concludes the first major part of this chapter - modelling techniques. A contrasting viewpoint will be illustrated in section 8.3 that shows that anatomical correctness is not the only way to improve communication quality of an avatar (artificially created face).

8.2 Automated Model Creation

In the previous section we described ways to manipulate models, but one of the major problems with models is the actual creation of the models. Many different projects have had success in automatically generating models or portions of models.

One of the ways in which mesh models are created is through the use of laser scanners. Using this technique, a physical model is created. The scanner then shines a laser beam at the model to determine its contours. At each point at which the laser scans the model a vertex is created, and the vertices joined to created a mesh.

Liu et al. [56] have developed software whereby a mesh model of a human face can be acquired from a video sequence. One identifies key facial features on two of the images, and then the software generates an animateable mesh. The software also extracts appropriate texture information.

8.3 An Alternative To Anatomical Correctness

As can be seen by reading the papers cited in the prrevious sections, there has been a lot of work in recent years to build models that are as anatomically correct as possible. Often the creation of these models take a lot of time, both from the artist, and at rendering time (due to complexity of the models, and the computational complexity of the rendering processing). This section illustrates a different objective to the quest for anatomical correctness.

Chris Crawford describes in his article about anatomical correctness [24] that there are

certain areas of a face that are important in conveying information. These are typically the eyes, eyebrows and mouth. The rest of the face is relatively unimportant in the process of conveying information. He says that a lot of the work being done by anatomists does not aid in making the artwork more effective. He gives an example by comparing the movie ‘Shrek’ to the movie ‘Final Fantasy’. He describes that in Shrek, a large part of artistic effort was spent on the emotion-conveying parts of the face, while less work is done ensuring absolute anatomical correctness. In the movie Final Fantasy, extreme care is taken to make the characters anatomically correct, down to individual strands of hair. He shows that people thought Shrek was “... a superb film on any level”, while the opinion of Final Fantasy was that “... its dreamscape images almost make up for its cardboard characters...”.

Chris Crawford explains this phenomenon by quoting from Scott McCloud’s book, Understanding Comics - “When we abstract an image through cartooning, we’re not so much eliminating details as we are focusing on specific details. By stripping down an image to its essential ‘meaning’ an artist can amplify that meaning in a way that realistic art can’t.” This opinion illustrates that the pursuit of anatomical correctness is a lack of focus on the important details at the ultimate cost of effectiveness of the artwork.

8.4 Additional References

Over time, modelling techniques have continued to improve, which has resulted in a wealth of additional reading on the topic. For more information about skinning, one can read [79]. For more information about muscle modeling see [99].

Advanced Animation using DirectX [2] provides details about many other modeling techniques including (inter alia.) ragdoll physics.

8.5 Summary

In this chapter we described techniques for the creation and animation of models. We discussed three techniques that focus on improving anatomical correctness: performance-based models, parameterised models and muscle based models. We showed that, of the three, muscle models produce the most realistic results. Work that focuses on aspects other than anatomical correctness for effectiveness was also described.

In the next chapter we will show how we can improve the quality of the visual impact of our rendered models.

Chapter 9

Techniques For Improving Rasterisation

In the previous chapter, we discussed techniques that could be used to create models of an animated face. In this chapter, we continue by describing photo-realism, and other concepts related to improving the quality of our rendered output.

There are several techniques that have been developed by programmers and artists from a wide range of disciplines (including huge input from computer-game programmers), that can be used to improve rendering speed and quality. This chapter describes these techniques.

The layout of this chapter is as follows:

- Shading Techniques - This discusses the three typical shading techniques: flat, Gouraud and Phong shading. It also describes PN triangles, which achieve geometrically, what the shading techniques achieve through lighting effects.
- Advanced Texturing - This discusses additional uses for textures (other than mere images that are displayed). It includes bump mapping, light mapping and similar techniques.
- Anti Aliasing - This is a technique that reduces blockiness of lines on pixellated displays.
- Cg - This is a graphics language which can be used to program advanced functionality
- Additional References
- Summary

9.1 Shading Techniques

Unless measures are taken, renderings appear blocky. This is due to the fact that models are approximations of the real object - several flat polygons are used to represent curved surfaces. In these cases, it would be undesirable for these polygons to be seen individually. For this reason shading techniques are employed to disguise these polygons. There are three commonly used shading techniques - each with their own advantages and disadvantages:

- Flat Shading
- Gouraud Shading
- Phong Shading

9.1.1 Flat Shading

Flat shading is the simplest shading technique. The normal vector for each triangle in the mesh is calculated, using the position (and order) of the vertices of the triangle. That normal vector is used to calculate the lighting for the entire triangle (see figure 9.1).

While flat shading is by far the fastest and easiest to implement, it causes each triangle making up a mesh to be highly visible (see figure 9.2). For this reason flat shading tends not to be used in practise with models that are supposed to be smooth in appearance. Two rendering techniques which *do* attempt to disguise polygons are described next.

9.1.2 Gouraud Shading

One of the most popular shading techniques for real-time rendering is Gouraud shading. With this technique meshes containing relatively few triangles can be rendered in such a way that the edges between the triangles are smoothly blended - thus disguising the fact that the model is actually made of very few polygons (see figure 9.3).

Gouraud shading calculates the lighting based on the normals of the vertices of the triangles and then interpolates those lighting values over the rest of the triangle (see figure 9.1). This shading technique is really fast (as shading techniques go), and is often implemented in hardware. It's speed makes it a favourite for real-time rendering. [63, ch 4.3]

Gouraud shading, while better than flat shading does still lack the ability to produce highlights from spotlighting effects in the middle of a single triangle. Phong shading, described next, can overcome this problem.

9.1.3 Phong Shading

As mentioned previously, Gouraud shading, which interpolates lighting values over vertices of a triangle, may miss certain highlights and spotlight effects that could occur in the middle of the triangle.

Phong shading interpolates the normal vector for each pixel of the triangle (see figure 9.1). These normal vectors are used to *calculate* the lighting effect at each pixel (as opposed to interpolating the lighting effect as with Gouraud shading). This technique has the advantage of producing highlights and other effects that would only appear when lighting is recalculated per pixel as opposed to merely being calculated once per vertex and interpolated for each pixel (see figure 9.4).

While Phong shading looks really good, the recalculation of the lighting values for each pixel is computationally expensive. This technique is therefore seldom used for real-time rendering of complex scenes.

In summary, the above two shading techniques (flat shading being excluded) cause the illusion of smooth geometry through effective lighting. The next technique helps create a similar illusion, except not through the use of lighting.

Figure 9.1: Normal calculations for shading techniques

Figure 9.2: An example of flat shading

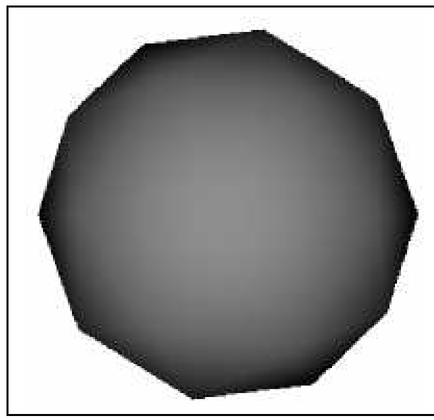


Figure 9.3: An example of Gouraud shading

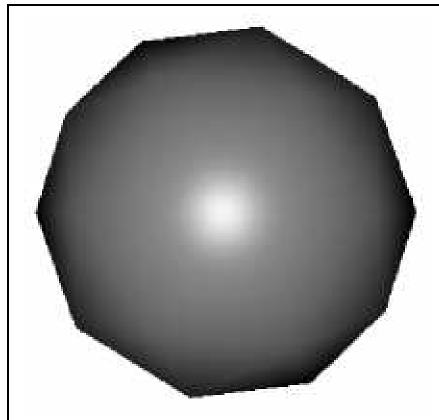


Figure 9.4: An example of Phong shading

9.1.4 PN Triangles

In previous sections we described techniques that use lighting effects to create the illusion that objects' surfaces are smooth. This is very effective except when one looks at the silhouette of the shape (the outer borders). The blockiness is then clearly shown - something more than mere lighting tricks are needed to improve this. This can be seen in the shading-related figures 9.2 to 9.4.

One possibility would be to increase the number of polygons in the model, thus making the surfaces rounder. Unfortunately this would introduce many other issues: the model would take more disk space to store, more time to load, and when animating the model there would be more calculations required as there would be more vertices to transform.

Vlachos et al. [90] describes the use of PN Triangles. With PN triangles, a large triangle is subdivided into smaller triangles, where the smaller triangles bulge outwards (Figure 9.5). The inner triangles' bulges are calculated by interpolating the normals of the outer

vertices as shown in figure 9.6. The number of sub-triangles determines how effectively this technique improves smoothness.

The advantage to using PN triangles is that the models are stored, loaded and animated in their low-polygon-count forms. The few polygons making up the model are sent to the graphics hardware, where the polygon count is then increased. This causes the increased workload to be isolated to the graphics hardware alone, while retaining the improved appearance of increased polygon count.

So in summary, PN triangles cause models to appear more smooth when seen from the side (see figure 9.7) - a feat that mere shading techniques do not achieve.

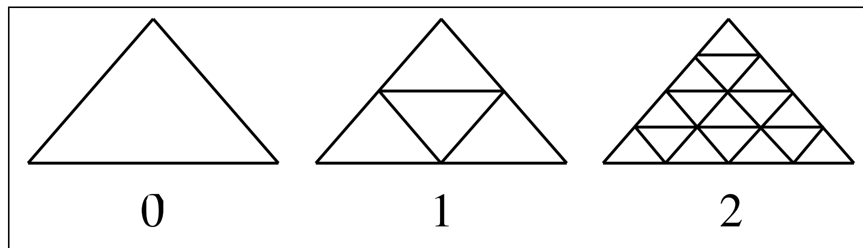


Figure 9.5: Subdividing a PN triangle [90]

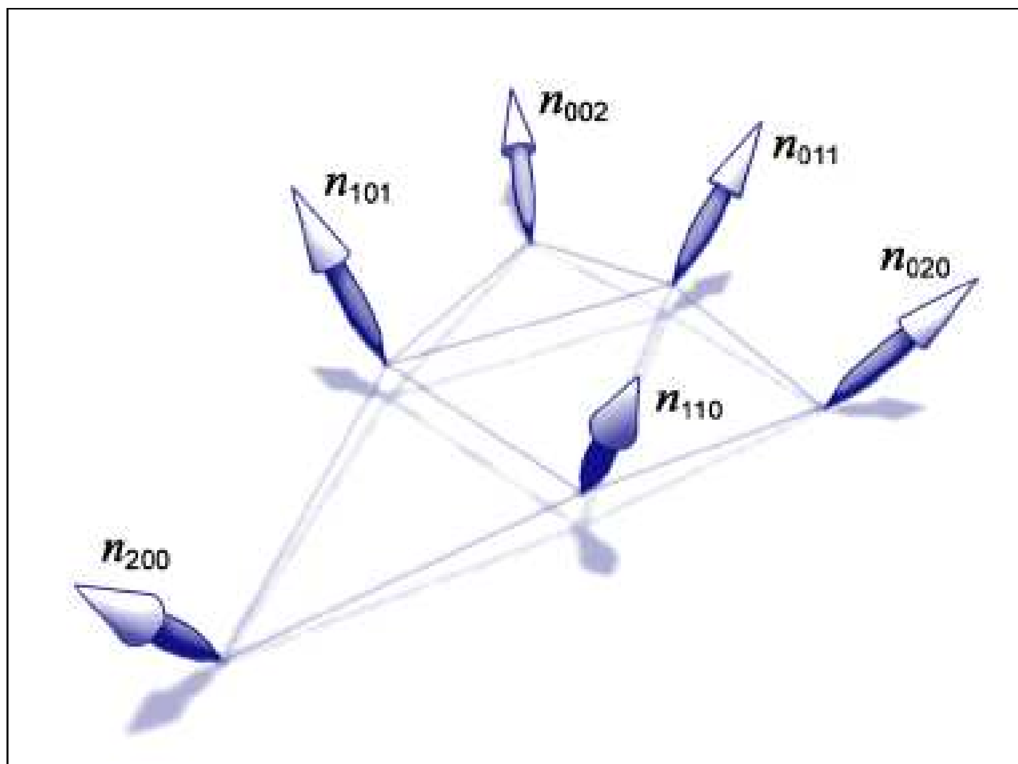


Figure 9.6: Curving the PN triangles [90]

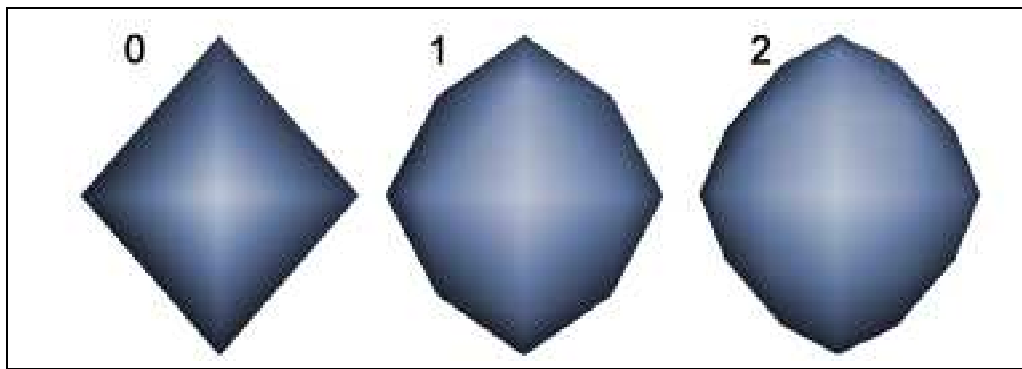


Figure 9.7: Output from PN triangles [90]

In the next section, we show which of these techniques is most suitable for real-time facial animation.

9.1.5 Shading Techniques for Facial Animation

The irregularities produced when using flat shading would be highly evident after rendering a face - the triangles making up the mesh would be highly evident (for an example see figure 11.1). This eliminates flat shading as the best choice.

Due to the fact that the majority of faces rendered are not shiny, (the specular component of the skin material would be either very low or non-existent) the advantages of using Phong shading would not be evident after rendering. The time taken to do Phong shading in a real-time render would however be evident, especially with high polygon counts. This eliminates Phong shading as the best choice, making Gouraud shading the best of the three for real-time facial animation.

PN triangles are typically used in conjunction with either Gouraud or Phong shading. When PN triangles are employed during facial rendering, the contours of the face would appear smoother.

Besides shading techniques, one can use advanced texturing techniques to improve the quality of rendered images [86].

9.2 Advanced Texturing

In the previous section, we explained how different shading techniques can maintain the illusion of smoothness of geometry by using special lighting effects. We continue now by describing other techniques for enhancing the appearance of models, by applying

textures in innovative ways. Most of the techniques described have existed for many years and have been especially useful in computer games due to the speed with which they can be executed.

9.2.1 Bump Mapping

Bump mapping refers to the technique of using a texture to create effects that appear to be altitude differences. Instead of increasing computational expense using additional polygons to represent these tiny features, an extra texture is used that alters the lighting normal for each pixel on a surface [63, ch 5.7]. This texture is not a visible image in the normal sense of textures. The ‘pixels’ of the texture are actually values from which normal vectors can be calculated.

When rendered, this additional texture alters the apparent angle that the light source strikes the surface (because the bump-map’s normal together with the polygon’s normal is used). This can create the effect of bumps and wrinkles on the otherwise flat surface.

When seen from the side, the effects are not apparent (just like Gouraud and Phong shading), as they are also a lighting illusion, not actual geometry. In the field of rendering human faces, the bump map could indent tiny pores in the skin, or even wrinkles that would add too much work (in time-critical scenarios) when represented as part of the geometry.

There are several algorithms to achieve bump-mapping, three of which will be described here:

- Height Maps
- Dot 3 Bump Maps
- Displacement Maps

9.2.2 Height Maps

One way of representing a bump map is to use a monochrome texture where each pixel of the texture represents the height of the texture above the polygon itself. The normal vector for each point is obtained by first calculating the slope between the point and its neighbours, then taking the normal of that slope. This is called a *height map*.

This technique can be computationally expensive as the calculation for each pixel relies on studying the pixel’s neighbours.

9.2.3 Dot 3 Bump Maps

A technique that works faster than height mapping is *dot 3 bump mapping*, where each pixel of the bump map texture is assigned a normal vector. (The 8 bit R, G, B values of the texture map to the range of values between -1 and 1 for x, y and z respectively). The lighting calculation merely involves computing the dot-product of the lighting vector with the normal vector read from the bump-map. Most modern day 3D hardware can do this calculation, so it can be implemented to run very quickly.

Like height maps, dot 3 bump maps do not modify the actual geometry of an object, so when seen from the side a polygon would still appear flat. The method described next overcomes this problem.

9.2.4 Displacement Maps

A technique described by Cook [22] is displacement mapping, where the bump map is used to modify the actual geometry of the object, not just the lighting illusion. Support for this technique has been added to Microsoft's DirectX 9.

Technically what happens is that the surface is tessellated into smaller triangles, then the height map is sampled to determine a displacement for the vertices of the new triangles. This is similar to PN triangles described earlier. As the process of tessellating the triangles can result in added computational expense, the degree of tessellation decreases with increased distance from the camera (this is similar to mip-mapping, where increasingly low-resolution textures are used, the further a mesh is from the viewer).

Volino and Thalmann [91] describe the technique of using displacement mapping to create wrinkles on the skin texture. Their technique bases the wrinkle effects on a skeletal mesh, to animate wrinkles easily and quickly.

9.2.5 Calculating Bump Maps

To calculate the texture required for dot 3 bump mapping, one usually begins with a height map, and then calculate normal values for that texture based on slopes, then store the results.

This process can also be automated. One could use either high-precision laser scanners [98] to scan the actual face (normally too expensive or inaccurate), or a feature detection program that analyses the skin-texture bitmap itself for spots of darkness and brightness, and computes normals appropriately [75].

Rushmeier et al. [75] describe a system for capturing bump-map information from a series of images. Their system makes using of lighting variation to predict surface geometry for the creation of bump-map textures. In their system the images of the object are all captured from the same angle, but the lighting parameters vary.

9.2.6 Self Shadow

Because bump maps are an illusion of additional geometry - we may want shadows to be displayed. The tiny elevations of the bump-map cast shadows onto the mesh itself to accentuate the slight nuances of the geometry. This could greatly add to the realism of features such as skin-wrinkles on a face.

Interpolated horizon maps are the usual technique to accomplish this, but Forsyth [38] shows how to use volumetric textures to achieve hardware accelerated texturing to calculate these shadows.

9.2.7 Light Mapping

Another technique that uses textures in an innovative manner is that of light-mapping [63, ch 5.7]. A 'texture' of lighting values is pre-computed and multiplied with the texture of the actual surface. This achieves Phong-like shading without the computational expense of true Phong shading. The downside of this technique is that the light source in the environment must remain stationary (the texture is pre-computed with values for that light source).

Because multiplying textures can be executed on modern-day hardware, this technique becomes computationally cheap to use, while the results are phenomenal (this technique was used in the rendering of Dr. Sid from the movie *Final Fantasy*). It works especially well to enhance specular lighting, which could be used to on a face that is slightly shiny.

9.2.8 Gloss Mapping

Related to standard light mapping is the technique of *gloss mapping* where sections of the texture are glossy, while others are less so. This glossiness is the quantity of reflected light as opposed to direction of the light (when using light mapping). This pre-computed texture map can be blended in with colour-based textures during specular lighting calculations to provide a glossy sheen. An excellent place to use this technique would be the trail of wetness on a model's cheek after a tear rolls down.

So in summary, textures can be used in innovative ways to create really brilliant effects that add to effectiveness or realism, without adding significant calculation overheads.

In the next section, anti-aliasing (which causes edges of polygons to appear smoother) will be discussed,

9.3 Anti-Aliasing

Besides shading techniques and effective texturing, one can do smoothing in another way - antialiasing. Due to the fact that images are rendered in pixels, the edge of an object that has been rendered typically is either represented in an entire pixel or not at all - there is no such thing as a partial pixel (see figure 9.8).

The technique of anti-aliasing blends the border pixels of the object with the background pixels depending on where the object's border lies relative to the centre of the pixel. (see figure 9.9).

Using this technique causes the rendering to look more smooth around the silhouette of the image - the way the edges would appear had they been photographed. It is slightly computationally expensive, but can be successfully implemented in real-time by sufficiently powerful hardware.

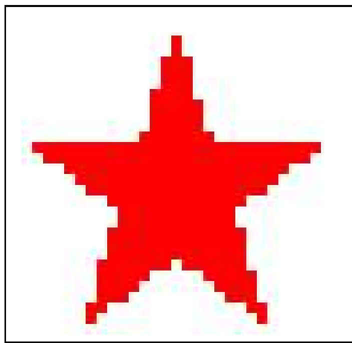


Figure 9.8: Aliasing

Many of the techniques described in this section can be implemented in hardware. In the next section a way to implement them in a platform independent manner will be described.

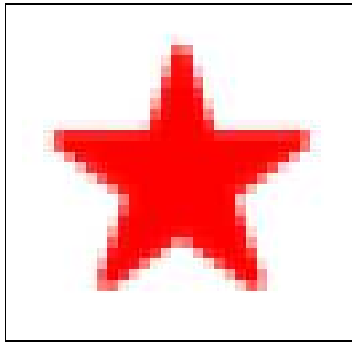


Figure 9.9: Anti-aliasing

9.4 The Cg Language

One of the newer tools of the trade is Cg - a language very similar in syntax to the C language. This language is used to write small programs that leverage the power of modern graphics card's powerful calculating abilities. The difference between normal C programs and Cg programs is that Cg programs are written to be executed on graphics hardware, not a standard CPU. The language is typically used to write one of two classes of programs: vertex shaders and pixel shaders.

Vertex shaders are used when vertices of polygons need to be shifted or interpolated by some formula. The vertex shader can use the graphics hardware to do these calculations as opposed to using the computer's CPU. This can really free up a lot of work from the computer's processor, leaving it available for other work.

A pixel shader is used to calculate the colour of a pixel based on lights in the scene, materials, textures and other inputs. The pixel shader needs to be a highly optimised piece of code, as it is executed once for each pixel drawn. By writing multiple versions of a pixel shader using Cg, one can choose one that best balances the performance of the graphics card with quality of the output (the typical time-quality tradeoff) [59].

Cg is not the only shader language that has been developed. HLSL and the OpenGL shaders can be used as well [55].

9.5 Summary

In this chapter, we showed various methods for improving photo-realism. We concluded that Gouraud shading provides the best balance between effectiveness and computational complexity. The texturing methods described could each be used add to the effectiveness, depending on the desired effects. We showed how full-screen anti-aliasing

can be used to create the appearance of smoother (sub-pixel) renderings. We concluded with a description of the Cg language which can be used to implement many of these techniques on dedicated graphics hardware.

In the next chapter, we continue with our descriptions by explaining methods of improving video-realism (the accuracy of motion of the animations).

Chapter 10

Video Realism

In previous chapters of this document, we discussed modelling and rendering techniques that can aid us in improving the quality of our output. We continue in this direction by illustrating ways to improve the quality of the *motion* of the animations.

10.1 Introduction

In the early days of computer rendering, many people were impressed by the quality of the models themselves, but complained that movement of the models appeared unnatural. Many different factors contribute to this unnatural motion, and as a result, many different research projects are underway to improve this. In this chapter we discusses some of this work, focusing on the following topics:

- Motion Capture - this describes ways to capture the motion of real human actors.
- Physics - by paying attention to the rules of physics, we can further improve our motion quality.
- Co-articulation - this focuses on co-articulation aspects of speech synchronisation efforts.
- Summary.

10.2 Motion Capture

Despite the best efforts of artists and animators, realistic model motion remains difficult. Many animation packages provide tools that aid in standard actions like walking and running, but even these tools can be time-consuming to apply correctly.

In a real-time environment, one cannot afford the time to delicately apply these tools

to the models - a faster method is necessary. Over the past years several direct motion capture devices have been used. In essence they all work in the same way: sensors are applied to an actor's body. These sensors track the movement of a specific part of that actors body in 3D space. The model that is being animated is then deformed in a way that matches the movements of the real actor. The model can therefore do nearly anything that a real actor can do.

What makes this method very versatile is that the model need not have the same characteristics as the actor. A prime example of this type of actor-controlled animation occurred in the movie 'The Lord of the Rings'. In an article about the making of the movie [40], the animation of Gollum (and other creatures) is described. Actors' motion was captured using high-resolution cameras that observed retro-reflecting spheres (placed on the actor's face, body, fingers and toes). This motion was then applied to a model with physical attributes that were different to the real actor's. Animation of the Gollum actually used several different models (character maps) for different types of motion (walking, crawling, climbing).

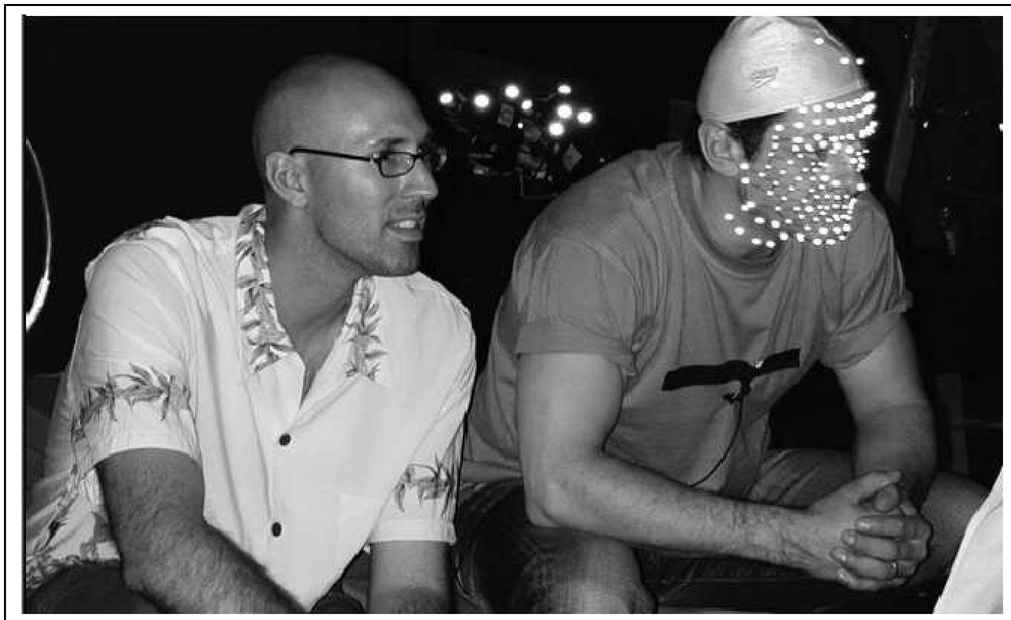


Figure 10.1: Left to right: Remington Scott and Andy Serkis (Gollum) exploring facial performance capture. Read about bringing Middle Earth creatures to life; "Sparking Life" by Remington Scott, cover story, SIGGRAPH, Nov 2003 (v37 n4) http://www.siggraph.org/publications/newsletter/ [12, 40]

This motion capture was extended to facial expressions as well with sensors being applied to key points on the actor's face. By tracking these sensors, the actor could control the facial expressions and lip movements of the model directly. This provided extreme

levels of emotional realism.

Not only are reflective spheres used in industry. Magnetic sensors can (within a relatively small area) be pinpointed in 3D space as well. When applied to an actor's body, these sensors can identify the movements of the actor's body. Magnetic positioning is not used to detect facial animations due to the physical size of the sensors. Instead markers placed on the actor's face are tracked using more than one video feed of the actor [44]. The positioning of these markers are usually in accordance with MPEG-4's facial points (FP).

In this manner facial animation can be controlled in real-time. There are cases where facial sensors are impractical, but can still be used in a once-off manner during training. By capturing video sequences of actors speaking, one can use tools such as neural networks to identify the relationship between spoken sounds and facial parameters. At run-time the networks can be used to predict the facial parameters given audio input.

For additional information about motion capture techniques, see [53] and [62].

For times when motion capture cannot be used at run-time, one can use mathematical formulae to help control these movements. In the next section we will describe techniques that control the speeds of these minute adjustments.

10.3 Physics

In the previous section, we described how to use motion capture sensors to study the movements of a real actor. Unfortunately, due to the necessary sparseness of the sensors, many movements are done through interpolation techniques instead of being measured. Usually the model itself takes care of these minute details, but with one problem. Different muscles contract and relax at different speeds. In this section we discuss how one can use spring physics to control the speed of muscle adjustments.

In his original paper about muscle modeling [93], Keith Waters mentions skin elasticity and muscle spring constants. These two related concepts are used to return a model to its relaxed position at an acceptable rate.

The mathematical principle concerning springs and elasticity is Hooke's Law. Hooke's Law [1] states that if a force (F) is applied to an elastic spring (of length L, cross-section A and elasticity E) then its extension (ΔL) is proportional to its tensile stress

(σ) :

$$\Delta L = 1/E \times F \times L/A = 1/E \times \sigma \tag{10.1}$$

By assigning appropriate parameters to the muscles in our models, the motion follows true spring physics. This allows relaxation of facial features to occur at the correct rate.

In order to make use of this technique, manipulations to the muscles should be described in terms of forces being applied, not as absolute lengths. In this manner several different forces acting on a muscle could be combined, and a net force calculated.

10.4 Co-articulation

In section 3.7 we introduced the concept of co-articulation. In this section, we describe one of the methods used to predict co-articulation effects, and show how to apply it to improve video realism.

It was mentioned that co-articulation is the interpolation of vocal tract structures (mouth, lips etc.) between adjacent phoneme utterances. All co-articulation models predict this interpolation, but the methods for doing so differ.

Attempts at identifying a mathematical formula to predict this mouth anticipation have met with varied degrees of success. One reason for this is cultural – different cultures, languages and accents accentuate different co-articulation effects [18]. Despite this problem, some of the attempts at modeling co-articulation have shown success.

In the model by Pelachaud et al. [67], phonemes are assigned a deformability ranking. This is a measure of the extent to which surrounding phonemes can influence that phoneme (in the sense of vocal tract shapes). A blending interpolation then occurs between adjacent sets of phonemes. For each phoneme the ideal lip shape is computed, and then in later passes, muscle movement is calculated based on contraction and relaxation times of the muscles (together with the deformability of the current phoneme).

Cohen and Massaro [18] describe in their model, that at any given time, phonemes have a given dominance over the various articulators of the vocal tract. A set of overlapping functions determines this dominance value which leads to a blending of the articulatory commands. This includes functions for dominance of the lips, jaw, tongue tip, tongue body, tongue root, velum and larynx. These functions differ in duration, magnitude and time offset. They are usually at their peak at the moment of utterance of the phoneme, and fall off to zero influence over a short period of time.

Cohen and Massaro describe a formula for predicting the dominance D of a facial control parameter p for speech segment (phoneme) s. The distance in time from the centre of the speech segment is τ and $\theta_{\leftarrow sp}$ and $\theta_{\rightarrow sp}$ represents the rate parameter.

$$D_{sp} = \begin{cases} \alpha_{sp} e^{-\theta_{\leftarrow sp} |\tau|^c}, & \text{if } \tau \geq 0 \\ \alpha_{sp} e^{-\theta_{\rightarrow sp} |\tau|^c}, & \text{if } \tau < 0. \end{cases} \tag{10.2}$$

The c parameter comes from the model proposed by Löfqvist in 1990 as mentioned in the paper by Cohen and Massaro.

In most common situations for calculating co-articulation effects, knowledge is needed of speech units (a) before, and (b) after the current one. The fact that we need to know (b) implies that it is not possible to accurately perform co-articulation calculations in true real-time. It is only possible in continuous-time, coupled with a slight delay in the output.

One can observe (by the fact that the function is a negative exponential) that the value of the function decreases with increasing $|\tau|$. These functions are asymptotic and as such we can assume some cut-off value for $|\tau|$ for each function. By so doing we can effectively eliminate the influence of a phoneme before and after some time period. This enables us to work with sounds spanning a finite time duration. The minimum value for τ is then used to calculate the time, after which buffered sound may be discarded. The maximum value for τ is used to calculate the delay in output needed to perform look-ahead co-articulation calculations.

10.5 Summary

In this chapter we defined video realism and looked at some of the techniques that can be applied to enhance it. These included using motion capture devices both at run-time and at training time, using physics models especially spring physics to allow for correct muscle movement speeds and co-articulation which more accurately predicts vocal tract positions.

Part IV

Implementation

Chapter 11

Implementation

Up until this point, we have described the different techniques that can, and have been used to perform the various aspects of interactive speech-driven facial animation. We now mention which ones we chose to implement our system, and describe the implications of our choices.

The techniques employed are described under the following categories:

- Digital Signal Processing,
- Modelling,
- Rasterisation, and
- Driving the model parameters.

11.1 Digital Signal Processing

In part II of this document we described that speech processing is a multi-step process consisting of: digitisation (sampling) of the signal, signal processing and then classification.

11.1.1 Sampling

In order to decide which sampling settings would be most appropriate, we needed to find a balance between (a) sufficient data for accurate classification and (b) not allowing sufficient time for the signal to change significantly.

The Advanced Linux Sound Architecture (ALSA) was the sound API used. We chose to use 22050 (16 bit) samples per second recorded using a single channel (mono).

During the digitisation process these samples were stored in a buffer that could store up to 1024 samples. Every time the buffer had 512 or more samples we removed and processed the first 512 of them. As will be explained in section 11.1.2, the number of samples we had to process at a time needed to be an integer power of 2. Had this number been too small, our classification system would have had insufficient samples with which to work and would not have classified the data accurately. Had the number been too large, the delay between classifications would have caused the model's movements to be too abrupt. Using 512 samples provides sufficient data to classify while still enabling the classification system to execute roughly 50 times per second – allowing for smooth animation while still leaving the features of the signal relatively constant through the frame.

Once we decided on the sampling rates, we needed to choose an appropriate technique to extract features from the digital signal.

11.1.2 Signal Processing

In chapter 4 we described essentially two major techniques to extract features from sound signals: Fourier Transforms and Wavelet Transforms. After testing, we found that despite the fact that the DWT provide better localisation in time, the broadness of the specrum of Fourier Transforms provides for more accurate classifications than the DWT. See chapter 13 for the results of this comparison.

We mentioned (in section 4.7.3) that the standard Fourier Transform executes in $O(n^2)$ time, which is fairly slow. If the number of samples, n, over which the Fourier Transform executes can be written as $n = 2^k, k \in \mathbb{N}$ then the algorithm can be optimised using the Cooley-Tukey Fast Fourier Transform (FFT) which executes in $O(n \log(n))$ time. For this reason we chose to use frames containing 512 samples.

We also mentioned that in order to execute the Fourier Transform, the input signal is expected to be periodic. Human speech is a non-periodic function (even within a single frame), so a windowing function needs to be applied to each frame to make the signal periodic (in that frame). We chose to use the Blackman windowing function because of the fact that all derivatives of the window at its borders are zero (as opposed to only the signal itself being zero at the borders). While the Blackman windowing function requires a relatively large number of calculations to execute, the results for a fixed size window can be pre-calculated once and applied many times. This implies that the computational cost of using the Blackman window is no higher than any of the other windowing techniques.

Towards the end of chapter 4, we described the fact that a Fourier Transform (or Wavelet Transform) alone does not transform the data sufficiently for classification. We described (in section 4.9.1) how cepstral coefficients provide a better classification vector. Thus after applying the Blackman windowing function and the Fourier Transform, we calculate the Mel cepstral coefficients.

Next we needed to choose the best method to classify phonemes based on these feature sets.

11.1.3 Classification

As mentioned in chapter 5, due to the age of the field and the difficulty of the research (see chapter 6), many different classification techniques can be employed to perform phoneme identification with different degrees of success. If one looks through the vast resources available regarding phoneme classification, one is able to see a trend that two of the more commonly used phoneme classification techniques are Hidden Markov Models and Neural Networks.

We decided to use neural networks for classification of the cepstral coefficients instead of Hidden Markov Models. The reason for our choice is that Hidden Markov Models are used to identify a state sequence (in our case a whole word) given the observed input vector [97]. Our problem requires real-time recognition, not continuous-time recognition. In other words Hidden Markov Models would require the utterance of complete words before identification occurs, while neural networks can effectively classify phonetic information without the presence of the rest of the word.

Based on the fact that our training data was English language utterances, and that our phoneme descriptions come from the CMU dictionary, we chose to identify 39 different phonemes and an extra one for silence (see table 3.1). We used individual networks for each of the 40 phonemes. The structure of the networks is as follows:

- the input to each network was the vector of cepstral coefficients,
- each network had one output,
- the activation function used was log-sigmoid,
- the training algorithm was resilient propagation (section 5.2.1) and
- the number of nodes in the neural networks is a parameter to the application

11.1.4 Training

The manner in which training data is passed to a neural network has a large influence on the success of the training. One of the problems with a neural network is that training information passed to the network only affects the current training epoch. Further training can hinder the network's ability to classify accurately. For example, consider a network that classifies phonetic information. Assume that the network is given feature sets from the phoneme '/zh/' and informed that the expected output is 1, and features for the phoneme '/ae/' and informed that a 0 is expected. Assume that after several training epochs the network has the ability to distinguish between the two phonemes. Then the network is given features from the phoneme '/s/' and informed that a 0 is expected. This last step is repeated many times. After several epochs of only the '/s/' as input, the network may lose the ability to identify the '/zh/' phoneme.

In order to prevent this problem, all available training data was pre-processed into sets of input vectors and desired outputs. Training data was then chosen at random from the entire set of available training data, instead of training the network using a single utterance at a time.

Due to the fact that the gradient vector is summed over each training set in the epoch, it is important to balance the number of positive (expected output of 1) training inputs with the number of negative (expected output of 0) training inputs. This will ensure that the magnitude of the gradient vector is biased neither towards the positive nor the negative training sets.

In order to accomplish the above-mentioned balance, counters (of the number of positive and negative trainings in the epoch) were used. If too many positive (or negative training vectors) had already been received, then additional positive (or negative) training vectors were ignored. An epoch was declared finished when the correct number of positive and negative training sets had contributed to the gradient vector.

Next we will describe the modelling techniques we chose to employ.

11.2 Modelling

As described in chapter 8, there are many different modelling techniques, each with their own advantages and disadvantages. The model to be chosen, therefore depends on the requirements of the application.

Our application requirement are:

- our model must be 3D,
- the manipulations to the model must be achievable in real-time,
- the manipulations should be implementable on dedicated 3D hardware and
- the quality of the model should be flexible, depending on available computing power.

We chose to use a mesh to represent our facial model due to the flexibility provided by doing vertex manipulation. Vertices of a mesh are easily transformed using matrices – a technique that is widely supported on common graphics hardware.

To control vertices we used two sets of transformations. We first applied skeletal manipulations to manipulate the jaw-bone and tongue. To achieve the more subtle manipulations such as cheeks and lips we implemented Waters' muscle model [93].

In order to implement this model, we stored the vertices of the mesh at rest. These vertices were transformed by the skeletal animations into a second buffer of vertices. A third buffer was then required to store the muscle-manipulated vertices. This latter transformation could not be done in place due to the fact that several muscles might affect a single vertex. The mesh of the vertices stored in this third buffer were then rendered.

The number of vertices in the mesh itself as well as relationships between the mesh, bones and muscles can be adjusted per model that is displayed.

Having decided on a model, the rasterisation parameters needed to be chosen.

11.3 Rasterisation

In order to rasterise our model, we chose to use the OpenGL API. Due to the speed constraints of our system (and the fact that a human face is not particularly shiny) we decided that Gouraud shading was adequate. In order to achieve Gouraud shading (see section 9.1, specifically figure 9.1), normals for vertices had to be calculated according to the following algorithm:

```
for each vertex v in mesh m:
    set v.normal = (0, 0, 0)
```

```
for each face f in mesh m:
        vector edge1 = f.vertex [1] - f.vertex [0]
        vector edge2 = f.vertex [2] - f.vertex [0]
        f.normal = crossproduct (edge1, edge2)
        for each vertex v in face f:
                v.normal = v.normal + f.normal

for each vertex v in mesh m:
        v.normal = v.normal / v.normal.length
```

This calculation leaves us with a normal value for a given vertex that is an average of the normals of the faces of which that vertex is a part. The difference in image quality this provides can be seen in figure 11.1. (The image on the left was rendered using non-averaged normals, while the one on the right did have averaged normals).

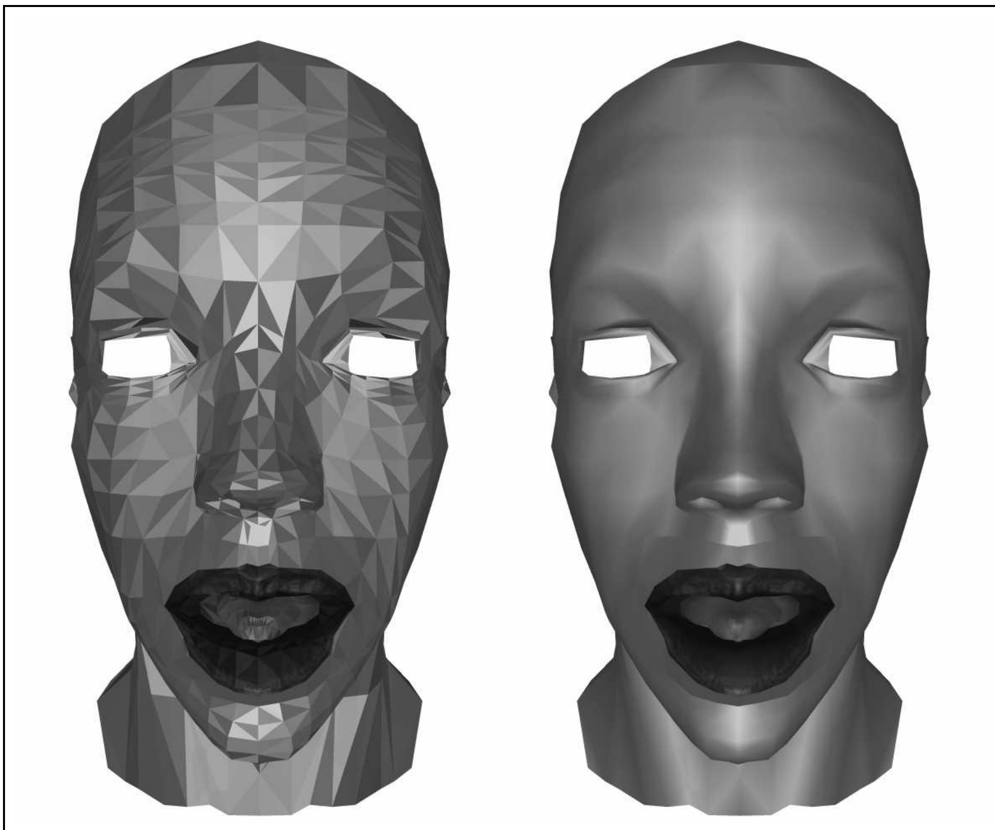


Figure 11.1: Implication of normals on shading

The final part of this chapter describes the relationship between the phonetic data that was acquired and the model parameters themselves.

11.4 Driving the Model Parameters

In order to make a decision about how to drive the model parameters using phonetic information, one must first understand the parameters themselves.

As mentioned in section 11.2 our model is manipulated by means of skeletal manipulations and muscle contractions. Skeletal manipulations are a rotation (in a specific direction) about the origin of a given bone of the skeleton. This rotation in 3D space is represented by two orthogonal angles. Muscle manipulations are a scalar value that describe the amount of contraction of the muscle. In our model the muscles can elongate as well as contract (muscles in our bodies only contract or relax – they do not elongate).

For each of the 40 phonemes, we stored the desired parameters for each bone and muscle in the face. Roughly 50 times per second, we acquired a frame of audio information from which we identified the current phoneme being spoken. We also knew what phoneme was being spoken during the previous frame. Using these two values we could, for each display cycle, interpolate the parameters linearly.

A problem arises out of the fact that our *target* phoneme is the one that has *already* been spoken, thereby causing the output to no longer be real-time, but only continuous-time (delayed by one frame of audio information).

In the next section, we provide some additional information about the implementation of the prototype.

Chapter 12

The Prototype System

In the previous two chapter, we described the techniques that we chose to use to implement our system. These results do not however give any information regarding the actual design and implementation of the prototype. This chapter describes some details of our implementation and highlights some of the problems we experienced while implementing the prototype. Also mentioned are conclusions that were drawn during practical research before and during implementation of the actual prototype.

12.1 The Overall Design

Due to the dynamic nature of the system (different models can be used), we chose to design our system as follows. We created an abstract class (DisplayManager) that provides an interface to handle any sound, keyboard, mouse and display events. This abstract class was then extended to create a model-choosing menu class, and a model-display class.

These classes were organised in a stack, with the menu at the bottom. In this manner, when one DisplayManager terminated, the system defaulted back to the previous one (the menu). In this manner all sound and graphics needed only be initialised once, but the events from these modules could be handled dynamically.

During implementation of a prototype, one often encounters problems that were not predicted at design-time. The next section describes some of these problems which we encountered.

12.2 Problems Encountered

During the implementation of our system, we encountered some problems:

- Initially our prototype was designed with the polygon normals being stored with the mesh itself. After adjusting the vertices of the mesh, the normals were inaccurate. This created very strange shading effects, such as a bulge in the cheeks of the model didn't produce a shadow under the bulge (the normals were still those of a vertical cheek surface).

  In order to solve this problem, we were forced to re-calculate the normals after any changes to the vertices.

- In order to use display-lists (an OpenGL feature that reduces function call overheads), our texture co-ordinate indices and vertex indices needed to coincide. This was a problem because a single vertex may have been on the border between two different textures, and would therefore require more than one texture co-ordinate.

  In order to solve this problem, we created a python-script for Blender (the 3D modelling tool we used), which duplicated vertices that used texture co-ordinates from two or more textures.

- During the testing phase of our classification system, our training data was pre-recorded wave files with very little noise. When we implemented the system using live recording, we found that ambient noise levels were extreme due to low quality sound-recording equipment.

  In order to overcome this problem, we used the pre-recorded sounds in a manner as if they were being recorded in real-time. This enabled us to observe the effectiveness of the system as it would be with high quality recording equipment.

12.3 Facts about the Prototype

The prototype was implemented in the C++ language. We chose the language due its execution speeds and the fact that it provided us with class inheritance (needed for the DisplayManager class). Other than the DisplayManager class, no classes nor structs use virtual methods. This causes the execution speed to increase as there is no method lookup time wasted, all method pointers can be determined at compile-time.

We made extensive use of template classes, which improved the maintainability and readability of the source code.

The combined size of the source-code (excluding standard libraries) is less than 6000

lines of code. The Python script to export our Blender models into our custom format is 315 lines of code. It took roughly six weeks to integrate the neural network with the facial model to achieve animation. The prior work converning fourier transforms, wavelets, cepstral coefficients, and neural network training took just over a year to implement. The binary compilation of the prototype is 124 Kb in size.

In the next chapter we describe the results obtained by driving our system as described in this and the previous chapters.

Chapter 13

Results

In this chapter we give descriptions about the system on which the prototype was implemented and results obtained. The source-code for the prototype can be found on the accompanying CD.

13.1 System Specification

The hardware and software settings for the execution environment were as follows:

- CPU - Intel Pentium 4 2400MHz (hyperthreaded)
- RAM - 1Gb 333MHz (dual channel)
- 3D Hardware - nVidia GeForce FX 5200 (128Mb)
- Operating System - Mandrake Linux 10.0 (Kernel 2.6.3)
- Graphics Driver - NVIDIA-Linux-x86-1.0.6111
- Sound API - Advanced Linux Sound Architecture
- Graphics API - OpenGL
- Screen Resolution - 1280 $\times$ 1024

13.2 Model Information

The model (on which the following results are based) had the following properties (unless otherwise specified):

- There were 3479 vertices,
- There were 6626 triangular faces,

- There were 4 textures - each 256 × 256 pixels in size.

The results of our implementation are divided into the following sections:

- Phoneme recognition performance and
- Mesh manipulation performance.

13.3 Phoneme Recognition Performance

The results of our phoneme recogniser are measured at two stages: training and usage. The training metric describes the various parameters of the neural networks, the type of input vectors given, epoch sizes, accuracy and convergence time. The usage metric describes the speed of phoneme identification during use. We first provide results for the DWT, then more detailed results for the Fourier Transform.

The sample training data was a collection of English utterances spoken by an adult female. In order to measure accuracy we took the ratio of: number of correctly identified phonemes / total number of test phonemes.

Due to the fact that the number of positive and negative trainings per epoch were limited (see 11.1.4 for more details about this training method), the number of epochs per neural network varied depending on the number of instances of the appropriate phoneme in the training data. The metric given, therefore, is number of training cycles executed - irrespective of whether some networks ignored the data or not. In all cases the number of tests executed is 5000.

13.3.1 The DWT

Due to the DWT's ability to provide more localised support, it was one of the choices as a transformation tool. Unfortunately as the following results illustrate, the DWT did not provide effective phoneme recognition in the prototype. (See the accompanying CD for the source code of the prototype).

Table 13.1 describes the classification accuracy of the DWT in our prototype. The Daubechies-4 Wavelet Transform was used. In the table, the numbers in brackets indicate the number of sets of training data passed to the neural network. As we can see, the DWT does not allow for accurate classification of phonetic information. In the next section we describe the classification statistics for log cepstral coefficients.

Coefficients	Accuracy (2000)	Accuracy (12000)	Time (12000)
10	16%	16%	4s
15	17%	15%	6s
50	16%	15%	44s
100	15%	17%	165s

Table 13.1: The accuracy of phoneme identification using the DWT

13.3.2 Classification results of log cepstral coefficients

As mentioned in the previous chapter, we used one neural network per phoneme. (We chose not to use Hidden Markov Models due to the fact that HMMs are used to identify most probable sequences (of phonemes in our case) - not a single isolated item). To train these networks, we extracted cepstral coefficients from the signal and passed them as inputs to the neural networks. Different numbers of coefficients resulted in different levels of accuracy and speeds of execution.

The first tests executed were designed to determine the minimum number of cepstral coefficients required as inputs before the neural networks performed sufficiently accurate classifications. Table 13.2 illustrates these findings (numbers in parentheses are the number of training inputs – not the number of epochs as might be expected). It shows that 50 coefficients provides a good balance between accuracy and speed, so we chose to use 50 coefficients for our system. The accuracy is almost 50% better than the DWT in this case. (See the accompanying CD for the source code of the prototype).

The next set of tests, was designed to choose the ratio of positive and negative trainings per epoch. The number of coefficients was fixed at 50. As can be seen from table 13.3, the ratio of positive and negative trainings per epoch has a large influence on the training rate of the neural networks. For our application the optimum ratio is 10 positive trainings to 10 negative ones per epoch.

The speed of execution of the classification module is left for the next section, which also deals with the speed of the mesh manipulations and renderings.

13.4 Mesh Manipulation Performance

In this section we will describe the performance of the various parts of the system. The metric used is frames per second. In table 13.4 we show the effect that each module has on the execution speed of the system.

Coefficients	Accuracy (2000)	Accuracy (12000)	Time (12000)
10	42%	49%	1s
15	46%	51%	1s
20	43%	44%	1s
25	53%	63%	3s
30	53%	66%	3s
40	31%	72%	5s
50	49%	74%	7s
75	39%	73%	13s
100	54%	80%	28s

Table 13.2: The effect of cepstral coefficient counts

All the above tests were done with relatively few polygons (6626 triangles) in the mesh. We will now show how polygon count affected the performance of the application. Each consecutive result was caused by smooth-subdividing the mesh from the previous result. Table 13.5 illustrates these results. We can see that polygon count has a severe effect on the performance of the system as a whole.

One area where a performance dip is expected is when the number of muscles is increased. Table 13.6, however, shows that effect of muscles on frame rates is not drastic. The reason for this is that the effect of the muscles is limited to a few vertices (approximately 60 vertices per muscle).

13.5 Conclusion

A full description of conclusions about these results can be found in the following chapter.

Positive	**Negative**	**Accuracy (12000)**
1	1	18%
1	3	36%
1	5	47%
1	10	62%
1	20	53%
3	1	25%
3	3	62%
3	5	69%
3	10	68%
3	20	69%
5	1	30%
5	3	55%
5	5	68%
5	10	71%
5	20	74%
10	1	6%
10	3	64%
10	5	65%
10	10	74%
10	20	73%
20	1	4%
20	3	51%
20	5	36%
20	10	44%
20	20	61%

Table 13.3: The effect of training data composition

Activated modules	**Frame rate**
Display	143
Display, skeleton	140
Display, skeleton, muscles	138
Display, skeleton, muscles, normals	130
Display, skeleton, muscles, normals, sound	128

Table 13.4: The speeds of the different modules

Vertices	**Faces**	**Frame Rate**
1443	2650	128
5537	10600	98
21675	42400	30
85751	169600	8

Table 13.5: The effect of polygon count

Muscles	**Frame Rate**
2	46
10	45
50	43

Table 13.6: The effect of muscles

Chapter 14

Conclusions and Contributions

This chapter takes the results of the previous chapter and uses them to show the extent to which the original problem has been solved. The chapter is broken up into three sections :

- Speech Classification - in this section we describe how the speech classification portion of the problem statement has been solved.
- Facial Modeling - in this section we show how our prototype solves the facial animation portion of the problem.
- Entire Solution - this section shows how the integration of all the modules of the prototype show that the problem stated has been completely solved.

14.1 Speech Classification

During the early stages of the prototype development, phoneme classification was attempted on Fourier Transformed data (without cepstral coefficients). In our attempts, no amount of training (with any structure of neural network) was able to classify this data. This information led us to initiate studies about wavelets and cepstral coefficients.

It was mentioned that Wavelets have been used to classify phonemes. Our attempts at classification using the DWT (see table 13.1) proved fruitless. Our opinion is that the results are poor due to the fact that the number of frequencies present is the log of the number of samples in the frame ($N_{frequencies} = \log_2(N_{samples})$), whereas after performing the FFT, data for far more frequencies is present.

The results for classification of cepstral coefficients (tables 13.2 to 13.3) show that cepstral coefficients provide for effective classification. They allow us to conclude that real-time speech processing is possible. While the accuracy could potentially be improved

with additional effort, the accuracy given is sufficient for purposes of the prototype.

Next we discuss the facial animation portion of the prototype.

14.2 Facial Modeling

During execution of our prototype, we observed that the lighting effects were incorrect. This was due to statically defined normals, which are inadequate for a flexible model. This led us to perform normal calculations for all vertices. The speed reduction was noticeable, which led us to perform the normal calculations only for necessary vertices.

During the implementation of texturing in our prototype, we realised that a single vertex may contain texture information for several textures. This would occur on the mesh at the border between two textured regions. Unfortunately due to an OpenGL optimisation, the index for texture co-ordinates for a vertex has to match the index for the vertex itself. This 1 to 1 relationship prevents a single vertex having multiple texture co-ordinates. In order to solve this problem, we enhanced our python script (which exports Blender meshes - see accompanying CD) to duplicate vertices that have multiple textures. One unsolved problem is the fact that when a vertex is duplicated for multiple textures, then the normal calculations are slightly incorrect as the calculation utilises fewer faces for that vertex than it should. To solve this problem, some manner of relating vertices during normal calculations would need to be implemented.

Overall, our results show that the facial model parameters can be driven in real-time by the classified speech signals. Tables 13.4, 13.5 show that the facial animation can occur in real-time.

In the next section, we discuss the conclusions about the integration of the two modules of the prototype.

14.3 Entire Solution

Our initial problem statement was to determine whether or not real-time human speech can be effectively used to drive a facial animation. It is insufficient to merely show that real-time speech classification and real-time facial animation are possible. The integration of the two is necessary to solve the problem.

In order to perform this step in our prototype, we manually adjusted the facial model

into an appropriate pose for each phoneme. The properties of each bone and muscle were then recorded. (This process is unfortunately a very labour intensive one). Upon identification of a given phoneme, the prototype interpolated the skeletal and muscle properties from those of the previous phoneme to those of the current one.

By integrating the phoneme classification and the facial animation modules in our prototype, we demonstrated that our problem is solved : real-time human speech can be effectively used to drive a facial animation.

In the next chapter, we describe additional uses for some of the techniques mentioned in this document.

Chapter 15

Additional Applications

Despite the fact that our software is intended to be used in the entertainment industry, much of the technology described has applications in other environments. In this chapter we will describe additional uses for:

- speech recognition techniques, and
- wavelet transforms.

Almost all of the other techniques described do have other applications, but the interested reader is referred to the additional resources section of the relevant chapter.

15.1 Speech Recognition Techniques

When we hear other people speaking, we gather much more information than just the sounds that person is saying. We understand the words and context of that person's message. We can often determine subtle undertones of emotion based on the speed and pitch of the speech. People also use speech to identify people by their voices.

By careful study of how our brains perform these functions, it becomes possible to implement them on a computer as well.

15.1.1 Speaker Identification

One of the earliest forms of biometric identification systems was speaker identification. Despite the fact that to a human, some people's voices sound very similar, a computer can detect very minute details in a person's voice signal, that a human cannot.

Speaker identification is done in one of two ways:

- speaker verification or

- speaker identification

Speaker verification is used to verify that a speaker is who they claim to be. Typically some form of identification such as a user name or smart card is provided to the system. This informs the system who is expected to be speaking. The system calculates the probability of the speaker being who they say they are (based on the recording of their voice). If the probability is above a certain threshold, then the speaker is assumed to be who they say they are.

Speaker identification is a far more difficult, and less accurate process. The system receives a recording of a person speaking. It then extracts the appropriate features from the signal for processing. By comparing those features with ones stored in a database, the closest matching profile is found, and the speaker is identified. By placing a minimum threshold on the identification criteria, non-matches can also be returned.

15.1.2 Speech Samples for Speaker Identification

Sherlock and Monro [78] mention that only voiced speech (section 3.4.1) is suitable for feature extraction for purposes of speaker recognition. Unvoiced speech is too chaotic to extract reliable features.

Claude Norton, III [17] mentions that differences between the speakers' voices are the result of different resonance characteristics of the speakers' vocal tracts. These resonance differences are attributed to length and cross-sectional area differences, as well as vocal chord characteristics.

Toutios and Margaritis [87] describe a speaker identification system that uses Mel-frequency cepstrum coefficients (MFCC) and perceptual linear prediction (PLP) features. They also mention that PLP was originally used to supress speaker-dependant features, but can also be used for speaker identification. Their system uses a neural network to perform the identification from the features.

15.1.3 Reliability of Speaker Identification

Unfortunately human speech is influenced by many different factors, some of which are not reliable. For example, when a person who has contracted a cold speaks, many of the original features of the sound can become distorted. This may prevent or seriously hamper the identification process.

Another important factor to consider when evaluating the effectiveness of speaker iden-

tification is physical changes to the vocal tract over time. People's voices change a huge amount during puberty. This is caused by an enlarging and stretching of the larynx and other parts of the vocal tract. During this phase a person's voice may drop as much as one octave in pitch. This pitch change in itself may not be sufficient to confuse a speaker recognition system, but many other unpredictable changes also occur at the same time.

The next application is also used in the biometrics field.

15.2 Wavelet Transforms

Another technology that was described in this document that has applications in other fields is the wavelet transform. Due to the speed with which the wavelet transform executes, it is a very useful tool in time-critical applications.

One of the more noteable applications of the wavelet transform is in fingerprint compression. In order to identify a person by their fingerprint, many details of the fingerprint are studied. This included not only the visible ridges and valleys of the print itself, but also tinier details such as sweat pores in the middle of a ridge. For a fingerprint recognition engine to be most effective, these tiny details must be provided as input. It is therefore critical that should the image of a person's fingerprint be digitised that the image retain these details, despite any compression.

The FBI database of fingerprints exceeds 200 million fingerprint images, which need to be compressed for storage purposes. One of the major concerns was the quality of the image after compression and the time taken to compress the images. During the digitisation process (the originals are inked impressions on paper cards), the resultant image is 8 bits/pixel, 500dpi grayscale images (figure 15.1. This means that each image is approximately 10Mb.

When the FBI researched different image compression techniques, the first one researched was the JPEG standard. JPEG is known to be very effective at compressing the size of images. The most major problem encountered was the 'tiling artifacts' that even moderate compression ratios produced (figure 15.2) [9].

Finally, the FBI made the decision to use wavelet transform/scalar quantisation (WSQ) for fingerprint image compression. The reasons are that using wavelet compression, the detail loss rates are extremely low, and the tiling artifacts are absent. Brislawn [9] writes that the standard that the FBI developed (which is entirely within the public domain) involves a 2D discrete wavelet transform, uniform scalar quantisation and Huffman en-

tropy coding.

Using this wavelet transform, the images can be compressed to an average of 0.6 bits/pixel. This results in approximately 750Kb images (figure 15.3). This saving becomes huge when there are 200 million such images.

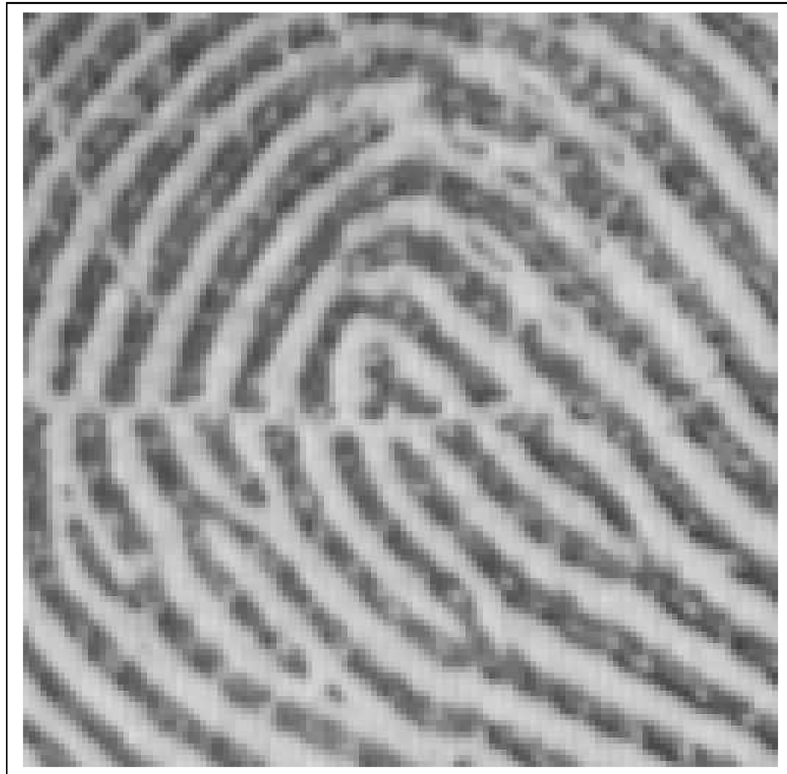


Figure 15.1: The original fingerprint [8]

The following chapter is an appendix that may be referred to for a better understanding of the working of the prototype.

Figure 15.2: The fingerprint compressed using the JPEG standard [8]

Figure 15.3: The fingerprint compressed using WSQ [8]

Appendix A

Data Structures Used

In this appendix, listings of appropriate extracts from the source code of the prototype are given. This may aid in the overall understanding of the operation of the prototype. For the complete source-code of the prototype please see the accompanying CD.

A.1 Windowing Algorithm

The following code defines the behaviour of the abstract concept of a window

```
template<typename T>
struct Window
{
  uint size;
  T * filter;

  Window (uint size);
  ~Window ();
  void transform (T * data);
};

template<typename T>
Window<T>::Window (uint size)
{
  this->size = size;
  filter = new T [size];
}

template<typename T>
Window<T>::~Window ()
```

```
{
  delete [] filter;
}

template<typename T>
void Window<T>::transform (T * data)
{
  for (uint i = 0; i < size; i++) data[i] *= filter[i];
}
```

A.2 Fast Fourier Transform

The following code extract defines the structure and behaviour of the Fast Fourier Transform used in the prototype.

```
#include <math.h>

template<typename T>
struct FFT
{
  FFT (uint logSize);
  ~FFT ();
  void calculate (T * data);

private:
  uint reverseBits (uint a, uint bits);
  uint logSize;
  uint size;
  uint * indices;
  T * real;
  T * imag;
  T * sines;
  T * cosines;
};

template<typename T>
FFT<T>::FFT (uint logSize)
{
    this->logSize = logSize;
```

```
    size = 1 << logSize;
    indices = new uint [size];
    for (int i = 0; i < size; i++)
      indices[i] = reverseBits (i, logSize);
    real = new T [size + 1];
    imag = new T [size + 1];
    sines = new T [size + 1];
    cosines = new T [size + 1];

    double angle = 0.0;
    double diffAngle = 2.0 * 3.1415926535 / size;
    for (int i = 0; i <= size; i++)
    {
      sines[i] = sin (angle);
      cosines[i] = cos (angle);
      angle += diffAngle;
    }
}

template<typename T>
FFT<T>::~FFT ()
{
  if (indices != NULL) delete [] indices; indices = NULL;
  if (real != NULL) delete [] real; real = NULL;
  if (imag != NULL) delete [] imag; imag = NULL;
  if (sines != NULL) delete [] sines; sines = NULL;
  if (cosines != NULL) delete [] cosines; cosines = NULL;
}

template<typename T>
void FFT<T>::calculate (T * data)
{
  uint halfSize = size / 2;
  uint logSize2 = logSize - 1;
  T tempReal;
  T tempImag;

  //Setup
  for (uint i = 0; i < size; i++)
```

```
{
  real [i] = data [i];
  imag [i] = 0.0;
}

uint k = 0;
for (uint logSkip = 1; logSkip <= logSize; logSkip++)
{
  while (k < size)
  {
    for (uint i = 1; i <= halfSize; i++)
    {
      uint trigIndex = indices [k >> logSize2];
      T cosVal = cosines [trigIndex];
      T sinVal = sines [trigIndex];

      uint offset = k + halfSize;
      tempReal = real [offset] * cosVal + imag [offset] * sinVal;
      tempImag = imag [offset] * cosVal - real [offset] * sinVal;

      real [offset] = real [k] - tempReal;
      imag [offset] = imag [k] - tempImag;

      real [k] += tempReal;
      imag [k] += tempImag;

      k++;
    }
    k += halfSize;
  }
  k = 0;
  logSize2--;
  halfSize /= 2;
}

k = 0;
uint r;
while (k < size)
{
```

```
    r = indices [k];
    if (r > k)
    {
      tempReal = real [k];
      tempImag = imag [k];
      real [k] = real [r];
      imag [k] = imag [r];
      real [r] = tempReal;
      imag [r] = tempImag;
    }
    k++;
  }

  for (uint i = 0; i < size / 2; i++)
  {
    tempReal = real [i];
    tempImag = imag [i];
    data [i] = (2 * sqrt (tempReal * tempReal +
      tempImag * tempImag) / size);
    data [i + size / 2] = 0;
  }
}

template<typename T>
uint FFT<T>::reverseBits (uint a, uint bits)
{
    uint ret = 0;
    for (uint i = 0; i < bits; i++)
    {
      ret += (a & 1) << (bits - i - 1);
      a >>= 1;
    }
    return ret;
}
```

A.3 Cepstral Coefficients Algorithms

```
#include <math.h>
```

```
#include "fft.h"

template<typename T>
struct LogCepstrum
{
  FFT<T> * fft;
  uint size;

  LogCepstrum (uint logSize);
  ~LogCepstrum ();

  void calculate (T * data);
  void calculateDelta (T * data);
private:
  void melLog (T * data);
};

template<typename T>
LogCepstrum<T>::LogCepstrum (uint logSize)
{
  this->size = 1 << logSize;
  fft = new FFT<T> (logSize);
}

template<typename T>
LogCepstrum<T>::~LogCepstrum ()
{
  if (fft != NULL) delete fft; fft = NULL;
}

template<typename T>
void LogCepstrum<T>::calculate (T * data)
{
  fft->calculate (data);
  melLog (data);
  fft->calculate (data);
}

template<typename T>
```

```
void LogCepstrum<T>::calculateDelta (T * data)
{
  calculate (data);
  for (uint i = size-2; i >= 1; i--)
  {
    data [i] = data [i] - data [i - 1];
  }
  data [size-1] = 0;
}

template<typename T>
void LogCepstrum<T>::melLog (T * data)
{
  for (uint i = 0; i < size; i++)
  {
    data [i] = 1127.010408 * log (1 + data[i] / 700.0);
  }
}
```

A.4 Neural Network Data Structure

The following code extract defines the structure of the Neural Networks.

```
#include <stdio.h>

class Trainer;
class Activation;
class Node;
class Layer;
class NeuralNetwork;

#define NN_DATA_TYPE float

#include "arrayList.h"

typedef ArrayListND<NN_DATA_TYPE> NNDATALIST;
typedef ArrayList<Node *> NODELIST;
typedef ArrayList<Layer *> LAYERLIST;

class Trainer
```

```
{
protected:
  NeuralNetwork * neuralNetwork;

public:
  Trainer ();
  Trainer (NeuralNetwork * neuralNetwork);
  ~Trainer ();

  virtual void train () = 0;
  virtual void free (void * data) = 0;
  virtual void * newData (Node * node) = 0;
};

class Activation
{
public:
  virtual NN_DATA_TYPE activation (NN_DATA_TYPE input) = 0;
  virtual void calculateOutputLayerErrorGradient (Node * node,
  NN_DATA_TYPE desiredValue) = 0;
  virtual void calculateNonOutputLayerErrorGradient
    (Node * node) = 0;
};

class Node
{
  friend class Trainer;
  friend class Activation;
  friend class Layer;
  friend class NeuralNetwork;
private:
  Layer * layer;
  uint id;
  NNDATALIST * inputs;
  NN_DATA_TYPE * output;
  NNDATALIST weights;
  NNDATALIST gradients;
  NNDATALIST epochedGradients;
```

```
  Activation * activation;

  NN_DATA_TYPE dEdOutput;
  void * trainingSpecific;

  uint weightCount ();
  uint inputCount ();
  void finalise ();
public:
  Node (uint id, Layer * layer);
  ~Node ();

  uint getID ();
  Layer * getLayer ();
  NNDATALIST * getWeights ();
  NNDATALIST * getInputs ();
  NN_DATA_TYPE * getOutput ();
  NNDATALIST * getGradients ();
  NNDATALIST * getEpochedGradients ();
  NN_DATA_TYPE * getdEdOutput ();
  void * getTrainingSpecific ();

  void randomiseWeights (NN_DATA_TYPE min, NN_DATA_TYPE max);
  void calculate ();
  void epocharise ();
  void clearEpochs ();
  void fprintf (FILE * file);
};

class Layer
{
  friend class Trainer;
  friend class Node;
  friend class NeuralNetwork;
private:
  NeuralNetwork * neuralNetwork;
  uint id;
  NODELIST nodes;
  uint nodeCount ();
```

```
  NNDATALIST * inputs;
  NNDATALIST * outputs;
  uint inputCount ();
  uint outputCount ();

  void finalise ();
public:
  Layer (uint id, NeuralNetwork * neuralNetwork);
  ~Layer ();

  uint getID ();
  NODELIST * getNodes ();

  Layer * previous ();
  Layer * next ();
  Node * addNode ();
  void randomiseWeights (NN_DATA_TYPE min, NN_DATA_TYPE max);
  void calculate ();
  void epocharise ();
  void clearEpochs ();
  void fprintf (FILE * file);
};

class NeuralNetwork
{
  friend class Trainer;
  friend class Node;
  friend class Layer;
private:
  Activation * activation;
  Trainer * trainer;

  bool finalised;
  ArrayList<NNDATALIST*> values;
  uint layerCount ();

  NNDATALIST * layerInputs (uint layer);
  NNDATALIST * layerOutputs (uint layer);
  void killNodeTrainingData ();
```

```
public:
  LAYERLIST layers;
  NNDATALIST * inputs;
  NNDATALIST * outputs;
  NNDATALIST desired;
  uint posTrainCount;
  uint negTrainCount;
  uint trainCount;

  NeuralNetwork (uint inputCount, Activation * activation);
  ~NeuralNetwork ();

  LAYERLIST * getLayers ();

  void finalise ();
  Layer * addLayer ();
  Layer * getLayer (uint id);
  void randomiseWeights (NN_DATA_TYPE min, NN_DATA_TYPE max);
  void calculate ();
  void calculateGradients ();
  void epocharise ();
  void clearEpochs ();
  void train ();
  void fprintf (FILE * file);
  void fprintfInputs (FILE * file);
  void fprintfOutputs (FILE * file);
};
```

A.5 Phoneme Identifier

The following code defines the structure of the phoneme identification modeul.

```
#include "arrayList.h"
#include "nn.h"
#include "logCepstrum.h"
#include "window.h"

struct PhonemeIdentifier
{
```

```
private:
  uint windowSize;
  uint featureVectorSize;
  uint allFeatureVectorsSize;
  uint windows;
  uint dataSize;
  uint currentWindowOffset;

  ArrayList<NeuralNetwork *> nn;
  LogCepstrum<NN_DATA_TYPE> * lc;
  NN_DATA_TYPE * data;
  Window<NN_DATA_TYPE> * w;

  void transform (short * samples);
  void loadNetworks (char * path, uint count);

public:
  PhonemeIdentifier (uint logSize, uint windows, char * path);
  ~PhonemeIdentifier ();
  uint identify (short * samples);

};
```

A.6 Geometry Structures

The following code defines the geometry structures used in the prototype.

```
template<typename T>
struct Vector2
{
  union
  {
    struct {T u, v;};
    struct {T x, y;};
  };

  Vector2 ();
  Vector2 (T u, T v);
  void copy (Vector2 v);
```

```
};

template<typename T>
struct Vector3
{
  union
  {
    T v[3];
    struct {T x, y, z;};
    struct {T r, g, b;};
  };

  Vector3 ();
  Vector3 (T x, T y, T z);
  Vector3 (T * v);
  Vector3 (Vector3<T> * v);
  Vector3 (Vector3<T> v1, Vector3<T> v2);
  bool equals (Vector3<T> v);
  void normalise ();
  T norm ();
  T normSquared ();
  void zero ();
  void set (T x, T y, T z);
  void copy (Vector3<T> v1);
  void subtract (Vector3<T> v1, Vector3<T> v2);
  void subtract (Vector3<T> v);
  void add (Vector3<T> v);
  void divide (float v);
  void multiply (float v);
  void interpolate (Vector3<T> v1, Vector3<T> v2, float factor);
  static T innerProduct (Vector3<T> v1, Vector3<T> v2);
  static T angle (Vector3<T> v1, Vector3<T> v2);
  void crossProduct (Vector3<T> v1, Vector3<T> v2);
  void fprintf (FILE * file, char * format, char * txt);

  static void transfer (Vector3<T> * source,
    Vector3<T> * dest, unsigned int count);
  static T dotProduct (Vector3<T> * v1, Vector3<T> * v2);
};
```

```
template<typename T>
struct Vector4
{
  union
  {
    T v [4];
    struct {T x, y, z, w;};
    struct {T r, g, b, a;};
  };

  Vector4 ();
  Vector4 (T x, T y, T z, T w);
  Vector4 (T * v);
  void fprintf (FILE * file, char * format, char * txt);

  static void transfer (Vector4 * source, Vector4 * dest,
    unsigned int count);
};

struct Matrix
{
  float v [16];

  Matrix ();

  void multiply (Vector3<float> * source);
  void multiply (Vector3<float> * source,
    Vector3<float> * dest);
  void multiply (Matrix * source);
  void multiply (Matrix * source, Matrix * dest);
  void zero ();
  void copy (Matrix * matrix);
  void identity ();
  void translate (float x, float y, float z);
  void translate (Vector3<float> v);
  void translateN (Vector3<float> v);
  void rotateX (float a);
  void rotateY (float a);
```

```
  void rotateZ (float a);
  void rotateAxis (Vector3<float> v1, Vector3<float> v2,
    float angle);
  void rotatePlane (Vector3<float> v1, Vector3<float> v2,
    float angle);
  void headingAttitudeBank (float h, float a, float b);
  void fprintf (FILE * file);
};

typedef Vector3<GLfloat> Vertex;
typedef Vector3<GLfloat> Normal;
typedef Vector3<uint> Face;
typedef Vector2<GLfloat> TextureCoord;
typedef Vector3<unsigned char> Color3;
typedef ArrayListND<Vertex> VertexList;
typedef ArrayListND<Normal> NormalList;
typedef ArrayListND<Face> FaceList;
typedef ArrayListND<TextureCoord> TextureCoordList;
typedef ArrayListND<bool> VertexMovedList;
typedef ArrayListND<char> VertexFaceCountList;
```

A.7 Mesh Structure

The following code defines the structure of the mesh itself.

```
struct DisplayItem
{
  Texture * texture;
  FaceList faces;
};

typedef ArrayList<DisplayItem *> DisplayItemList;

struct Mesh
{
public:
  bool initialised;

  VertexList oVertices;  //Original vertices
```

```
VertexList sVertices;  //Skeletal transformed vertices
VertexList mVertices;  //Muscle transformed vertices
ArrayListND<uint> vertexTextureIndices;
NormalList normals;
MuscleList muscles;
TextureCoordList textureCoords;
VertexMovedList vertexMoved;
VertexFaceCountList vertexFaceCount;
Skeleton skeleton;

DisplayItemList displayItems;

bool textured;

Mesh();
uint vertexCount();
uint itemCount();
uint muscleCount();
uint boneCount();
void calculateNormals();
void calculateAllNormals();
void moveBones();
void moveMuscles();
void display();
void displayNormals();
void normalise();
void normalise(Vertex ignore);
void normalise(Mesh * mesh, Vertex ignore);

int addVertex(Vertex v);
int addTextureCoord(TextureCoord tc);
Muscle * addMuscle();
DisplayItem * addDisplayItem();
void addMuscle(Muscle * muscle);
void fprintf(FILE * file);

void transfer();
void show_stats(char * description, Vertex * nonexist);
```

```
  static Mesh * cloneIndex (Mesh * base, Mesh * sub,
    Vertex defaultVertex, float minDist);
  static Mesh * cloneIndexNot (Mesh * base, Mesh * sub,
    Vertex defaultVertex, float minDist);
};
```

A.8 Skeleton Data Structures

The following code extract defines the data structures related to skeletal manipulations.

```
#define VERTEX_BONE_COUNT_TYPE uint

struct Bone;
struct Skeleton;

struct Bone
{
  uint index;
  Vertex start;
  Vertex stop;
  bool initialised;
  Matrix matrix; //Rotations in this matrix are about the
                 //origin - the API handles the rest

  Skeleton * skeleton;
  ArrayListND<uint> boneIndices;
  ArrayListND<uint> vertexIndices;
  ArrayListND<uint> muscleIndices;

  Bone ();
  Bone (Skeleton * skeleton, uint index, Vertex start,
    Vertex stop);
  ~Bone ();
  uint childBoneCount ();
  uint vertexCount ();
  uint muscleCount ();
  void init (Vertex start, Vertex stop, Skeleton * skeleton);
  void addVertexIndex (uint index);
  void addMuscleIndex (uint index);
```

```
  void addVertexIndices (VertexList * vertices);
  void addVertexIndices (VertexList * vertices, Vertex ignore);
  void addBoneIndex (uint index);
  void transform (VertexList * oVertices, VertexList * sVertices,
    Matrix * baseMatrix);
  void draw ();
};

struct Skeleton
{
  bool initialised;

  void * mesh;
  ArrayList<Bone *> bones;
  ArrayListND<VERTEX_BONE_COUNT_TYPE> vertexBoneCount;

  Skeleton ();
  ~Skeleton ();
  uint boneCount ();
  void init (void * Mesh);
  Bone * addBone (Vertex start, Vertex stop);
  void transform (VertexList * oVertices,
    VertexList * sVertices, VertexMovedList * vertexMoved);
};
```

A.9 Muscle Data Structures

The following code extract defines the data structures used to store and manipulate the muscles in the prototype.

```
#include "arrayList.h"
#include "geometry.h"
#define PI 3.141592f

struct Muscle
{
  uint index;

  bool initialised;
```

```
    void * mesh;
    Vector3<float> v1;
    Vector3<float> v2;
    Vector3<float> oV1; // original start
    Vector3<float> oV2; // original stop
    float omega;
    float rs;
    float rf;
    float contraction;

    ArrayListND<uint> indices;
    ArrayListND<float> weights;

    ArrayListND<Vertex> oConeVertices;
    ArrayListND<Vertex> coneVertices;
    ArrayListND<uint> coneIndices;

    Muscle ();
    void init (Vertex v1, Vertex v2, float omega, float rs,
        float rf, VertexList * originalVertices,
        bool useAngular, bool useRadial);
    void init (Vertex v1, Vertex v2, float omega, float rs,
        float rf);
    void calcCone ();
    void addVertex (uint index, float weight);
    ~Muscle ();
    void copy (Muscle * muscle);
    void contract (VertexList * vertices,
        VertexList * originalVertices,
        VertexMovedList * vertexMoved);
    void transform (Matrix * matrix);
    void draw ();
};

typedef ArrayList<Muscle*> MuscleList;
typedef ArrayListND<Muscle*> MuscleListND;
```

References

[1] Robert K. Adair. *Concepts in Physics*. Academic Press Inc., 1969. Library of Congress catalog card number 69-13481.

[2] Jim Adams. *Advanced Animation with DirectX*. Muska and Lipman/Premier-Trade, 2003.

[3] Irene Albrecht, Jörg Haber, and Hans-Peter Seidel. Speech synchronization for physics-based facial animation. In *WSCG*, pages 9–16, 2002.

[4] Sébastien Baehni. Neural java - neural networks tutorial with java applets. http://diwww.epfl.ch/mantra/tutorial/english/, October 2000.

[5] Bores introduction to dsp. http://www.bores.com/courses/intro/index.htm, 2004.

[6] Jeremy Bradbury. Linear predictive coding. http://www.cs.queensu.ca/home/bradbury/pdf/lpc_paper.pdf, December 2000.

[7] Christoph Bregler, Michele Covell, and Malcolm Slaney. Video rewrite: driving visual speech with audio. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 353–360. ACM Press/Addison-Wesley Publishing Co., 1997.

[8] Chris Brislawn. The fbi fingerprint compression standard. http://www.c3.lanl.gov/ brislawn/FBI/FBI.html, June 2002.

[9] Christopher Brislawn. Fingerprints go digital. *Notices American Mathematical Society*, 42(11):1278–1283, November 1995.

[10] Robert Bryll. Introduction to wavelets, September 2000.

[11] C. S. Burrus. Notes on the fft. http://www-dsp.rice.edu/res/fft/fftnote.asc, 1997.

[12] Photo by Karin Last. Exploring facial performance capture. Contact remington@gmail.com.

[13] Yong Cao, Petros Faloutsos, and Frédéric Pighin. Unsupervised learning for speech motion editing. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 225–231. Eurographics Association, 2003.

[14] Joseph Carey, editor. *Brain Facts - A Primer on the Brain and Nervous System*. The Society for Neuroscience, 4 edition, 2002.

[15] Tim Carmell, Andrew Cronk, Ed Kaiser, Richard Wesson, Johan Wouters, and Xintian Wu. Spectrogram reading. http://cslu.cse.ogi.edu/tutordemos/SpectrogramReading/⇒ spectrogram_reading.html, March 1997.

[16] M. Chetouani, B. Gas, J.L. Zarader, and C. Chavy. Neural predictive coding for speech discriminant feature extraction. In *ESANN*, pages pp275–280, April 2002.

[17] III Claude Norton. Text independent speaker verification using binary-pair partitioned neural networks, December 1995.

[18] M. Cohen and D. Massaro. Modeling coarticulation in synthetic visual speech. In N. Thalmann and D. Thalmann, editors, *Models and Techniques in Computer Animation*, pages 139–156, Tokyo, 1994. Springer.

[19] Tan Colin and Kim Teng Lua. A neural network based phoneme recognizer. http://www.comp.nus.edu.sg/ ctank/tdnn.pdf, 2005.

[20] Computation of the discrete fourier transform. http://jwc.njust.edu.cn/nj/jxwdxz/wdw/szxh/chapter9.pdf, August 2002.

[21] Conversational technologies. http://www.conversational-technologies.com/, 2005.

[22] Robert L. Cook. Shade trees. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231. ACM Press, 1984.

[23] Stephen C. Cook. Speech recognition howto. http://www.gear21.com/speech/html/index.html, April 2002.

[24] Chris Crawford. Artists against anatomists. *Computer Graphics*, 36(1):pp 8–10, February 2002.

[25] Howard B. Demuth, Mark H. Beale, and Martin T. Hagan. Neural network design. http://hagan.ecen.ceat.okstate.edu/nnd.html, 1996.

[26] N. Rex Dixon and Thomas B. Martin, editors. *Automatic Speech and Speaker Recognition*. IEEE Press, 1979.

[27] Tim Edwards. Discrete wavelet transforms: Theory and implementation. http://qss.stanford.edu/~godfrey/wavelets/wave_paper.ps, June 1991.

[28] P. Ekman and W.V. Friesen. Facial action coding system. http://www-2.cs.cmu.edu/afs/cs/project/face/www/facs.htm, 1978.

[29] Fatih Erol and Ugur Güdübay. An interactive facial animation system. In V. Skala, editor, *WSCG 2001 Conference Proceedings*, 2001.

[30] Tony F. Ezzat. Example-based analysis and synthesis for images of human faces. Master's thesis, Massachusetts Institute of Technology, February 1996.

[31] Tony F. Ezzat, Gadi Geiger, and Tomaso Poggio. Trainable videorealistic speech animation. In *Proceedings of Siggraph*, 2002.

[32] Tony F. Ezzat and Tomaso Poggio. Facial analysis and synthesis using image-based models. In *Second International Conference on Automatic Face and Gesture Recognition*, August 1996.

[33] Tony F. Ezzat and Tomaso Poggio. Videorealistic talking faces: A morphing approach. In *Proceedings of the AVSP '97 Workshop, Rhodes, Greece*, September 1997.

[34] Tony F. Ezzat and Tomaso Poggio. Miketalk: A talking facial display based on morphing visemes. In *Computer Animation Conference, Philadelphia*, June 1998.

[35] O. Farooq and S. Datta. Phoneme recognition using wavelet based features. *Information Sciences*, 150:5–15, April 2001.

[36] The fft demystified. http://www.eptools.com/tn/T0001/INDEX.HTM, 1999.

[37] Marcus Fillipsson. Speech analysis tutorial. http://www.ling.lu.se/research/speechtutorial/tutorial.html, 1995.

[38] Tom Forsyth. Self-shadowing bump map using 3d texture hardware. *J. Graph. Tools*, 7(4):19–26, 2002.

[39] Matteo Frigo and Steven G. Johnson. Fastest fourier transform in the west. http://www.fftw.org/, 2005.

[40] Patricia Galvis-Assmus, editor. *Computer Graphics*, volume 37(4). ACM Siggraph, November 2003.

[41] Peter De Gersem, Bart De Moor, and Marc Moonen. Applications of wavelets in audio and computer music. http://www.esat.kuleuven.ac.be/sista/yearreport96/node22.html, March 1997.

[42] Amara Graps. An introduction to wavelets. *IEEE Computational Science and Engineering*, 2(2), Summer 1995.

[43] Maya Gupta and Anna Gilbert. Robust speech recognition using wavelet coefficient features. In *ASRU*, 2001.

[44] R. Gutierrez-Osuna, P. Kakumanu, A. Esposito, O. N. Garcia, A. Bojorquez, J. Castillo, and I. Rudomin. Speech-driven facial animation with realistic dynamics. In *IEEE Transactions on Multimedia*, 2003.

[45] Paul Heckbert. Fourier transforms and the fft algorithm. http://www-2.cs.cmu.edu/afs/cs/project/anim/ph/463.95/pub/www/ps/fourier.ps, January 1998.

[46] Forrest Hoffman. An introduction to fourier theory. http://gershwin.ens.fr/vdaniel/Doc-Locale/Cours-Mirrored/Maths-Stuff/fourier/index.ps, 2004.

[47] Evans M. Harrell II and James V. Herod. Linear methods of applied mathematics. http://www.mathphysics.com/pde/, September 1997.

[48] JR. Joseph P. Campbell. Speaker recognition: A tutorial. In *Proceedings of the IEEE*, volume 85 no. 9, September 1997.

[49] Pushkar Joshi, Wen C. Tien, Mathieu Desbrun, and Frédéric Pighin. Learning controls for blend shape based realistic facial animation. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 187–192. Eurographics Association, 2003.

[50] Kolja Kähler, Jörg Haber, and Hans-Peter Seidel. Geometry-based muscle modeling for facial animation. In *No description on Graphics interface 2001*, pages 37–46. Canadian Information Processing Society, 2001.

[51] Ian Kaplan. The daubechies d4 wavelet transform. http://www.bearcave.com/misl/misl_tech/wavelets/daubechies/, July 2001.

[52] Eric Keller. Signalyze. http://www.signalyze.com, February 2002.

[53] Tony Kobayashi. Using recorded motion for facial animation. Master's thesis, University of British Columbia, April 1994. http://www.cs.ubc.ca/labs/imager/th/pdf/kobayashi.msc.1997.pdf.

[54] University of Colorado at Boulder Kristian Sandberg, Dept. of Applied Mathematics. The daubechies wavelet transform. http://amath.colorado.edu/courses/4720/2000Spr/Labs/DB/db.html, April 2000.

[55] Feng Liu, G. Scott Owen, and Ying Zhu. Universal converter for platform-independent procedural shaders in x3d. In *Siggraph*, August 2004.

[56] Zicheng Liu, Zhengyou Zhang, Chuck Jacobs, and Michael Cohen. Rapid modeling of animated faces from video. Technical Report MSR-TR-2000-11, Microsoft Research, Microsoft Corporation, February 2000.

[57] C.J. Long and S. Datta. Wavelet based feature extraction for phoneme recogition. In *Proc. ICSLP '96*, volume 1, pages 264–267, Philadelphia, PA, 1996.

[58] Lonnie C. Ludeman. *Fundamentals of Digital Signal Processing*, chapter 4.6, pages pp196–197. John Wiley and Sons, New York, 1986.

[59] William R. Mark, R. Stephen Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics hardware in a c-like language. *ACM Trans, Graph*, 22(3):896–907, 2003.

[60] Liset web site. http://www.microsoft.com/msagent/default.asp, April 2003.

[61] Bartosz Milewski. The fourier transform. http://www.relisoft.com/Science/Physics/sound.html, 2004.

[62] Tom Molet, Zhiyong Huang, Ronan Boulic, and Daniel Thalmann. An animation interface designed for motion capture. In *Computer Animation '97 Conference, Geneva, Switzerland*, pages 77–85. IEEE Press, 1997.

[63] Thomas Akenine Möller and Eric Haines. *Real Time Rendering, 2nd edition*. AK Peters Ltd, 2002.

[64] Nils J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kauffman Publishers Inc., 1998.

[65] Christophe Pallier, Laura Bosch, and Nuria Sebastian-Gallés. A limit on behavioral plasticity in speech perception. *Cognition*, 64(3):B9–B17, 1997.

[66] Frederic Parke and Keith Waters. *Computer Facial Animation*. A.K. Peters, 1996.

[67] Catherine Pelaud, Norman I. Badler, and Mark Steedman. Linguistic issues in facial animation. In N. Magnenat-Thalmann and D. Thalmann, editors, *Computer Animation '91*, pages 15–30. Springer-Verlag, 1991.

[68] Vikram Pudi. Neural networks. http://www.iiit.net/ vikram/nn_intro.html, 2004.

[69] Hyewon Pyun, Yejin Kim, Wonseok Chae, Hyung Woo Kang, and Sung Yong Shin. An example-based approach for facial expression cloning. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 167–176. Eurographics Association, 2003.

[70] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In A. Waibel and K.-F. Lee, editors, *Readings in Speech Recognition*, pages 267–296. Kaufmann, San Mateo, CA, 1990.

[71] David Rehagen and Rebecca Kirk. The gnnv project tutorial. www.iwu.edu/ shelley/gnnv/tutorial.html, 2004.

[72] Martin Riedmiller. Advanced supervised learning in multi-layer perceptrons - from backpropagation to adaptive learning algorithms. *Int. Journal of Computer Standards and Interfaces - Special Issue on Neural Networks*, 5, 1994.

[73] Dr Iain A Robin. Digital signal processing tutorial. http://www.dsptutor.freeuk.com/index.htm, 2004.

[74] Philip Rubin and Eric Vatikiotis-Bateson. Talking heads. In *Auditory-Visual Speech Processing (AVSP'98)*, December 1998.

[75] Holly Rushmeier, Gabriel Taubin, and André Guéziec. Applying shape from lighting variation to bump map capture. In *Eurographics Rendering Workshop Proceedings*, 1997.

[76] J. Salomon, K. Simon, and M. Osborne. Framewise phone classification using support vector machines. In *7th International Conference on Spoken Language Processing*, pages 2645–2648, 2002.

[77] Ruhi Sarikaya and John H. L. Hansen. Analysis of the root-cepstrum for aucoustic modeling and fast decoding in speech recognition. In *Eurospeech*, September 2001.

[78] B.G. Sherlock and D.M. Monro. Optimized wavelets for fingerprint compression. In *International Conference on Accoustic, Speech and Signal Processing (ICASSP '96, Atlanta, Georgia)*, volume III, pages 1447–1450, May 1996.

[79] Karan Singh and Evangelos Kokkevis. Skinning characters using surface-oriented free-form deformations. In *Proceedings of the Graphics Interface*, pages 35–42, May 2000.

[80] Stephen W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing 2nd Edition*. California Technical Publishing, 1999.

[81] Mark Stamp. A revealing introduction to hidden markov models. http://www.cs.sjsu.edu/faculty/stamp/RUA/HMM.pdf, 2004.

[82] Gilbert Strang. Wavelets. In *American Scientist*, volume 82, pages 250–255, April 1994.

[83] Wim Sweldens and Peter Schröder. Building your own wavelets at home. http://cm.bell-labs.com/who/wim/papers/athome/athome.pdf, 1996.

[84] Beng T. Tan, Minyue Fu, Andrew Spray, and Phillip Dermody. The use of wavelet transforms in phoneme recognition. In *The Fourth International Conference on Spoken Language Processing (ICSLP) , Philadelphia*, October 1996.

[85] Yu Tao, Ernest C.M. Lam, and Yuan Y. Tang. Feature extraction using wavelet and fractal. *Pattern Recognition Letters*, 22:271–287, November 2001.

[86] Marco Tarini, Hitoshi Yamauchi, Jörg Haber, and Hans-Peter Seidel. Texturing Faces. In *Proc. Graphics Interface*, pages 89–98, May 2002.

[87] Asterios Toutios and K. G. Margaritis. Development of a text dependent speaker identification system with the ogi toolkit. In *2nd Hellenic Conference on AI, SETN*, pages 525–530, 2002.

[88] J.W. Tukey, B. P. Bogert, and M. J. R. Healy. The quefrency alanysis of time series for echoes: cepstrum, pseudo-autocovariance, cross-cepstrum, and saphe-cracking. In *Proceedings of the Symposium on Time Series Analysis*, 1963.

[89] Marting Vetterli and Jelena Kovačevicć. *Wavelets and Subband Coding*. Prentice-Hall PTR, 1995.

[90] Alex Vlachos, Jörg Peters, Chas Boyd, and Jason L. Mitchell. Curved pn triangles. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 159–166. ACM Press, 2001.

[91] Pascal Volino and Nadia Magnenat Thalmann. Fast geometric wrinkles on animated surfaces. In *WSCG*, October 1999.

[92] A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K.J. Lang. Phoneme recognition using time-delay neural networks. In *IEEE Transactions on Accoustic, Speech and Signal Processing ASSP-37*, 1989.

[93] Keith Waters. A muscle model for animating three-dimensional facial expression. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 17–24. ACM Press, 1987.

[94] George M. White. Speech recognition: A tutorial overview. *Computer*, 9:pp. 40–53, May 1976.

[95] Jin xiang Chai, Jing Xiao, and Jessica Hodgins. Vision-based control of 3d facial animation. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 193–206. Eurographics Association, 2003.

[96] Shin Yoshizawa, Alexander G. Belyaev, and Hans-Peter Seidel. Free-form skeleton-driven mesh deformations. In *Proceedings of the eighth ACM symposium on Solid modeling and applications*, pages 247–253. ACM Press, 2003.

[97] Steve Young, Gunnar Evermann, Thomas Hain, Dan Kershaw, Gareth Moore, Julian Odell, Dave Ollason, Dan Povey, Valtcho Valtchev, and Phil Woodland. The htk book, 2002.

[98] Qingshan Zhang, Zicheng Liu, Baining Guo, and Harry Shum. Geometry-driven photorealistic facial expression synthesis. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 177–186. Eurographics Association, 2003.

[99] Li Zuo, Jin tao Li, and Zhao qi Wang. Anatomical human musculature modeling for real-time deformation. In *Winter School of Graphics Computing (WSGC)*, 2003.